Portable Systems Group

Windows NT Alerts Design Note

Author: *David N. Cutler*

Original Draft 1.0, February 9, 1989
Revision 1.2, March 30, 1989

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

This design note discusses a proposal to implement alerts in both kernel and user mode. The alert capability can be used to interrupt thread execution in either processor mode at well defined points. A companion design note on **APC**'s contains information and algorithms that are pertinent to this design.

There are three alert specific kernel services; *TestAlertThread*, *AlertThread*, and *AlertResumeThread*. In addition, the kernel *Wait* functions take a mode and an alertable flag as arguments.

Each thread has an alerted flag for each of the processor modes user and kernel. These flags are set by calling the AlertThread function and specifying the thread and the mode which are to be alerted.

If AlertThread is called and the target thread is in a wait state, then several additional tests are performed to determine the correct action to take.

If the mode of the wait is user, the alertable flag is set, and the alert mode is user, then a thread specific **APC** is queued to user mode which will raise the condition "alerted", the user **APC** pending flag is set, and the thread is unwaited with a completion status of "alerted".

If the mode of the wait is kernel or user, the alertable flag is set, and the alert mode is kernel, then the thread is unwaited with a status of "alerted". There is no **APC** queued for kernel mode.

The following pseudo code describes the logic of AlertThread:

```
PROCEDURE AlertThread (
     IN Mode : KtProcessorMode;
     IN Tcb : POINTER KtThread;
     );

BEGIN

     Acquire dispatcher database lock;
     IF Tcb.State == Waiting THEN
          IF Tcb.WaitMode >= Mode AND Tcb.Alertable THEN
               IF Mode == User THEN
                    Queue Tcb.AlertAcb;
                    Tcb.UserApcPending = True;
               END IF;
               Unwait thread with a status of Alerted;
          ELSE
               Tcb.Alerted[Mode] = True;
```

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

```
          END IF;
     ELSE
          Tcb.Alerted[Mode] = True;
     END IF;
     Release dispatcher database lock;
END AlertThread;
```

When the user mode alerted flag gets set, it remains set until either a TestAlert or a Wait alertable is performed which clears the flag.

The kernel mode alerted flag is treated somewhat differently in that it is cleared on each system service entry to the system. The reasoning behind this is that a kernel mode alert should only persist for the duration of time that execution continues in kernel mode. As soon as execution leaves kernel mode, the alerted flag is no longer significant. This is a very important feature which allows the conditional aborting of native system services by protected subsystems which provide system services for other operating system **API's**. This subject is discussed in more detail at the end of this document.

The kernel service *AlertResumeThread* allows a thread to be alerted and then resumed in a single operation. This operation is really a kernel mode AlertThread followed by a ResumeThread, but is provided as a kernel service so that is can be executed without any race conditions.

The following pseudo code describes the logic of AlertResumeThread:

```
PROCEDURE AlertResumeThread (
     IN Tcb : POINTER KtThread;
     ) RETURNS integer;

VARIABLE

     OldCount : integer;

BEGIN

     Acquire dispatcher database lock;
     IF Tcb.State == Waiting THEN
          IF Tcb.Alertable THEN
               Unwait thread with a status of Alerted;
          ELSE
               Tcb.Alerted[Kernel] = True;
          END IF;
     ELSE
          Tcb.Alerted[kernel] = True;
```

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

```
     END IF;
     OldCount = Tcb.SuspendCount;
     IF Tcb.SuspendCount <> 0 THEN
          Tcb.SuspendCount = Tcb.SuspendCount - 1;
          IF Tcb.SuspendCount == 0 THEN
               Release Tcb.SuspendSemaphore;
          END IF;
     END IF;
     Release dispatcher database lock;
     RETURN OldCount;
END AlertResumeThread;
```

TestAlertThread tests the alerted flag for a specified processor mode and returns a status value of "alerted" if the flag was set and "normal" if the flag was clear. If the alerted flag was set, then it is cleared, and if the specified mode is user, then an alert **APC** is queued to user mode and user **APC** pending is set in the calling thread's **TCB**.

In addition, TestAlertThread also tests whether a user **APC** should be delivered. If the specified mode is user and the user **APC** queue contains an entry, then **APC** pending is set in the calling thread's **TCB**.

The following pseudo code describes the logic of TestAlert:

```
PROCEDURE TestAlertThread (
     IN Mode : KtProcessorMode;
     ) RETURNS KtStatus;

BEGIN

     Acquire dispatcher database lock;
     Get current TCB address;
     IF Tcb.Alerted[Mode] THEN
          Tcb.Alerted[Mode] = False;
          IF Mode == User THEN
               Queue Tcb.AlertAcb;
               Tcb.UserApcPending = True;
          END IF;
          Release dispatcher database lock;
          RETURN Alerted;
     ELSE
          IF Mode == User AND Tcb.ApcQueue[User] <> NIL THEN
               Tcb.UserApcPending = True;
          END IF;
          Release dispatcher database lock;
          RETURN Normal;
     END IF;
END TestAlertThread;
```

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

Wait tests the alerted flags for the specified and all more privileged processor modes if the alertable argument value is true. If an alerted flag is set, then a status value of "alerted" is returned.

In addition, Wait also tests whether a user **APC** should be delivered if the alertable argument value is true and the specified mode is user. For this case, if the user **APC** queue contains an entry, then **APC** pending is set in the calling thread's **TCB** and a status value of "UserApc" is returned.

The following pseudo code describes the logic of Wait:

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

```
PROCEDURE Wait (
     IN Mode : KtProcessorMode;
     IN Alertable : boolean;
     IN WaitObject : POINTER KtDispatcherObject;
     IN Timeout : POINTER integer;
     ) RETURNS KTStatus;

BEGIN

Repeat:
     Acquire dispatcher database lock;
     Get current TCB address;
     IF Alertable THEN
          IF Tcb.Alerted[Mode] THEN
               Tcb.Alerted[Mode] = False;
               IF Mode == User THEN
                    Queue Tcb.AlertAcb;
                    Tcb.UserApcPending = True;
               END IF;
               Release dispatcher database lock;
               RETURN Alerted;
          ELSEIF Mode == User THEN
               IF Tcb.UserApcQueue <> NIL THEN
                    Tcb.UserApcPending = True;
                    Release dispatcher database lock;
                    RETURN UserApc;
               ELSEIF Tcb.Alerted[Kernel] THEN
                    Tcb.Alerted[Kernel] = False;
                    Release dispatcher database lock;
                    RETURN Alerted;
               END IF;
          END IF;
     END IF;
     IF WaitObject.Signal THEN
          Satisfy wait for WaitObject;
          Release dispatcher database lock;
          RETURN Tcb.WaitStatus;
     ELSE
          Tcb.Alertable = Alertable;
          Construct wait control block for WaitObject;
          Initialize Tcb.Timer with time out value;
          Insert wait control block in wait queue;
          Insert Tcb.Timer in timer queue;
          Select new thread to run;
          Swap context to new thread;
          IF Tcb.WaitStatus == KernelApc THEN
               Goto Repeat;
          ELSE
               RETURN Tcb.WaitStatus;
          END IF;
     END IF;
END Wait;
```

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

It is the responsibility of the executive to test for the "alerted" return status from TestAlert and Wait and perform the correct operation (e.g. cleaning up data structure, unwinding, etc).

Wait and AlertThread both allow a thread that is waiting user mode alertable to be awakened by a kernel mode alert. If this were not done, then it would not be possible to abort the Wait system service.

The interesting combinations of initial conditions and the resultant action when a Wait system service is executed are given below.

Case 1

```
Wait Mode = Kernel
Tcb.Alerted[User] = True
Tcb.Alerted[Kernel] = False
Alertable = True

Action - Put thread in wait state
```

Case 2

```
Wait Mode = Kernel
Tcb.Alerted[User] = x
Tcb.Alerted[Kernel] = True
Alertable = True

Action - Clear Tcb.Alerted[Kernel] and return Alerted
```

Case

```
Wait Mode = User
Tcb.Alerted[User] = True
Tcb.Alerted[Kernel] = x
Alertable = True

Action - Clear Tcb.Alerted[User], queue Tcb.AlertAcb,
     and set Tcb.UserApcPending
```

Case 4

```
Wait Mode = User
Tcb.Alerted[User] = False
Tcb.Alerted[Kernel] = True
Alertable = True

Action - Clear Tcb.Alerted[Kernel] and return Alerted
```

Case 5

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

```
Wait Mode = User
Tcb.Alerted[User] = False
Tcb.Alerted[Kernel] = False
Alertable = True

Action - Put thread in wait state
```

Kernel mode alerts can be used to implement the semantics necessary to abort native system services. The following discussion describes how this can be implemented in **Windows NT**.

In **Mach** the operations necessary to abort a native system service are suspend, abort service, and resume. This capability is used to get a thread out of a possible wait state in the system and deliver a signal, terminate execution, etc.

A similar set of primitives can be provided in **Windows NT** using the kernel alert capability.

Windows NT suspends a thread by sending it a normal kernel **APC** that causes the thread to wait on an semaphore that is built into the thread object. The resume operation simply releases the builtin semaphore which continues thread execution.

The suspend wait operation is nonalertable to ensure that the alert and resume operation functions properly; see below.

If a thread is in a wait state when it is suspended, then the wait completion status is set to "kernel **APC**". This is done so the wait can be repeated when the **APC** returns.

Implementing the primitives to abort native system services does not quite solve the whole problem. Each native service that can result in a long wait must be written such that it is responsive to kernel alerts. This means that a native service should wait alertable in kernel mode when it does a wait that could take a long time. Also, if very long algorithms are being performed, then TestAlert should also be called at appropriate points.

It is preferable that a native service either complete successfully or be entirely aborted. For those cases where there are really two parts to the service such as an operation followed by a wait, the service should be broken into two parts. Each part should be executed separately from the calling mode.

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

A protected subsystem that is a system service server can stop, alter, and a resume a thread by performing the sequence of operations suspend, get state, set state, and alert and resume.

If a native service is active when the suspend operation takes place, then the kernel alerted flag will remain set for the duration of the service after the thread is resumed. The alerted flag can be tested by the service using the TestAlert function.

A more interesting case is when the native service is waiting kernel mode alertable. The suspend service causes a normal kernel **APC** to be sent to the target thread which completes its wait with a status of "kernel **APC**". The target thread then waits nonalertable on its builtin suspend semaphore.

When the subsystem executes the alert and resume service, the kernel alerted flag is set in the target thread and the target thread's suspend semaphore is released. This causes the target thread to be unwaited with a status that is the key value of the semaphore.

Unwaiting the thread causes it to continue execution in the suspend **APC** routine which simply returns to the kernel **APC** delivery code. The kernel **APC** delivery code restores the state of the thread and resumes execution at the point of interruption which is in the wait code. The wait code tests the wait completion status and determines that the wait was satisfied to deliver a kernel **APC**. The wait is repeated and finds that the kernel alerted flag is set and that the wait is alertable. Thus it returns immediately with a wait completion status of "alerted".

Note that the kernel **APC** delivery code must save and restore the wait completion status in the **TCB** so that the subsequent suspend wait does not destroy it.

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, February 9, 1989

Revision 1.1, February 10, 1989

1. Include tests for nonempty user **APC** queue in TestAlert and Wait algorithm descriptions.

Revision 1.2, March 30, 1989

1. Minor edits to conform to standard format.

[end of alerts]

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT APC Design Note

Author: *David N. Cutler*

Original Draft 1.0, February 6, 1989
Revision 1.2, March 30, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following design note describes a proposal for the handling of **APC**'s in **Windows NT**. The companion design notes on alerts and attach process contain information and algorithms that are pertinent to this design.

The nice thing about **APC**'s is that they interrupt thread execution at any point and cause a procedure to be executed in the context of a specified thread. This capability can be used to reduce the number of threads required to perform a particular function and can alleviate the need for polling.

The new model for implementing **OS/2** and **POSIX** compatibility with protected subsystems would suggest that **APC**'s could be used to substantially reduce the overhead and implementation complexity of these subsystems. For instance **OS/2** timers could be implemented by **NT** timers that queue an **APC** when they expire. The **APC** would be fielded by the **OS/2** subsystem which would clear the appropriate semaphore and delete or repeat the timer as appropriate.

As good as this all sounds it is not without flaw. The very thing that makes **APC**'s so useful is also the same thing that makes them so bad. This is the fact that they interrupt a thread at arbitrary points. To get past this liability, the capability to "disable" **APC**'s over short regions of code is needed. But this then has the problem of not being very modular and also requires a lot of thought on the part of the user. Writing code that is "**APC**" safe is VERY difficult.

SRC never recognized the need for **APC**'s but did recognize that it was useful to be able to send a thread an alert signal. This signal typically means quit what you are doing and reset to some canonical state. **SRC's** system provides a function to send an alert to a thread (*AlertThread*), a function to test if a thread had been alerted (*TestAlert*), and a form of wait that allows a thread to be alerted while it is waiting (*WaitAlertable*).

When TestAlert or WaitAlertable is called and the subject thread has been alerted, then the condition "alert" is raised. In addition, if AlertThread is called while a thread is waiting as the result of a call to WaitAlertable, then the thread is unwaited and the "alert" condition is raised.

The nice thing about the **SRC** alert design is that the alert condition occurs at well defined points in the execution of a program. These points are exactly the points where the program says it is alertable. Writing code that is "alert" safe is easy.

We do not want to drop the flexibility of **APC**'s, but at the same time we do not want to interrupt the execution of a thread at arbitrary points. Therefore why not combine the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

notion of alertable with the functionality of **APC**'s? To do this we simply do not deliver an **APC** unless the thread is alertable or calls TestAlert.

We only need to do this for user mode, and it fact, do not want to do this for kernel mode as we need to break into the kernel mode execution of a thread at an arbitrary point. As system designers this does not (or more succinctly better not!) present us with the same level of difficulty that it does the run of the mill user.

Thus in user mode, **APC**'s are only delivered at points where the program is alertable. In kernel mode **APC**'s are delivered when the appropriate enabling conditions are present.

The following is an explantion of how **APC**'s would work using the concepts described above.

There are three types of **APC**'s:

1. special kernel
2. normal kernel
3. normal user

A special kernel **APC** is deliverable whenever the Interrupt Request Level (**IRQL**) of the corresponding thread is equal to zero, and executes in kernel mode at **IRQL** 1. This type of **APC** is used to break into a thread's execution and perform some short operation such as posting **I/O** status. Code that runs as the result of a special kernel **APC** is not allowed to acquire any mutexes that can also be acquired at **IRQL** 0. Special kernel **APC** code is allowed to take page faults, and thus memory management code must ensure that it runs at **IRQL** 1 when it owns a mutex that could also be acquired during a special kernel **APC**.

A normal kernel **APC** is deliverable whenever the **IRQL** of the corresponding thread is equal to zero, a normal kernel **APC** is not already in progress, and the thread does not own any kernel level mutexes. Normal kernel **APC** code executes at **IRQL** 0 and is allowed to execute any code including all system services.

A normal user **APC** is deliverable at any time the target thread is user mode alertable. Normal user **APC** code executes at **IRQL** 0 and is allowed to execute any code including all system services.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Both normal kernel and normal user **APC's** can also specify a routine that is to be executed in kernel mode at **IRQL** 1 just prior to executing the normal **APC** routine.

Each thread has a machine state which includes **IRQL**, an **APC** pending flag for each of the modes kernel and user, an **APC** in progress flag for kernel mode, and the number of mutexes that are owned in kernel mode. This state is used to determine when an **APC** should be delivered to a thread.

Unlike **VAX** or **PRISM**, there is no hardware support for **APC**'s. Thus at each exit from kernel mode (i.e. on each **REI** type of operation), appropriate tests must be made to determine whether an **APC** should be delivered or not.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following pseudo code describes the logic of system exit:

```
ExitFromSystem:

     disable interrupts;
     IF Previous IRQL == 0 THEN
          Get current TCB address;
          IF Previous mode == Kernel THEN
               IF Tcb.KernelApcPending THEN
                    IRQL = 1;
                    Call kernel APC delivery code;
               END IF;
          ELSEIF Tcb.UserApcPending THEN
               IRQL = 1;
               Call user APC delivery code;
          END IF;
     END IF;
     Restore state and continue execution;
```

The user **APC** delivery code is only called when an **APC** can actually be delivered to user mode. Calling the kernel **APC** delivery code, however, does not guarantee that a kernel **APC** can really be delivered. Further checks must be performed to ensure that proper enabling conditions are present. These tests include whether the thread currently owns any mutexes and whether a normal kernel **APC** is already in progress.

A thread in **Windows NT** can be in one of six states:

1. initialized - the thread has been initialized but has not been readied for execution.
2. running - the thread is currently in execution on some processor.

3. ready - the thread is either in a processor ready queue (i.e. ready to execute) or in a process ready queue (i.e. process is not in balance set).

4. standby - the thread has been selected to run on a processor but has not actually started its execution.

5. terminated - the thread has terminated but has not yet been rundown (e.g. all resources have not been returned).

6. waiting - the thread is waiting on one or more dispatcher objects to attain a state of signaled.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

When an **APC** is queued, certain tests must be performed to determine what action if any should be taken.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following pseudo code describes the logic of queuing an **APC**:

```
PROCEDURE QueueApc (
     IN Acb : POINTER KtApc;
     IN Tcb : POINTER KtThread;
     );

BEGIN

     IF Acb.Mode == Kernel THEN
          IF Acb.Type == Special THEN
               Insert APC at front of thread kernel APC
                    queue selected by Acb.ApcIndex;
          ELSE
               Insert APC at end of thread kernel APC queue
                    selected by Acb.ApcIndex;
          END IF;
          IF Tcb.State == Running AND
             Acb.ApcIndex == Tcb.ApcIndex THEN
               IF Tcb.NextProcessor == CurrentProcessor THEN
                    Set software interrupt at IRQL 1;
               ELSE
                    Set APC delivery request for target
                         processor;
                    Set interrupt request for target
                         processor;
               END IF;
          ELSEIF (Tcb.State == Waiting AND
                 Acb.ApcIndex == Tcb.ApcIndex AND
                 Tcb.WaitIrql == 0) AND
                 (Acb.Type == Special OR
                 (Tcb.MutexCount == 0 AND
                      NOT Tcb.KernelApcInProgress)) THEN
                    Unwait thread with status of KernelApc;
          END IF;
          Tcb.KernelApcPending = True;
     ELSE
          Insert APC at the end of thread user APC queue
               selected by Acb.ApcIndex;
          IF Tcb.State == Waiting AND
             Acb.ApcIndex == Tcb.ApcIndex AND
             Tcb.WaitMode == User AND
             Tcb.Alertable THEN
               Tcb.UserApcPending = True;
               Unwait thread with status of Alerted;
          END IF;
     END IF;
END QueueApc;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

A thread may be unwaited to execute a special kernel, normal kernel, or normal user **APC**.

If the **APC** executes in kernel mode then the **APC** will have already been executed by the time that execution continues in the wait code. For this case the wait function is merely repeated.

If the **APC** executes in user mode, then execution continues in the wait code without having deliverd the user **APC**. For this case, the wait code simply returns the status "alerted" to the executive level Wait routine. The executive level Wait routine must return a status of "RepeatService" to the system service dispatch. The system service dispatcher backs up the **PC** so that the wait service will be repeated, restores state as necessary, and then executes the "**REI**" which will cause a user **APC** to occur.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Original Draft 1.0, February 6, 1989

Revision 1.1, February 10, 1989

1. Move alert algorithms to alert design note.
2. Add test for attached process in QueueApc procedure.
3. Add software interrupt request when **APC** is queued to the current processor in kernel mode.
4. Correct algorithm for delivery of user **APC**.

Revision 1.2, March 30, 1989

1. Minor edits to confrom to standard format.
2. Add capability to receive **APC**'s while attached to another address space.

[end of apc.doc]

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Argument Validation Specification

Author: *David N. Cutler*

Original Draft, May 4, 1989
Revision 1.1, May 5, 1989
Revision 1.2, May 10, 1989
Revision 1.3, July 15, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview 1

2. Requirements 1

3. Operation 1

4. Interfaces 3
   4.1 Probe for Readability and Read Argument Value 3
   4.2 Probe for Writeability and Read Argument Value 4
   4.3 Probe for Writeability and Read/Write Argument Value 5
   4.4 Probing An Aggregate Value 7

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This document describes the argument probing and capture requirements to which all system services must adhere.

System services must be written such that they are robust and provide protection against malicious attack and inadvertent program bugs. It must not be possible to crash or corrupt the system by passing an invalid argument value, a pointer to memory that is not accessible to the caller, or by dynamically altering or deleting the memory occupied by an argument in a simultaneously executing thread.

2. Requirements

Every system service must ensure that the arguments on which it operates are valid (i.e., values are correct). This is essential to robust system operation and involves the capturing of values and the probing of argument addresses at appropriate points.

In general, a system service should capture all arguments on entry to the procedure. This ensures that the caller or one of its cohorts (buddy threads) cannot dynamically alter the value of the argument after it has been read and verified, or delete the memory in which it is contained.

In some cases, it is not necessary to capture the value of an argument immediately. Such is the case for I/O buffers and name strings. However, all pointers **MUST** be captured and the addresses to which they point **MUST** be probed for accessibility.

Fortunately, most arguments do not need explicit capture since they are passed in registers. Arguments that are passed in memory are probed and captured by the system service dispatcher as necessary.

3. Operation

The address space layout of **Windows NT** contains a boundary that delineates user address space from system address space. All addresses above the boundary are considered system addresses and all addresses below the boundary are considered user addresses.

Pages in the system part of the address space are owned by kernel mode and are not accessible to the user unless they are double mapped into the user part of the address space. Pages in the user part of the address space are owned by user mode and the access for kernel mode is identical to that for user mode.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The executive **NEVER** creates a page in the user part of the address space that is owned by kernel mode. Furthermore, at the boundary between user address space and system address space, there are **64K** bytes that are inaccessible to all modes. This address space layout makes it possible to determine whether an address is a valid user address simply by doing a boundary comparison.

When a system service is called, the trap handler gets control, saves state, and transfers control to the system service dispatcher. The system service dispatcher determines which system service is being called, and obtains the address of the appropriate function and the number of in-memory arguments from a dispatch table. If the previous processor mode is user mode and there is one or more in-memory arguments, then the in-memory argument list is probed and then copied to the kernel stack. If an access violation occurs during the copy, then the system service is completed with a status of access violation. If an access violation does not occur, then the the pointer to the in-memory argument list is changed to point to the copy of the arguments on the kernel stack. The system service dispatcher sets up a catchall condition handler, and then calls the system service function.

The first thing the system service should do is establish a condition handler. This handler should be prepared to handle access violations that may occur as argument pointers are dereferenced to read or write actual argument values.

Next, the system service code should obtain the previous processor mode. If the previous processor mode was kernel, then there is no need to probe any arguments. The executive does not call itself with bad arguments.

If the previous processor mode was user, then any argument values that are read or written by dereferencing a pointer must be probed for accessibility. Probing is accomplished by first ensuring that the address of the variable is within the user's address space and then reading or writing the variable as appropriate. The code that actually probes pointer-related arguments does not set up a condition handler. It merely does the boundary check and then reads or writes the indicated location. If the boundary check fails, an access violation condition is raised. If the memory is inaccessible, an access violation is raised by hardware. Thus probes are extremely cheap.

The complete code at the beginning of a system service should be constructed as follows:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
// set up condition handler to catch access violations
            .
            .
            .

if (GetPreviousMode() != KernelMode) {
            .
            .
            .
      // probe and capture reference arguments
            .
            .
            .
}
```

At this point in the execution of a system service, all input values have been captured and all output variables have been probed for writeability. The system service performs its function, writes output values as necessary, and returns a status that indicates whether the service succeeded or failed.

During the writing of output values, an access violation can occur because another thread or user altered the address space of the calling thread. Access violations that occur at this time are silent and do not cause the service to fail. If this were not the case, then it would be very difficult to actually complete a system service since code would have to be added to back out and undo the service right up until the very last output value is written. If the caller receives a success status under such conditions, it is likely that the caller will attempt to access one of the output values and get an access violation.

4. Interfaces

The following sections describe the interfaces that are provided to probe arguments for read and write accessibility.

4.1 Probe for Readability and Read Argument Value

The following functions provide the capability to probe a primitive data type for readability and to read an argument value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
CHAR
ProbeAndReadChar (
    IN PCHAR Address
    );

UCHAR
ProbeAndReadUchar (
    IN PUCHAR Address
    );

SHORT
ProbeAndReadShort (
    IN PSHORT Address
    );

USHORT
ProbeAndReadUshort (
    IN PUSHORT Address
    );

LONG
ProbeAndReadLong (
    IN PLONG Address
    );

ULONG
ProbeAndReadUlong (
    IN PULONG Address
    );

QUAD
ProbeAndReadQuad (
    IN PQUAD Address
    );

UQUAD
ProbeAndReadUquad (
    IN PUQUAD Address
    );
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
HANDLE
ProbeAndReadHandle (
    IN PHANDLE Address
    );

BOOLEAN
ProbeAndReadBoolean (
    IN PBOOLEAN Address
    );
```

The previous functions are used to probe and read a value pointed to by a safe pointer. A safe pointer is one that has either been captured on procedure entry or which has been previously captured with one of the these functions. The functions compare the pointer value to the user/system address boundary, read the appropriate data-type value, and return the value as the function value. If the value is not of consequence, then the function value is simply not assigned to a variable. Note that both signed and unsigned data types are provided.

4.2 Probe for Writeability and Read Argument Value

The following functions provide the capability to probe a primitive data type for writeability and read an argument value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
CHAR
ProbeForWriteChar (
    IN PCHAR Address
    );

UCHAR
ProbeForWriteUchar (
    IN PUCHAR Address
    );

SHORT
ProbeForWriteShort (
    IN PSHORT Address
    );

USHORT
ProbeForWriteUshort (
    IN PUSHORT Address
    );

LONG
ProbeForWriteLong (
    IN PLONG Address
    );

ULONG
ProbeForWriteUlong (
    IN PULONG Address
    );
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
QUAD
ProbeForWriteQuad (
    IN PQUAD Address
    );

UQUAD
ProbeForWriteUquad (
    IN PUQUAD Address
    );

HANDLE
ProbeForWriteHandle (
    IN PHANDLE Address
    );

BOOLEAN
ProbeForWriteBoolean (
    IN PBOOLEAN Address
    );
```

The previous functions are used to probe for writeability and read a value pointed to by a safe pointer. A safe pointer is one that has either been captured on procedure entry or which has been previously captured with one of these functions. The functions compare the pointer value to the user/system address boundary, read the appropriate data type value, write the value that was read back into memory, and return the original value as the function value. If the value is not of consequence, then the function value is simply not assigned to a variable. Note that both signed and unsigned data types are provided.

4.3 Probe for Writeability and Read/Write Argument Value

The following functions provide the capability to probe a primitive data type for writeability, read an argument value, and write a specified value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
CHAR
ProbeAndWriteChar (
    IN PCHAR Address,
    IN CHAR Value
    );

UCHAR
ProbeAndWriteUchar (
    IN PUCHAR Address,
    IN UCHAR Value
    );

SHORT
ProbeAndWriteShort (
    IN PSHORT Address,
    IN SHORT Value
    );
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
USHORT
ProbeAndWriteUshort (
    IN PUSHORT Address,
    IN USHORT Value
    );

LONG
ProbeAndWriteLong (
    IN PLONG Address,
    IN LONG Value
    );

ULONG
ProbeAndWriteUlong (
    IN PULONG Address,
    IN ULONG Value
    );

QUAD
ProbeAndWriteQuad (
    IN PQUAD Address,
    IN QUAD Value
    );

UQUAD
ProbeAndWriteUquad (
    IN PUQUAD Address,
    IN UQUAD Value
    );

HANDLE
ProbeAndWriteHandle (
    IN PHANDLE Address,
    IN HANDLE Value
    );

BOOLEAN
ProbeAndWriteBoolean (
    IN PBOOLEAN Address,
    IN BOOLEAN Value
    );
```

The previous functions are used to probe a primitive data type for writeability and read a value pointed to by a safe pointer. In addition, the value that is to be written is specified as an argument to the function. A

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

safe pointer is one that has either been captured on procedure entry or which has been previously captured with one of these functions. The functions compare the pointer value to the user/system address boundary, read the appropriate data-type value, write the specified value to memory, and return the original memory contents as the function value. If the value is not of consequence, then the function value is simply not assigned to a variable. Note that both signed and unsigned data types are provided.

4.4 Probing An Aggregate Value

The following functions provide the capability to probe aggregate data types (i.e., structures, arrays, strings, etc.) for read and write accessibility.

```
VOID
ProbeForRead (
    IN PCHAR Address,
    IN ULONG Length
    );

VOID
ProbeForWrite (
    IN PCHAR Address,
    IN ULONG Length
    );
```

The previous functions are used to probe an aggregate for read or write accessibility using a safe pointer. A safe pointer is one that has either been captured on procedure entry or which has been previously captured with one of the preceding functions. The functions compare the starting and ending addresses of the specified aggregate for read or write accessibility and then read or write one character from each page that is spanned by the aggregate. Note that these functions do not capture the aggregate value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, May 4, 1989

Revision 1.1, May 5, 1989

1. Add capturing of reference arguments to sample system service code.

2. Change data type definitions to make Portable System Group conventions.

Revision 1.2, May 10, 1989

1. Move the capturing and probing of the in-memory argument list into the system service dispatcher.

Revision 1.3, July 15, 1989

1. Add functions to probe handle and boolean values.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Attach Process Design Note

Author: *David N. Cutler*

Original Draft 1.0, February 8, 1989
Revision 1.2, March 30, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This design note discusses a proposal that would allow a thread to attach to the address space of another process, execute code in the attached process's address space, and then detach and resume execution in the original process address space. It is envisioned that this capability will be required to implement the newly proposed system structure.

This capability would not be exported to user mode at all. It is intended for internal use by the executive layer of the system.

The new system structure (i.e. system service servers) requires the ability to perform certain operations on behalf of another process. Typical of these operations is creating and deleting virtual memory. In order to implement these operations, we either have to build the data structures and algorithms such that they can be done outside the recipient process or architect a way to actually execute code within the address space of another process.

A good example of a difficult service to build outside of a process is the deletion of virtual memory. **Mach** stands on its head to implement this capability and, while it is doing such an operation, a global virtual memory lock must be held.

Graham Hamilton (of exDECwest fame) suggested that a way to do this was to have some number of anonymous system threads which could do such an operation. A requesting thread would build a request packet that contained the arguments of the operation to be performed, the function that was to be executed, a pointer to the address map that the thread was to execute in, and an event to synchronize the completion of the operation. The request packet would then be queued to the worker thread, a semaphore signaled, and the requesting thread would wait on the event. A worker thread would be awakened by the signal of the semaphore and would remove an entry from the request queue. The thread would attach to the new address space, perform the operation, set the event, detach from the address space, and then look for more work to do. The requesting thread would then resume execution.

In analyzing Graham's proposal it is clear that there are two extra context switches, a copy of the argument data, two extra translation buffer and data cache flushes, and the need to attach to an address space. So why not just let the requesting thread directly attach to the target process address space and avoid the worker threads, the argument copy, and the two extra context switches?

When a thread wanted to execute in another process' address space it would execute the following logic:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

verify that source process has the rights necessary to perform the desired operation on the destination process
obtain pointers to objects in the source process as necessary
KeAttachProcess(*pPcb*)
perform desired operation in address space of target process
KeDetachProcess()
resume execution is source process

There are several questions and complications that arise from doing this kind of operation. These include:

1. How is the kernel stack of the source thread addressed in the target process?

2. What happens if the source process gets removed from the balance set while an attach operation is in progress and causes the process' thread's kernel stacks to be made pageable?

3. What happens if the target process is not in the balance set?

4. What happens if the source or target processes are terminated?

5. What happens if the source thread is terminated?

6. What happens if a thread tries to do a second attach after having attached to a target process' address space?

7. What object table is visible when a thread is attached to the address space of another process?

8. What working set is manipulated while a thread is attached to the address space of another process?

9. What process gets charged for the time that is consumed while the thread is attached to another process' address space?

10. How is mutex ownership handled between the source and target processes?

11. What happens if user and/or kernel mode are alerted while a process is attached?

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

12. What happens to **APC**'s that are queued to the thread after it has entered the target process' address space?

13. Can the attached thread receive **APC**'s?

14. What happens if a suspend or resume is performed on the specified thread?

Before attempting to answer these questions it is useful to review the kernel data structures that correspond to process and thread objects. These data structures are described in more detail at the end of this note.

There is a Process Control Block (**PCB**) and a Thread Control Block (**TCB**).

A **PCB** contains a pointer to a process address map (actually the physical address of the Page Directory for the process), a list of all the **TCB**'s that are members of the **PCB**, a count of all the kernel mutexes owned by member **TCB**'s, and a state which is either "included" or "excluded" (corresponds to whether the process is, or is not, in the balance set).

A **TCB** contains a pointer to the **PCB** of which it is a member, an **APC** queue for each of the modes kernel and user, a kernel **APC** in progress flag, a kernel **APC** pending flag, a user **APC** pending flag, a user alert **APC** Control Block (**ACB**), an alerted flag for each of the modes kernel and user, an alertable wait flag, an owned mutex count, and link pointers for linking the thread into the **PCB**'s **TCB** list.

Actually there are several other fields in the **TCB** and **PCB**, but they are not really pertinent to this discussion.

The kernel data structures that describe the **TCB** and **PCB** are contained within the executive data structures that describe the process and thread objects. The executive must use the linkage structures provided by the kernel and cannot keep a separate set of linkage pointers that tie the data structures together.

The below discussion addresses the questioins raised above and gives an explanation of how **KeAttachProcess** and **KeDetachProcess** work.

How is the kernel stack of the source thread addressed in the target process?

We would like to make kernel stacks addressible in the process part of the address space. However, in order to attach to another process' address space we will need to map

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

kernel stacks in the system part of the address space so we can avoid an argument copy and allocation of a temporary kernel stack. If we do not do this, then we will have to allocate a temporary kernel stack in the system part of the address space, copy necessary argument information to the temporary stack, switch to the temporary stack, attach the target process' address space, execute the necessary logic, switch back to the source address space, switch back to the original stack, and then deallocate the temporary stack.

When a process is in the balance set the kernel stacks of all its threads must be locked in memory (there are several ways we can do this - the reference count on the pages being the most likely candidate). When a process is not in the balance set, the kernel stacks of all its threads are pageable. The locking and unlocking of these pages is performed by the balance set manager when it brings a process into or out of the balance set.

What happens if the source process gets removed from the balance set while an attach operation is in progress and causes the process' thread's kernel stacks to be made pageable?

If the source process is allowed to leave the balance set while a thread is attached to another process, then the kernel stack on which the thread is running would become pageable. This cannot be allowed to happen since it would cause the system to crash if a page fault occurred on the kernel stack itself. In order to prevent this situation from happening, the **Pcb.MutexCount** in the source **PCB** is incremented by one on attach to ensure that the process is not allowed to leave the balance set. When the corresponding detach is executed the count if decremented by one.

Even though the process is not allowed to leave the balance set any threads that do not own mutexes are prevented from further execution if the process is excluded from the balance set. Threads that do own mutexes are allowed to continue execution until they release all the mutexes they own. Therefore **Tcb.MutexCount** in the **TCB** is incremented by one on attach to ensure that the thread continues to execute. When the corresponding detach is executed the count is decremented by one.

What happens if the target process is not in the balance set?

If the target process is not in the balance set, then the subject **TCB** is inserted in the target **PCB**'s ready queue. When the corresponding process is brought into the balance set, the thread's **TCB** will be inserted in the appropriate dispatcher ready queue. We must ensure that once the target

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

process is brought into the balance set, it is not allowed to leave the balance set until the detach operation is performed. This is required since we have incremented **Tcb.MutexCount** which allows the thread to continue running in the target process' address space even though the process might be removed from the balance set. Therefore **Pcb.MutexCount** is also incremented in the target process' **PCB** during the attach operation. When the detach operation occurs all the mutex counts will be corrected to enable the respective processes to leave the balance set.

What happens if the source or target processes are terminated?

What happens if the source thread is terminated?

The kernel does not allocate or deallocate any data structures that control the execution of threads within the system. It depends on the executive to keep appropriate reference counts, and only when the reference count is zero, can the executive delete data structures. Therefore the executive must ensure that the reference count of the source process, the target process, and the subject thread are such that they cannot be deleted during the execution of a attach/detach sequence.

What happens if a thread tries to do a second attach after having attached to a target process' address space?

The **TCB** of a thread contains the storage necessary to save information for a single execution of an attach/detach sequence. Therefore the rule is that only one level of attach is allowed. If an attempt is made to attach to another address space while an address space is already attached, then a bug check will occur.

What object table is visible when a thread is attached to the address space of another process?

The object table of the attached process is visible to a thread when it is attached to another process' address space. It is doubtful that it will ever be necessary to create an object in another process' object table, but this operation can be performed if necessary.

What working set is manipulated while a thread is attached to the address space of another process?

While a thread is attached to another process' address space it takes page faults and manipulates the working set of that process as if it were really a thread in that process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

What process gets charged for the time that is consumed while the thread is attached to another process' address space?

While a thread is attached to a target process' address space, the target process is charged for the execution time accumulated by the thread. When the detach operation occurs, execution time is again charged to the source process.

How is mutex ownership handled between the source and target processes?

There is simple rule for mutex ownership. When a thread does and attach or detach process it cannot own any mutexes. If an attempt is made to attach/detach while a thread owns a mutex, then a bug check will occur.

What happens if user and/or kernel mode are alerted while a process is attached?

There is no interaction between alert and attach process. Kernel alert applies to whatever context the thread is currently in. The thread can either respond or ignor kernel alert as appropriate. User alert only applies to the source context since user mode cannot be entered when a process is attached.

A user mode alert cannot occur while a thread has a process attached since the thread will never do a wait alertable for user mode. An alert **ACB** may have been queued just prior to attaching the process in which case it will occur when the thread detachs and returns to user mode.

What happens to APC's that are queued to the thread after it has entered the target process' address space?

Can the attached thread receive APC's?

An **ACB** is initialized and directed to a thread running in a specific address space. Therefore **APC**'s directed to a source process context cannot be allowed to occur while the subject thread is attached to the address space of another process. This means that there must be a way to direct an **APC** to the right context and make sure it does not occur at the wrong time.

To accomplish this, each **TCB** will contain an **APC** state index (**Tcb.ApcStateIndex**) which can have a value of zero or one (only one level of attach is allowed). When an **ACB** is initialized the address of the associated **TCB** must be specified. This allows **Tcb.ApcStateIndex** and **Tcb.CurrentApcState.Pcb** to be captured and stored in the **ACB** in addition to the address of the **TCB** itself.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Two sets of **APC** context are stored in the **TCB**; the current **APC** context (**Tcb.CurrentApcState**) and the saved **APC** context (**Tcb.SavedApcState**). Each set of context contains the **APC** state information described for the kernel **TCB** data structure.

An array of pointers is used to address the two sets of **APC** context. When an **ACB** is queued, the appropriate set of **APC** context is selected by using **Acb.ApcStateIndex** to obtain the appropriate array member which contains the address of the corresponding set of **APC** context. A comparison is then made between the **PCB** address stored in the **ACB** and the **PCB** address stored in the selected **APC** context. If a mismatch occurs, then a bug check is executed (i.e. an attach was performed, an **ACB** was initialized (e.g. associated with a timer), a detach was performed, and then the **ACB** was queued). Otherwise the **ACB** is inserted in the selected **APC** queue and appropriate **APC** state bits are updated. If **Tcb.ApcStateIndex** is equal to **Acb.ApcStateIndex**, then the **APC** effects the current context of the subject thread and checks are made to determine if an **APC** should be delivered immediately.

When **Tcb.ApcStateIndex** is zero, the first pointer of the array points to **Tcb.CurrentApcState** and the second pointer points to **Tcb.SavedApcState**. To ensure a **PCB** address mismatch occurs if an attempt is made to queue an **ACB** with an **Acb.ApcStateIndex** value of one, a value of **NIL** is stored in **Tcb.SavedApcState.Pcb**.

When **Tcb.ApcStateIndex** is one, the first pointer of the array points to **Tcb.SavedApcState** and the second pointer of the array points to **Tcb.CurrentApcState**. Both sets of context have a valid **PCB** pointer.

When an attach process is executed, **Tcb.ApcStateIndex** is examined. If the value is one, then a bug check occurs (i.e. an attempt is being made to attach another process while one is already attached). Otherwise **Tcb.ApcStateIndex** is incremented and the current **APC** context is copied to the saved **APC** context. The two pointers in the array that address the **APC** context blocks are switched and the current **APC** state is initialized.

While a thread is executing in another process' address space, the thread can initialize and receive **APC**'s targeted to that address space.

When a detach process is executed, **Tcb.ApcStateIndex** is examined. If the value is zero, then a bug check occurs (i.e. an attempt is being made to detach an address space when one is not attached). The current **APC** context is also examined to

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

determine if the thread has a "clean" **APC** context. If a kernel **APC** is in progress, the kernel **APC** queue contains an entry, or the user **APC** queue contains as entry, then a bug check occurs. Otherwise **Tcb.ApcStateIndex** is decremented, the saved **APC** context is moved to the current **APC** context, the saved **APC** context **PCB** address is set to **NIL**, and the two entries in the pointer array are switched.

What happens if a suspend or resume is performed on the specified thread?

A thread is suspended by queuing the thread's builtin suspend **ACB**. This **ACB** is initialized such that it's target is the source process' address space and causes a normal kernel **APC**. In an attempt is made to suspend a thread while it attached to another process, then the suspend **ACB** will get queued to the source context and the suspend count will get adjusted. Suspension of the thread will not actually occur until the thread does a detach and reenters the source context. The thread may be suspended and resumed several times while it is attached to another process. This works in the same way as the case where the suspend **APC** cannot be delivered because the thread is either currently in a kernel **APC** or has kernel **APC**'s blocked (**IRQL** raised).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following pseudo code describes the operation of attach to address space:

```
PROCEDURE KeAttachProcess (
     IN Pcb : POINTER KtPcb;
     );

BEGIN

     Acquire dispatcher database lock;
     Get current TCB address;
     IF Tcb.ApcStateIndex == 1 OR Tcb.MutexCount <> 0 THEN
          Call bugcheck with fatal error;
     ELSE
          Tcb.ApcStateIndex += 1;
          Tcb.SavedApcState = Tcb.CurrentApcState;
          Tcb.CurrentApcState.Pcb = Pcb;
          Tcb.CurrentApcState.KernelApcInProgress = FALSE;
          Tcb.CurrentApcState.KernelApcPending = FALSE;
          Tcb.CurrentApcState.UserApcPending = FALSE;
          Initialize APC queue headers for current state;
               Swap APC context pointers in APC pointer array;
               Tcb.MutexCount += 1;
               Pcb.MutexCount += 1;
               Tcb.SavedApcState.Pcb->Pcb.MutexCount += 1;
               IF Pcb.Active OR Pcb.MutexCount > 1 THEN
                    Flush data cache;
                    Set new page directory pointer;
                    Release dispatcher database lock;
               ELSE
                    Tcb.PcbReadyQueue = TRUE;
                    Insert TCB in PCB's ready queue;
                    Select new thread to run;
                    Call context switch routine;
               END IF;
          END IF;
          RETURN;
     END KeAttachProcess;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following pseudo code describes the operation of detach from address space:

```
PROCEDURE KeDetachProcess (
     );

BEGIN

     Acquire dispatcher database lock;
     Get current TCB address;
     IF Tcb.ApcStateIndex == 0 OR Tcb.MutexCount <> 1 OR
        Tcb.CurrentApcState.KernelApcInProgress OR
        Current kernel APC queue not empty OR
        Current user APC queue not empty THEN
          Call bugcheck with fatal error;
     ELSE
          Tcb.ApcStateIndex -= 1;
               Tcb.CurrentApcState.Pcb->Pcb.MutexCount -= 1;
               IF Tcb.CurrentApcState.Pcb->Pcb.MutexCount == 0
                  AND NOT Tcb.CurrentApcState.Pcb->Pcb.Active
                    THEN
                    Set Tcb.CurrentApcState.Pcb->Pcb.Event;
               END IF;
          Tcb.CurrentApcState = Tcb.SavedApcState;
          Tcb.SavedApcState.Pcb = NIL;
               Swap APC context pointers in APC pointer array;
               Tcb.MutexCount -= 1;
               IF Kernel APC queue not empty THEN
                    Tcb.CurrentApcState.KernelApcPending = TRUE;
                    Set software interrupt at IRQL 1;
               END IF;
               Tcb.CurrentApcState.Pcb->Pcb.MutexCount -= 1;
               IF Tcb.CurrentApcState.Pcb->Pcb.MutexCount == 0
                  AND NOT Tcb.CurrentApcState.Pcb->Pcb.Active
                    THEN
                    Set Tcb.CurrentApcState.Pcb->Pcb.Event;
                    Tcb.PcbReadyQueue = TRUE;
                    Insert TCB in PCB's ready queue;
                    Select new thread to run;
                    Call context switch routine;
               ELSE
                    Flush data cache;
                    Set new page directory pointer;
                    Release dispatcher database lock;
               END IF;
          END IF;
          RETURN;
     END KeDetachProcess;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, February 8, 1989

Revision 1.1, February 17, 1989

1. Add text to explain what interactions exist between attach/detach process and suspend/resume, **APC**'s, alerts, and mutexes.

2. Allow APC's to be queued and processed in either the source or target address on attach/detach operations.

Revision 1.2, March 30, 1989

1. Minor edits ot conform to standard format.

[end of attproc]

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS Base Product Contents

Author: *Lou Perazzoli*

Original Draft 0.0, September 19, 1990
Revision 0.1, September 25, 1990
Revision 0.2, October 2, 1990
Revision 0.3, October 15, 1990
Revision 0.4, October 18, 1990
Revision 0.5, October 30, 1990
Revision 0.6, November 27, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction 1

2. Internal development workstation 1

3. Beta testing SDK kit (includes DDK) 2
   3.1 API Sets 2
   3.2 Subsystems 2
   3.3 File Systems 3
   3.4 Device Drivers 3
      3.4.1 MIPS R4000 PC drivers: 4
      3.4.2 Intel 486/MP and uni-processor drivers: 4
   3.5 Fault tolerance 5
   3.6 Language support 5
   3.7 MIPS support 5
   3.8 Intel 486 support 5
   3.9 Hardware booting support 6
   3.10 Installation / Setup 6
   3.11 Performance utilities 6
   3.12 Development utilities 7
   3.13 Internal Development Utilities (not shipped with SDK) 7

4. Retail Product for RISC/PC (includes an SDK) 8
   4.1 API Sets 8
   4.2 Subsystems 8
   4.3 Device Drivers 8
   4.4 File Systems 8
   4.5 Fault tolerance 8
   4.6 Language support 8
   4.7 Hardware booting support 8
   4.8 Installation / Setup 8
   4.9 Security 9
   4.10 Performance utilities 11
   4.11 Development utilities 12

5. Retail Product for Servers (RISC, 486 and 486MP) 13
   5.1 API Sets 13
   5.2 Subsystems 13
   5.3 File Systems 13
   5.4 Device Drivers 13
   5.5 Fault tolerance 13
   5.6 Language support 13
   5.7 Hardware booting support 13
   5.8 Installation / Setup 14

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.9 Security 14
5.10 Performance utilities 14
5.11 Development utilities.......... 14

6. Retail Product for 486 workstations 15
6.1 API Sets 15
6.2 Subsystems 15
6.3 File Systems.......... 15
6.4 Device Drivers.......... 15
6.5 Fault tolerance.......... 15
6.6 Language support 15
6.7 Intel 486 support 15
6.8 Installation / Setup.......... 15
6.9 Security 16
6.10 Performance utilities 16
6.11 Development utilities.......... 16

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This document describes the NT Base group deliverables for the **NT OS** for four product releases:

- o beta testing SDK kit for RISC and 486
- o retail product for MIPS and 486 workstation (includes retail SDK kit).
- o retail product for RISC, uniprocessor 486, and 486 mutliprocessor servers
- o retail product for 486 workstation which includes MVDM and Win-16 support.

  Note that 386 workstations will be supported (B6 stepping and above), but they will not have kernel support for correcting the deficiencies in i386 memory management. This deficiency manifests itself by allowing one thread to change the page protection on a page to read-only and having another thread (which is executing a kernel service) write to that page. The 486 has hardware support to honor page protections in kernel mode.

The Base group is responsible for those portions of **NT OS** which do not include networking or windowing, for example, device drivers, files systems, scheduler, loader.

2. Internal development workstation

Allows self-hosting of NT on an NT workstation. This includes CMD.EXE, compiler, assembler, linker, SLM, editor (MEP), redirector, and other tools.

As the windowing environment will still be under development, a stopgap character mode window driver will be developed which will allow the VGA on the 386/486 and frame buffer on JAZZ to appear as an ANSI terminal device. This allows character based applications to operate using the graphics device as an output device. The ANSI terminal emulation will be incorporated into the Windows environment for the SDK release. This support is described in the document titled *NT Console Interface Specification*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. Beta testing SDK kit (includes DDK)

The beta testing SDK kit contains the basic features of **NT OS** to allow ISVs and OEMs to begin developing applications and device drivers targeted specifically at Win-32 and/or NT.

3.1 API Sets

The following API sets are provided (including necessary header files for C language):

Win-32 Base API - provides the 32-bit interface for integrating with the base operating system. These APIs are described in the document titled *Win32 Base APIs* and are designed as a logical extension to the Windows 3.0 Base APIs thereby allowing a straightforward conversion of software developed for Windows 3.0. This same API set is offered on the 32-bit version of Windows.

NT Native API - this is the underlying API set for NT. It is currently undecided if this API set is formally documented, though certain features may be provided through an "NT Extension" API set. One such feature which would improve server based applications is asynchronous I/O. *Issue: if the NT API set is provided, documentation must exist.*

Device Drivers - this is the "public executive" (device helper) API set exposed by NT kernel mode components. The User Ed group is developing documentation for device driver developers. The **NT Design Workbook** specifies the device driver model and interface in documents titled *NT OS Driver Model Specification* and *NT OS I/O System Specification*.

3.2 Subsystems

The **NT OS** base provides a number of subsystems which act as servers for various applications. Subsystems operate as user mode processes but may have amplified privileges beyond the client application. This allows subsystems to manage global state, open key files, and manage critical resources on behalf of its clients.

The following subsystems exist in the **NT OS** base:

- Session Manager - provides a mechanism to start processes executing images which were developed for a different API environment then the current process. For example, a POSIX application can "exec" an image which was developed with the OS/2 API set. The session manager is describes in a document titled *NT OS Session Management and Control*.

- Security

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

    - o Local Security Authority - maintains security policy information, including list of privileged users, audit control, and security domain membership. This is described in a document titled *NT OS Local Security Specification*.

    - o Security Account Manager - maintains user and group account information as described in the document *NT OS Security Account Manager Protected Server (SAM)*.

- o Loader - provides mechanism for locating DLLs, translating symbol names to executable images, and other DLL related functions.

- o Windows Base - provide mechanism for maintaining shared state between window processes and groups. The functionality provided by this subsystem may be moved to the subsystem which provides windows graphic support.

- o Debug - provides dispatching of debug events. This subsystem is described in the document titled *NT OS Debug Architecture*.

Issue: Is DOS emulation required on the RISC/PC? How about Win-16 emulation?

3.3 File Systems

- o FAT - supports the FAT file format. This allows floppy disks to be exchanged between NT and DOS. The overall file system design is described in the document titled *NT File System Design Note*.

- o HPFS - supports the HPFS file format as defined by OS/2 v1.21.

- o NTFS - supports the NT native fully recoverable file system. This file system provides enhanced data integrity features to provide basic support for transactions. The NTFS is described in the document titled *NT Recoverable File System Specification.*

- o CD-ROM - supports the ISO CD-ROM file format.

- o NPFS - supports named pipes. The named pipe file system is described in the document titled *NT Named Pipe File Specification*.

- o BOOT - supports multiple boot partitions and allows new file formats to be bootable as described in the *NT Boot Architecture*.

3.4 Device Drivers

Device drivers provide the necessary logic to bind the I/O functions to a physical device. **NT OS** supplies the proper mechanisms to allow drivers to be loaded either at system initialization or later once the system is operational.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.4.1 MIPS R4000 PC drivers:

- floppy as described in the document *NT Floppy Driver Specification*.
- SCSI driver with support for disk, CD-rom and tape as described in *NT SCSI Design Note*.
- serial - western digital part (2 serial, 1 parallel port), supports modems, printers, basic serial devices as described in the *NT Serial Driver Specification.*
- parallel - western digital part, supports printers and basic parallel devices as described in the *NT Parallel Driver Specification*.
- video - frame buffer as described in *NT Screen Device Driver Design Note.*
- keyboard as described in *NT Keyboard Device Driver Design Note.*
- mouse - in port as described in *NT Mouse Device Driver Design Note.*
- sound
- EISA support - verification driver to show that EISA functions properly.

3.4.2 Intel 486/MP and uni-processor drivers:

- floppy as described in the document *NT Floppy Driver Specification*.
- SCSI driver with support for disk, CD-rom and tape as described in *NT SCSI Design Note*.
- disk - ST506 EDSI driver as described in the *NT EDSI Driver Specification.*
- serial - Intel 8250 part supports modems, printers, basic serial devices as described in the *NT Serial Driver Specification.*
- parallel supports printers and basic parallel devices as described in the *NT Parallel Driver Specification.*.
- video - frame buffer as described in *NT Screen Device Driver Design Note.*
- keyboard as described in *NT Keyboard Device Driver Design Note.*
- mouse - in port and serial variants as described in *NT Mouse Device Driver Design Note.*
- EISA support - verification driver to show that EISA functions properly

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o MCA support - verification driver to show that MCA functions properly

3.5 Fault tolerance

For systems with battery backed up memory, power fail recovery is supported. This support involves saving volatile hardware registers and caches into RAM during loss of power and restoring the system state when power is regained. At restoration time, all drivers requesting powerfail notification are notified and any I/O operations in progress are restarted by the drivers.

3.6 Language support

3.7 MIPS support

- o C compiler for MIPS (from either MS or MIPS)
- o MIPS assembler for R4000 (only runs on RISC/PC)
- o Linker for R4000 (provided by NT/Base group)
- o Debugger similar to symdeb
- o Kernel debugger for device driver ISV's (requires separate host machine, currently running OS/2)
- o C Run time libraries for Win-32 applications
- o Cross development tools for 486 development:
    - o C6.0 compiler
    - o MASM Assembler
    - o Linker for 486 modules. Current plan is for the NT native linker to support both MIPS and 486 OMFs (Object Module Formats).

3.8 Intel 486 support

- o C6.0 compiler
- o MASM Assembler
- o Linker for 486 modules.
- o Debugger similar to symdeb

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Kernel debugger for device driver ISV's (requires host machine, currently OS/2).

  Issue: the kernel debugger should be ported to the Win-32 environment at a minimum and possibly to the Win-16 environment. Porting to the Win-16 environment provides the least disruption to the target audience.

- o C Run time libraries for Win-32 applications

3.9 Hardware booting support

The following platforms are being utilized for development and/or testing and as such hardware booting support and configuration will be provided.

- o Power PC/RISC (Jazz)
- o Compaq Deskpro-486 (EISA)

For 386 environments, Intel 387 floating point emulation is provided for system without 387 coprocessors.

3.10 Installation / Setup

The beta SDK release will have minimal installation / setup support. This includes support for building a bootable system from a floppy disk kit and copying the appropriate SDK header files and utilities to the hard disk.

3.11 Performance utilities

The beta SDK will have basic performance utilities.

- o profiler - provides mechanism to obtain a time sampled PC histogram. The profiler is implemented like a debugger; no changes are required to the application to enable profiling. The profiler operates in its own address space and creates profiling objects on behalf of the process being profiled. When the process completes, the profiler closes the profile objects and analyzes the collected data. The beta SDK version of the profiler will not be GUI based. The profiler fucntionality is not currently documented.

- o show system information - shows the current resource usage, active processes, active threads, etc. within the system. The show system functionality is not currently documented.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.12 Development utilities

CMD.EXE - command interpreter (ported from OS/2) provides basic commands (dir, ren, del, etc) and batch script capability.
format - format disks, supports FAT format for floppy, HPFS, FAT, and NTFS for hard disks.
chkdsk - check disk, checks disk for consistent file structure and bad blocks
chmode - allows protection on file to be changed
diskcopy - sector based floppy disk copy
diskcomp - sector based disk comparison
du - disk usage by directory
ech - echo string
fcom - compare files (both text and binary)
fcopy - general purpose file/directory copy
fdel - general purpose file/directory deletion
fview - extensible file viewer, views text files, objects, images, etc.
ls - list directory contents
nmake - program maintenance utility
ppr - remote print
qgrep - search for strings in files
sort - sort file contents base on keys
timer - simple execution timer
touch - change file time stamps
walk - walk a directory tree applying command to files and directories
where - locate files in a directory tree
ync - single character batch file prompts (yes, no, continue)
editor (MEP) which utilizes WinHelp

3.13 Internal Development Utilities (not shipped with SDK)

cp - copy file to file or files to directory
delnode - delete directory tree
exp - remove deleted files
mv - rename files and directories
rm - make files deleted
slm - source control maintenance facility
t - terminal emulator
undel - undelete deleted files
upd - timestamp based file copy
updrn - timestamp base file copy for directories
xcopy - copy file and directory tree

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Retail Product for RISC/PC (includes an SDK)

The retail product for RISC/PC includes the final version of the components provided in the beta SDK release plus installation/setup features, POSIX compliance and security at the C2 level.

4.1 API Sets

Same as beta SDK with addition of POSIX support.

POSIX 1003.1 API - provides the POSIX compliant APIs. These APIs are defined by the *IEEE 1003.1 POSIX specification.* The APIs supported are the minimum set required for to obtain POSIX certification, i.e., none of the optional APIs will be supported.

4.2 Subsystems

Same as Beta SDK with the addition of POSIX.

- o POSIX - provides support for all processes executing the POSIX API set.

4.3 Device Drivers

Same as beta SDK.

4.4 File Systems

Same as beta SDK.

4.5 Fault tolerance

Same as beta SDK.

4.6 Language support

Same as beta SDK plus the addition of C run time libraries for POSIX applications.

4.7 Hardware booting support

Same as beta SDK.

4.8 Installation / Setup

Complete installation / setup support including configuration management. The documentation for installation and system management is currently under development.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Architecture dependent kernel routines
- o System configuration / configuration management
- o System management
  - o Error log reporting mechanism. This is a character mode application that allows error log reports to be generated based on error type, time, and device type. For example, list all Fatal errors on device Harddisk0 between Jan 1 1990 12:00 and Jan 1 1990 18:00.
  - o System crash dump and analysis utility. This provides a mechanism to dump the contents of physical memory to a file on the disk in the case of a system crash. When the system is rebooted, the analysis utility allows the cause of the crash to be analyzed. In severe cases, crash dump contents may be copied to floppy or tape and sent to product support specialists for analysis.
  - o File backup on SCSI tape. This utility provides the ability to backup and restore complete volumes or selected files onto tape.

  Issue: *does this need to be SYTRON compatible to provide the ability to read files written on an OS/2 system? How about just supporting TAR format??*

- o Application installation - provides a mechanism to install application software on an NT system.
- o National Language Support (NLS) - provides a mechanism for tailoring an NT system to a specific language environment.
- o Shutdown - allow orderly shutdown of the system as a reasonable alternative to Ctrl-Alt-Del. The shutdown mechanism flushes file caches, terminates network connects, and does an orderly shutdown of the system.

4.9 Security

NT OS provides security features to allow the base operating system to be certified at the C2 level (discretionary access control) for the first product release, and eventually at the B1 level. In order to gain certifications certain features and utilities must be present in the system to allow the detection and analysis of break-in attempts and suspected attempts. In addition, a mechanism must be provided to allow users to display and manipulate security information on objects, most notably files.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following components are provided to support security:

User Interface:

User Account Manager - This utility is based upon the LAN Manager 3.0 User Account Manager utility. It includes minor extensions to support administration of Security Account Manager concepts that don't exist in LAN Manager.

Local Security Manager - This utility allows the security parameters of each NT system to be administered. This is a new utility with no corresponding LAN Manager functionality. This utility will utilize the Local Security Manager DLL described below.

Win32 Logon User Interface - This is the user interface presented at logon time. It collects the user name and password and prevents password stealing by unauthorized processes. This UI is projected by the Win32 logon process described below.

Win32 File Browser extensions - The Win32 File Browser will be extended to support security by:

- Displaying security of files and directories upon request.
- Allow modification of file and directory protection and auditing requirements (using the Object Security editor DLL described below).
- Allow modification of file and directory owner values.

The Win32 Shell will allow a user to establish security *personas* and to modify the user's active security persona. This will allow the user to perform actions such as changing default protection or enabling and disabling privileges.

Some aspects of installation will deal with establishing the customer's mode of operation (secure or non-secure) and collecting security parameters, if running securely. A secure system may also have to convert a LAN Manager UAS database to a Security Account Manager database.

Some aspects of configuration control will deal with the security attributes associated with components of the configured system. For example, protected subsystems, such as the NT Session Manager, may be assigned privileges to be run with.

Runtime Library & Client Stubs:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Runtime Library routines will be included for the manipulation of security data structures, such as access control lists.

Client RPC stubs will be included for Security Account Manager services, making the security Account Manager a network-wide service. This allows administration of security accounts from remote nodes.

Client RPC stubs will be included for Local Security Authority services, making the Local Security Authority a network-wide service. This allows administration of individual system security from remote nodes.

Executable Images And DLLs:

Security Account Manager protected subsystem image (sam.exe). This image is run as a native NT protected subsystem. It services user/group account administration requests, as well as user authentication requests. This image will only be run on Domain Controller nodes.

Local Security Authority protected subsystem (lsa.exe). This image is run as a native NT protected subsystem. This image is responsible or maintaining and enforcing all security policy for an individual system, such as what audit messages to generate. This protected subsystem will be active on each NT system.

Win32 Logon Process (w32logon.exe). This image is responsible for monitoring Win32 for logon requests, and processing them when received. It prevents Trojan programs from stealing user passwords. This is a customer modifiable or replaceable module and we will ship the source code for this module. This image will be active on each NT system.

Local Security Manager DLL (lsm.dll). This DLL provides Win32 user Interface screens for administering the local system security. This is implemented as a DLL to allow this functionality to be activated from a number of related UI utilities (such as the security account administrator).

Object Security Editor DLL (objsec.dll). This DLL provides object protection viewing and modification capabilities. It is implemented as a DLL to allow a standard view of object security to be used anyplace it is needed. For example, the file browser will use this DLL for file and directory protection modification and the Security Account Manager will use this DLL for user and group account protection modification.

4.10 Performance utilities

Same as beta SDK with a GUI interface to show system information utility.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.11 Development utilities

Same as beta SDK with the addition of UI enhancements to some utilities and the user debugger.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Retail Product for Servers (RISC, 486 and 486MP)

The retail product for servers includes the retail product components provided in the above product in addition to a more robust networking environment.

5.1 API Sets

Same as RISC workstation product.

5.2 Subsystems

Same as RISC workstation product.

5.3 File Systems

Same as RISC workstation product.

5.4 Device Drivers

Same as retail product (both MIPS and 486).

5.5 Fault tolerance

- Disk Mirror - allows files mirroring of disk image on another disk(s) block for block. While this is implemented as a layered driver, it is listed under file systems.
- UPS - uninterruptable power systems support
- Dual controller support ??

5.6 Language support

Same as RISC workstation product with the addition of C++ support.

5.7 Hardware booting support

The following platforms are being utilized for development and/or testing and as such hardware booting support and configuration will be provided.

- Power PC/RISC (Jazz)
- Power PC/486 with EISA bus
- Power PC/486 with MCA bus
- Compaq Deskpro 486

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o IBM PS/2 Model 90
- o Power MP/486 - to be determined.

5.8 Installation / Setup

Same installation / setup features provided in the RISC workstation product plus the addition of:

- o Disk mirroring management
- o Logical volume management - allows multiple disks to be configured such that they appear as a single drive.

5.9 Security

More network based security? remote admin?

5.10 Performance utilities

network performance things?

5.11 Development utilities

Same as RISC workstation product.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6. Retail Product for 486 workstations

The retail product for 486 workstations provides the support for running Windows 16-bit applications and DOS applications as well as support for 32-bit OS/2 non PM base (i.e., server) applications.

6.1 API Sets

Same as server product.

6.2 Subsystems

Same as server product plus the addition of:

- MVDM subsystem
- Windows 16-bit subsystem
- OS/2 subsystem

6.3 File Systems

Same as server product.

6.4 Device Drivers

Same as server product.

6.5 Fault tolerance

Same as server product.

6.6 Language support

Same as server product.

6.7 Intel 486 support

Same as server product.

5.8 Hardware booting support

Same as server product.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.8 Installation / Setup

Same installation / setup features provided in the server product plus the addition of:

- MVDM installation
- Windows 16-bit installation
- OS/2 Subsystem installation

6.9 Security

Same as server product.

6.10 Performance utilities

Same as server product.

6.11 Development utilities

Same as server product.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Caching Design Note

Author: *Tom Miller*

Revision 1.3, October 31, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview 1
  1.1 File Streams and Cache Maps 1
  1.2 Target Clients of the Cache Manager 2
  1.3 Cache Manager Interfaces 3

2. WalkThrough of Cache Manager Interaction 5
  2.1 Setting up the File Object on Create 5
    2.1.1 FsContext 5
    2.1.2 SectionObjectPointer 7
    2.1.3 PrivateCacheMap field 7
  2.2 Initializing Cache Maps for a File Stream 7
  2.3 Accessing Data in the Cache 8
    2.3.1 Copying Data To and From the Cache 8
    2.3.2 DMA Transfer of Data To and From the Cache 9
    2.3.3 Accessing Data Directly in the Cache 9
  2.4 Uninitializing Cache Maps for a File Stream 10
  2.5 Fast I/O Optimization 10
  2.6 Use of the Wait Input Parameter 11
  2.7 Use of Stream Files 11
  2.8 File System Cleanup and Close Routines 12
  2.9 Using Write Through and Cache Flushing 13
  2.10 Valid Data Length and File Size Considerations 14
  2.11 Resource Locking Rules 15
  2.12 Network File Server Interfaces 17

3. File System Maintenance Functions (FSSUP) 19
  3.1 CcInitializeCacheMap 19
    3.1.1 Cache Manager Callbacks 20
  3.2 CcUninitializeCacheMap 21
  3.3 CcExtendCachedFileSize 23
  3.4 CcExtendCacheSection 23
  3.5 CcFlushCache 24
  3.6 CcPurgeFromWorkingSet 24
  3.7 CcPurgeCacheSection 25
  3.8 CcTruncateCachedFileSize 25
  3.9 CcZeroData 26
  3.10 CcRepinBcb 27
  3.11 CcUnpinRepinnedBcb 27
  3.12 CcIsFileCached 27
  3.13 CcReadAhead 28
  3.14 CcSetAdditionalCacheAttributes 29

4. Copy Interface (COPYSUP) 30
  4.1 CcCopyRead 30
  4.2 CcCopyWrite 30

5. Mdl Interface (MDLSUP) 32
  5.1 CcMdlRead 32
  5.2 CcMdlReadComplete 33
  5.3 CcPrepareMdlWrite 34

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.4 CcMdlWriteComplete 35

6. Pin Interface (PINSUP) 36
6.1 CcPinRead 36
6.2 CcMapData 37
6.3 CcPinMappedData 38
6.4 CcPreparePinWrite 40
6.5 CcSetDirtyPinnedData 41
6.6 CcUnpinData 41

7. Revision History 42

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This design note describes the Cache Manager for Windows NT. The Cache Manager uses a file mapping model, which is closely integrated with memory management.

The file mapping model or virtual block cache, has been chosen over a logical block cache for the following reasons:

- o Virtual block caching is more compatible with the ability of user programs to map files. It is possible for some programs to do NtReadFile and NtWriteFile at the same time that other programs have the file mapped with read-only or read/write access. With proper synchronization, both types of programs are able to see the most current data.

- o By using a file mapping model, all of physical memory becomes available for data caching, with the allocation of pages reacting dynamically to the changing needs for image file pages versus data file pages.

- o Cache hits are processed more efficiently by handling virtual block hits directly in a mapped file. In most cases an I/O request is able to access the data directly in the cache, without calling the file system at all (see Section 0). The I/O system makes a subroutine call to access the cache, and the Cache Manager resolves the access via a single hardware virtual address lookup.

- o For the a recoverable file system such as NTFS, it is necessary to have caching closely synchronized with logging. This requires that all cache entries be directly identifiable by the recoverable file to which they belong.

The Cache Manager also provides a simple mechanism for dealing with unaligned buffers. If a file has been opened with caching disabled (FILE_NO_INTERMEDIATE_BUFFERING specified in the Create/Open options), then an NtReadFile or NtWriteFile *will fail* if the alignment and size of the specified transfer is less than that required by the target disk. The assumption is, that if a program specified a request with caching disabled, then it really does not want to pay the cost of having the transfer go to an intermediate buffer and be copied.

1.1 File Streams and Cache Maps

The Cache Manager is a central system component which may be thought of as being layered closely on top of the Memory Management support. Key to understanding the Cache Manager is the concept of *File Streams*.

A File Stream is a linear stream of bytes associated with a File Object. Each File System creates, deletes and manipulates File Streams both for external use via NT File System APIs, as well as for internal use by the File System itself. Examples of File Streams maintained by File Systems are the data of a given file, the EAs of a file, the Acl of a file, a directory, or any other file system metadata. How virtual byte offsets within the File Stream are mapped to physical locations in nonvolatile store is strictly an opaque operation determined by the File System, and may vary for different types of file streams.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Once a file system has identified which streams it wishes to support, it needs to decide which of these streams it wishes to cache. For all streams which are to be cached, the file system must actually support both cached and noncached access. Noncached access is always issued via a read or write I/O Request Packet (IRP), in which the **IRP_NOCACHE** flag is set in the Irp flags. (See the *NT I/O System Specification.*) For streams which may be accessed by normal user programs, such as the data of a file, the file system will also receive cached I/O requests via read or write IRPs with the **IRP_NOCACHE** flag not set. Also for internal use a file system may perform cached access to any of the streams it defines via direct calls to the Cache Manager.

As mentioned earlier, the Cache Manager uses mapping to implement the caching of streams, and to integrate caching with Memory Management's policy with other uses of pageable memory. Thus when a file system calls the Cache Manager to intitiate caching of a stream, the Cache Manager immediately maps all or a portion of the stream via a call to memory management. For larger streams, the Cache Manager may subsequently find it necessary to map additional portions of the stream on an as-needed basis. To keep track of which portions of a file stream the Cache Manager currently has mapped, it uses private data structures which it refers to as *Cache Maps*. For each stream being cached, the Cache Manager maintains a single *Shared Cache Map*. For each File Object through which the cached stream is being accessed, the Cache Manager also maintains a *Private Cache Map*. The Shared Cache Map describes an initial portion of the file stream which is mapped for common access via all File Objects for this stream. Each Private Cache Map optionally describes an additional nonoverlapping portion of the stream mapped on an as-needed basis to access bytes in the stream which were not mapped by the Shared Cache Map.

Again, the Cache Maps are private structures maintained by the Cache Manager, and a further understanding of these structures is not required by a person writing a file system. However, a file system writer does have to be aware of the respective relationships between a file system, the Cache Manager, and Memory Management. For example, when an attempted cache access results in a "miss", this miss results in a page fault which is serviced by Memory Management who subsequently makes a (recursive) call back to the file system with a noncached I/O request.

1.2 Target Clients of the Cache Manager

The Cache Manager interfaces have been primarily designed to support the following clients:

- Normal file systems such as FAT, HPFS and CDFS. File systems may create and cache File Streams for normal data files, the EAs associated with a file, the volume structure of a volume, etc. Note that the Cache Manager knows nothing about different types of streams; it only knows about File Objects and different modes of access.

  For example, HPFS creates File Streams to cache normal file data, the first time the data is actually accessed. It also creates a File Stream for a "Volume File", which is a compressed mapping of the volume structure on a HPFS volume. If the EAs or ACL for a given file fit in the Fnode, then they are

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

simply cached with the Fnode in the Volume File. The other case HPFS has is that the EA or ACL is too large to fit in the Fnode, and is described by one or more runs of contiguous sectors external to the Fnode. In this case, a separate stream is created to cache the EA or ACL the first time they are accessed.

Interfaces are provided for File Systems to access data by copying, or accessing it directly in the cache.

- Network File System clients, such as the Lan Manager Redirector. For starters, a Network File System looks like any other File System, with normal data streams, and potentially other types of streams associated with files. However, a Network File System client would normally not be maintaining any "volume" structure of its own.

- Network File Servers, such as the Lan Manager Server. A file server is not expected to look like a file system at all. However, it also may be considered a "client" of the Cache Manager via the host file system(s) which it calls. Indeed, some of the file system calls which are ultimately supported by the Cache Manager (such as the Mdl interfaces defined later), were designed with Network File Servers in mind.

1.3 Cache Manager Interfaces

The Cache Manager has four sets of interfaces. One is for basic File Stream maintenance, and the other three implement different access methods for the cache. The three access methods share common support routines, but acknowledge the different ways in which the cache will be used.

Following is a brief description of the four sets of interfaces supported by the cache manager, which are described in detail in the following sections:

- File Stream maintenance functions.

  The File Stream maintenance functions are implemented in the Cache Manager module fssup.c. These routines are for initializing and uninitializing cached operation for a stream, extending and truncating cached streams and file sizes, flushing pages to disk, purging pages from the cache without flushing, zeroing file data, and so on.

- Copy Interface.

  The Copy Interface is implemented by the Cache Manager module copysup.c. The copy interface is the simplest form of cached access. It supports copying a range of bytes from a specified offset in a cached file stream to a buffer in memory, or from a buffer in memory to a specifiedd offset in a cached file stream. The copy interface also has a related call to initiate read ahead.

- Mdl Interface.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The Mdl Interface is implemented by the Cache Manager module mdlsup.c. The Mdl interface supports direct access to the cache via DMA. For example, a network file server can efficiently support large client reads via DMA of the desired bytes directly out of the cache to a network device. Similarly a network file server is able to support large client writes by DMA directly into the cache. The Mdl interface shares the same Read Ahead call as the copy interface.

- o A *Pinning* Interface.

  The Pinning Interface is implemented by the Cache Manager module pinsup.c. The pin interface may be used to lock (pin) data in the cache and access it directly via a pointer, and then unpin the data when the pointer is no longer required. Pinning is a database concept, and it is the optimal way for a File System to deal with the caching of file system metadata:

The following table summarizes which of the Cache Manager's clients are intended to use which of the four interface classes. Note that Network File Servers never call the Cache Manager directly, but rather benefit from the specified interfaces via associated calls to local file systems.

	Local File Systems	Network FS Clients	Network File Servers
FS Maint.	x	x	
Copy Int.	x	x	x
Mdl Int.		x	x
Pin Int.	x		

The next section walks through what a file system has to do to set up for and use the Cache Manager. Then, subsequent sections will document the individual routines belonging to the four classes of interfaces presented above.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2. WalkThrough of Cache Manager Interaction

This section attempts to present all of the background information which is important to understand when about to write a File System (including a Network File System client) or File Server which intends to use the Cache Manager. All of the following subsections but the last one relate only to file systems, but may provide some insight to someone writing a file server.

The final subsection describes how a file server accesses cached file streams. The final section should also be understood by anyone writing a local file system.

The following include files, present in \nt\private\inc, define the data structures and procedure calls described in this section and the rest of this document:

cache.h	Cache Manager structures and routines
fsrtl.h	File System Rtl structures and routines
io.h	I/O system structures and routines
ex.h	Executive structures and routines

2.1 Setting up the File Object on Create

When a file system is called at its Create Fsd entry point, one of the important fields in the Irp is a pointer to a File Object (see io.h) for the file being opened. There are three pointers in the File Object which must be initialized in a particular way for a file system which wishes to use the Cache Manager. These fields are FsContext, SectionObjectPointer, and PrivateCacheMap. (A fourth pointer, FsContext2, has no significance to the Cache Manager, and is usually used to point to a per file object context called the Channel Control Block or CCB.) The following subsections describe how these fields are to be initialized.

2.1.1 FsContext

The Cache Manager expects FsContext to point to a structure defined by FsRtl, called the FSRTL_COMMON_FCB_HEADER. This structure must be allocated from nonpaged pool, and must exist only once for the respective file stream, no matter how many times it is opened. So, for example, for normal files, exactly one Common Fcb Header must exist for each open file on the volume, no matter how many times the file is opened. If the same file is opened multiple times, then for each File Object which has the file open, FsContext must point to the same Common Fcb Header. The Common Fcb Header will generally be contained at the beginning of a common structure maintained by the file system for this file, typically called the File Control Block, or Fcb.

> *Note that currently the Common Fcb Header is actually only used to support Fast I/O, which means it is actually only required for file objects describing user file opens. However, it might be recommedable for a new file system to always point FsContext to this structure, even for stream files.*\\

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The Common Fcb Header is defined in fsrtl.h, and at the time of this revision has the following format.

```
typedef struct _FSRTL_COMMON_FCB_HEADER {

    CSHORT NodeTypeCode;
    CSHORT NodeByteSize;

    BOOLEAN IsFastIoPossible;

    LARGE_INTEGER AllocationSize;
    LARGE_INTEGER FileSize;
    LARGE_INTEGER ValidDataLength;

    PERESOURCE Resource;

} FSRTL_COMMON_FCB_HEADER;
typedef FSRTL_COMMON_FCB_HEADER *PFSRTL_COMMON_FCB_HEADER;
```

Here is a brief definition of these fields:

NodeTypeCode - A unique code identifying the Fcb for this particular file system. This field is unused by the Cache Manager.

NodeByteSize - The size of the entire containing Fcb in bytes. This field is also not used by the Cache Manager.

IsFastIoPossible - This boolean contain TRUE (0x01) whenever the file system believes it is acceptable for the I/O system to call the Cache Manager directly to read or write byte ranges directly in the cache, without calling the file system. It must contain FALSE (0x00) whenever it is not acceptable to access cache data directly. In this case, all cached reads and writes must be passed to the file system via Irp.

Examples of cases where this field might contain FALSE, would be if there are active FileLocks or Network Oplocks in the file, or the media is undergoing volume verification.

AllocationSize - The current allocation size of this file in bytes, typically an integral multiple of the underlying device sector size or allocation cluster size. This field must be initialized to the correct value when the Fcb is created, and thereafter the Cache Manager must be notified when it changes.

FileSize - The logical size of the file up to which the file may be read. Reads beginning before this point are truncated, and reads beyond this point return STATUS_END_OF_FILE. This field must be initialized to the correct value when the Fcb is created, and thereafter the Cache Manager must be notified when it changes.

ValidDataLength - The size of the initialized portion of the file. If ValidDataLength is less than FileSize, then reads extending beyond ValidDataLength return

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

binary 0 in the read buffer. This field must be initialized to the correct value when the Fcb is created, and thereafter the Cache Manager will inform the file system when it is safe to change this value for the file on disk (see Section 0).

Resource - Pointer to an ERESOURCE structure (defined in ex.h). The ERESOURCE structure is usually allocated elsewhere in the Fcb. The executive resource structure is a synchronization structure which supports multiple Shared accessors at once, or one Exclusive accessor via the routines ExAcquireResourceShared, ExAcquireResourceExclusive and ExReleaseResource. FsRtl and the Cache Manager require that all file system operations for this stream be synchronized by this resource. Of course, modifying operations generally require that the file system take out exclusive access, and nonmodifying access require that the file system take out shared access. The synchronization requirements of streams will be further discussed later.

2.1.2 SectionObjectPointer

Memory Management and the Cache Manager require that the SectionObjectPointer field of the file object point to a structure call SECTION_OBJECT_POINTERS. This structure must also exist only once for the file stream, and it must also be allocated in nonpaged pool. Generally this structure is also allocated somewhere in the file system's Fcb.

The Section Object Pointers structure is defined in io.h, and at the time of this revision has the following format.

```
typedef struct _SECTION_OBJECT_POINTERS {
    PVOID DataSectionObject;
    PVOID SharedCacheMap;
    PVOID ImageSectionObject;
} SECTION_OBJECT_POINTERS;
typedef SECTION_OBJECT_POINTERS *PSECTION_OBJECT_POINTERS;
```

Here is a brief description of these fields, however the file system only has to initialize this structure by clearing it:

DataSectionObject - This pointer is used by Memory Management whenever a data section has been created for this stream, including when the Cache Manager has done so.

SharedCacheMap - This pointer is used by the Cache Manager to point to its SharedCacheMap structure whenever the file is currently being cached.

ImageSectionObject - This pointer is used by Memory Management whenever an image section has been created for this stream.

2.1.3 PrivateCacheMap field

This pointer must simply be initialized with NULL (0x00000000). It is filled in by the Cache Manager if the stream is cached. The only way for the file system to

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

reliably determine if this file object is currently being cached (**CcInitializeCacheMap** has been called), is to have the Fcb Resource shared or exclusive, and test the PrivateCacheMap field for NULL. It is not valid for the file system to capture this information elsewhere, because under certain circumstances the File Object has to be forcibly uninitialized.

2.2 Initializing Cache Maps for a File Stream

The previous section described how the Cache Manager expects the File Object to be initialized on Create. The Cache Manager however does not expect Create to initiate caching, but rather that this work be deferred to the first read or write of the file. This is basically for two reasons:

First, experience has shown that it is very inefficient to immediately initiate caching on a file when it is opened, since there are quite a few apps which open a file, get or set some file information on the file or mark it for delete, and then close the file without ever accessing its data. These applications may run noticeably slower if the file system is needlessly initializing and uninitializing caching for the files.

More importantly, however, is the fact that under certain circumstances it becomes necessary to forcibly uninitialize the Cache Maps on a file object. Examples are when a file is truncated, or the file system supports removeable media and a volume fails mount verification. A network file system client may decide to call **CcUninitializeCacheMap** on all of its files in the event of an oplock break, or a virtual circuit goes down, and the data can no longer be trusted. Forced uninitialization works because we know that the next read or write, if there is one, will simply initialize the cache again.

Note that if there are multiple accesses to the same file, as represented by multiple file objects, the file system must call **CcInitializeCacheMap** for each file object that attempts to access file data. It must also eventually call **CcUninitializeCacheMap** for each of these file objects.

For further detail on initialization, see Section 0 on **CcInitializeCacheMap**.

Once, caching has been initialized on a file object, and the Fcb resource is still acquired, a file system may access the cache via one of the classes of access routines described in the next section.

2.3 Accessing Data in the Cache

Once the file object is set up and **CcInitializeCacheMap** has been called, a file system may now access data in the cache. There are three methods for accessing the cache, and these methods are described in the following subsections.

2.3.1 Copying Data To and From the Cache

The simplest way of accessing a stream is to copy data into and out of it. The routines **CcCopyRead** and **CcCopyWrite** are provided for this purpose. They both take a previously initialized file object along with a description of the desired byte range in the file and an input or output buffer in memory. It is essential that the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Fcb resource remain acquired from some time before the point where the file object was initialized (if it was not already), until after the copy operation is complete.

In the case of a call to **CcCopyRead**, the caller may also wish to call **CcReadAhead** to see if any Read Ahead is desired. The call to **CcReadAhead** takes some of the same parameters as the call to **CcCopyRead**. **CcReadAhead** automatically determines whether or not read ahead is appropriate, based on the recent history of calls to **CcReadAhead**, and whether or not the data has perhaps already been read ahead. If read ahead is required, it is scheduled to be performed by one of the Cache Manager's worker threads, so as not to hold up the current thread.

For the case of **CcCopyWrite**, all modified pages are lazy written by default. For most cases the file system simply does not have to worry about it, and the data will typically get to disk within about five seconds of when it was modified. For cases where Lazy Writing is not appropriate because the data has to be written through, see Section 0.

2.3.2 DMA Transfer of Data To and From the Cache

Network file servers and network file system clients sometimes have to transfer large amounts of data into or out of the cache from a network device. For such large transfers, it is inefficient to allocate a temporary buffer, call the copy interfaces above, transfer the data on the network device, and then free the temporary buffer. In order to eliminate the large copy and temporary buffer in the above scenario, the Cache Manager provides a second class of interface to the cache, called the Mdl interface.

The Mdl (Memory Descriptor List) contains a physical description of a buffer in memory, according to the physical pages it occupies. This structure should already be familiar to anyone dealing with the network (see the *Windows NT I/O System Specification*).

> *Please note in the following discussion that the network software does not actually call the CcMdl routines directly, however we describe it that way here for simplicity. See Section 0.*\\

To read from the cache and write to the network, network software may first call **CcMdlRead**, specifying the initialized file object and range of bytes required. **CcMdlRead** returns an Mdl (actually a linked list of Mdls called an Mdl Chain) describing the desired byte range directly in the cache. Note that the reader does not have to specify a transfer that starts on a page or sector boundary, he only needs to make sure he is specifying a file offset with sufficient alignment to satisfy his network device. Once **CcMdlRead** has returned, the pages containing the desired data are locked in memory, and the reader may use the Mdl chain to effect the transfer on the network. Prior to that the network software may wish to prepend an Mdl to the Mdl chain returned by the Cache Manager, in order to describe header information. When the network transfer is complete, **CcMdlReadComplete** must be called to unlock the cache buffers and delete the Mdl chain. Just as described with the Copy interfaces, **CcReadAhead** may be called to have the Cache Manager decide whether he should schedule some data to be read ahead after a **CcMdlRead**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Similarly for writing to the cache, network software may call **CcPrepareMdlWrite** to prepare a space in the cache to receive the desired data for a specified byte range in the file. The Mdl returned may then be used to specify a direct DMA transfer of the data into the cache off of the network. When the DMA is complete, **CcMdlWriteComplete** must be called to unlock the buffers and free the Mdl chain. Also as in the Copy case, after receiving the **CcMdlWriteComplete** call, the Cache Manager automatically guarantees that the new data is eventually written to disk.

It is acceptable to mix Copy access and Mdl access to the same file.

2.3.3 Accessing Data Directly in the Cache

Local file systems sometimes wish to access data directly in the Cache, possibly modifying it in place. This is particularly interesting for file streams which have been defined to describe file system metadata, such as directories. For this purpose the Cache Manager provides a third interface class referred to as the Pin interface.

If a file system wants to access a structure directly in a stream, possibly modify it, and then release it, it may start by calling **CcPinRead**. **CcPinRead** takes an initialized file object, and the offset and length of the desired byte range. It returns a virtual address at which the desired file data may be accessed, and an opaque Buffer Control Block address (Bcb) which will be used to free the buffer later. If the file system subsequently modifies the pinned data, then it must call **CcSetDirtyPinnedData** before unpinning it. If the file system knows in advance that it will be modifying an entire range of bytes, then it may call **CcPreparePinWrite** instead of **CcPinRead**, and the data will automatically be set dirty (and optionally zeroed in advance). In any case, when the file system is done with the pinned data, it must call **CcUnpinData**, to release the buffer, and allow it to be written if it is dirty.

If the file system knows in advance that it does not need to modify the desired data, or knows in advance that it may not need to modify the data, then instead of calling **CcPinRead** it can use a faster call which is **CcMapData**. **CcMapData** has the same interface as **CcPinRead**, but it is much cheaper since it does not lock the data in memory. If the caller later decides that he does need to modify the data, then he may call **CcPinMappedData** to lock it in memory (and then call **CcSetDirtyPinnedData**). In any case, when done with the mapped and optionally pinned data, the caller must call **CcUnpinData** when done.

Since pinning is generally used for random access to file system metadata, read ahead is usually not performed. As to modified data, the Cache Manager guarantees that any data that was set dirty will eventually be written to disk, typically within about five seconds.

For reasons relating to Cache Manager implementation details, it is not acceptable to mix Pin access to a file with Copy or Mdl access.

2.4 Uninitializing Cache Maps for a File Stream

When a file system is done accessing a given file on a given file object, it must call **CcUninitializeCacheMap**. This routine should generally be called in the file

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

system's cleanup processing. **CcUninitializeCacheMap** must be called for each file object on which **CcInitializeCacheMap** was called.

By default, **CcUnintializeCacheMap** does not remove the file from the cache, it simply tells the Cache Manager that the file system is no longer accessing that file from the specified file object. The file may still remain in the cache for some time until its pages get reclaimed for caching another file (or program image, etc.).

If for any reason a file system does wish to have all or part of a file removed from the cache, **CcUnitializeCacheMap** provides this capability as well (see Section 0).

2.5 Fast I/O Optimization

There is a module in FsRtl which provides fast access to cached data without calling the file system. The routines in this module may be called by the I/O system when caching has already been initialized on a file object. They may also be called by file servers.

Since the file system is never called on the Fast I/O path, it is important that it have the ability to enable or disable these calls. Fast I/O should generally be left enabled unless some condition exists in a file for which correct handling can only be guaranteed by executing the normal file system read and write paths. For example, if any file locks exist in the file, or network oplocks, then execution of a fast I/O path may not work correctly.

If the file system detects a case which makes Fast I/O unsafe, then it must simply clear the IsFastIoPossible boolean in the common Fcb header. This boolean will be tested while owning the Fcb resource shared, and if it is FALSE, the Fast I/O routine returns FALSE as an indication that Fast I/O is not currently possible.

Once the file system detects that the last condition making Fast I/O impossible has been removed, then it should set the IsFastIoPossible boolean to TRUE again.

2.6 Use of the Wait Input Parameter

A number of the Cache Manager routines and the FsRtl routines implementing Fast I/O take a boolean Wait input parameter, and return a boolean result. Use of the Wait parameter is the same in all cases, and is explained here in detail. By far the most efficient operation is always afforded to synchronous callers, i.e., callers who supply Wait as **TRUE** signifying that it is ok to block. This design encourages callers to be multi-threaded in order to get parallel operation, rather than adding lots of threads to file systems and having to pay the expense of locking down and mapping buffers and then context switching to the next available file system thread.

The Wait parameter should be used as follows.

If the caller does not want to block (such as for disk I/O), then *Wait* should be supplied as **FALSE**. If *Wait* was supplied as **FALSE** and it is currently impossible to supply all of the requested data without blocking, then this routine will return **FALSE**. However, if the data is immediately accessible in the cache and no blocking is required, this routine supplies the data and returns **TRUE**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the caller supplies *Wait* as **TRUE**, then this routine is guaranteed to supply the data and return **TRUE**. If the data is immediately accessible in the cache, then no blocking will occur. Otherwise, the the data transfer from the file into the cache will be initiated, and the caller will be blocked until the data can be returned.

File system Fsd's should typically supply *Wait* = **TRUE** if they are processing a synchronous I/O request, or *Wait* = **FALSE** if they are processing an asynchronous request.

File system or Server Fsp threads should supply *Wait* = **TRUE**.

2.7 Use of Stream Files

All of the Cache Manager routines which have been presented take a file object as input in order to tell which file a particular operation is directed to. For normal user file opens, it is the user's own file object, which the file system initializes during create, which may be specified to all of the Cache Manager calls. For the case where a file system wishes to cache file system metadata, there is no user file object at hand.

For this case, the I/O system provides the capability of creating a "stream file object", to represent an arbitrary stream as defined by the file system. The file system simply calls IoCreateStreamFileObject (see the *Windows NT I/O System Specification*), and sets up the file object fields as described in Section 0. Note that in this case the common Fcb header and the section object pointers may generally not be resident in an Fcb, but rather in any structure convenient to the file system.

Once the stream file is created and the various pointer fields initialized, the file system may call **CcInitializeCacheMap** at any time to enable caching on this stream.

When done with the stream file, the file system should call **CcUninitializeCacheMap** to turn off caching on that file, and **ObDereferenceObject** with the address of the file object to cause it to subsequently get deleted.

2.8 File System Cleanup and Close Routines

Now that we have presented a walkthrough of the normal Cache Manager interaction, and presented the special case of how stream files may be used, it is important to complete the picture by explaining exactly what expectations are placed on the file system cleanup and close routines (which respond to the Irps with function codes IRP_MJ_CLEANUP and IRP_MJ_CLOSE).

Cleanup is called each time that the last user file handle to a given file object goes away. For normal user files, the file system is guaranteed to get exactly one cleanup call on a given file for each successful create operation which it performs. If the same file is opened and accessed by multiple users, then each open results in a separate file object, and separate cleanup calls on this file will be received as each user file handle is closed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Within the I/O system, there are various cases where a system component wishes to guarantee that a file object will not be deleted, even if the user closes its handle. It does this typically by calling **ObReferenceObjectByPointer** for the file object. When the system component no longer needs to rely on this file object it calls **ObDereferenceFileObject**. So, for example, a file object is referenced each time an I/O request is issued on it, and dereferenced each time the request is completed. The Cache Manager references a file object the first time the file is cached, and it dereferences the file object when no file objects have it cached, and there are no more dirty pages to flush. A final example is that Memory Management references a file object when a user or the Cache Manager creates a section for mapping that file, and it dereferences the file object when there are no more sections in existence for the file, and the last page has been removed from memory for the file. Note that regardless of how many times the Cache Manager and Memory Management is called for a given file, they only reference the first file object they were called with.

A file system is called to close a file object when the last reference to that file object goes away. This may not occur until some time after cleanup is received on the file object. For example if a system is idle for hours and memory management still has pages for a file that was once mapped, the close call will not occur during this time.

In order to keep track of all these file objects, and thus assist the cleanup and close routines to do the right thing, the file system is expected to maintain two counters in the Fcb. The first counter is essentially a count of user handles, but has been traditionally referred to as the "UncleanCount". The second count is a count of how many referenced file objects referenced exist for a given file, and it has been traditionally called the "OpenCount".

For normal user files, a file system should increment both the UncleanCount and the OpenCount on each successful create. The UncleanCount should be decremented on each cleanup call for a given file, and the OpenCount should be decremented on each close call for a given file.

For stream files, a file system generally only needs to maintain (at most) an OpenCount. Note that a cleanup call will be issued for a stream file object from within the call to **IoCreateStreamFileObject**. It is important to recognize this cleanup call in the file systems cleanup routine (by the way the stream file object was set up), and expediently dismiss it; i.e., simply Noop all cleanup calls to stream file objects. Generally it is also not necessary to maintain an OpenCount for stream files, as a single close call will be received when the one and only file object for the stream is dereferenced the last time.

The Cache Manager expects to be called at **CcUninitializeCacheMap** for each file object which was initialized. If a file is being truncated or deleted, the TruncateSize parameter should be correctly specified to this routine. It is acceptable to call **CcUninitializeCacheMap** on a file object that was never initialized; the Cache Manager will detect this case and do the right thing. In fact, if the file is being deleted or truncated, the Cache Manager definitely should be unconditionally called with the correct TruncateSize, because otherwise the file may not be purged from the cache properly if it had been earlier cached or otherwise mapped via a different file object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The only way that a file system really knows if both the Cache Manager and memory management (or potentially anyone else) are done with a file, is when the OpenCount finally goes to 0 in the close routine. This is the only time that it is safe for the file system to delete its Fcb, or whatever other structure the file system has associated with a given stream file.

2.9 Using Write Through and Cache Flushing

So far we have only discussed the Cache Manager's default method of lazy writing all dirty data. There are two different ways for either the user program or a file system to force dirty data out to disk and know when it is safely out there. These two methods are write through or flushing.

A user program specifies that it wants all operations on a given file performed write through by specifying FILE_WRITE_THROUGH in its Create options when it opens a file, which the file system can later see in the file object via the FO_WRITE_THROUGH flag in the file object flags. Once this flag is set in the file object, the copy write and Mdl write routines automatically perform write through. As a result, the Lazy Writer will never see dirty data modified through this file object and will never attempt to write any.

Now the only question that remains is, how is write through dealt with in conjunction with pin access? The current file systems in NT have chosen to write through all structure information that is modified as the result of performing an operation on a file object with FO_WRITE_THROUGH set. For such a file object, each time that a pinned Bcb is set dirty, **CcRepinBcb** is called at the same time to guarantee that the Bcb will not be deleted when it is unpinned. In addition, the file systems remember all Bcbs that they have repinned. When the file system request is complete, and all Bcbs have been unpinned and all resources have been released (both very important to prevent deadlocks), and just before completing the Irp, the file systems loop to call **CcUnpinRepinnedBcb** for each Bcb that was repinned. This call is made with the WriteThrough flag specified as TRUE. An unpinned Bcb causes the Bcb to be flushed, and the resulting I/O status is returned. This write through is synchronized with the Lazy Writer, and the Lazy Writer will not lazy write this page a second time.

Flushing is considerably simpler. A user request to flush file buffers results in a flush Irp to the file system. **CcFlushCache** may be called to immediately flush all dirty data to the file. In addition, a file system may choose to flush buffers in any cached file or stream file at any time by also calling **CcFlushCache**. Unlike write through, flushing is not synchronized with the Lazy Writer. However, this only means two things:

- o Any buffer which is currently dirty and pinned will not be flushed. If the file system does not eliminate this possibility by synchronizing this properly within itself, then the affected buffer will still eventually get Lazy Written.

- o If a dirty buffer which is waiting to be flushed by the Lazy Writer is flushed first, then the Lazy Writer will eventually go through the motions of flushing this buffer anyway, but the flush will be nooped by Memory Management, when it realizes that the buffer is no longer dirty.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The current NT file systems only use flushing to mark volumes clean when they have been idle for a while. For this case flushing activity is serialized with everything else on a volume and the first case above does not occur. The second case above can occur, but is benign.

2.10 Valid Data Length and File Size Considerations

Nearly every file system has system has separate concepts of Allocation Size (how much space is allocated to a file) and File Size (how far may a caller read in the file). Allocation Size will typically be a multiple of the disk sector size or allocation quantum (aka cluster size), while File Size may be any number of bytes.

Some file systems (such as HPFS) have, in addition, a concept called Valid Data Length, which is an indication of how much of the file has actually been initialized. Reading beyond Valid Data Length is allowed (unlike reading beyond File Size), however all zeros are returned in the buffer, regardless of what may actually be present on the respective allocated sectors on disk. Returning 0's is both an optimization (we do not have to read the sectors) as well as a security feature (the caller does not get to read the data that used to be in those sectors from some previous file).

It is a very good idea, even for file systems that do not have a concept of Valid Data Length, to present and maintain a concept of Valid Data Length in their implementation and in their interaction with the Cache Manager. This is advisable for both the optimization and security related reasons discussed above. Consider the frequent case where a user creates a file and is sequentially writing to the file. As each user write comes in, the file system typically has to check if it needs to extend the file allocation, and it also may want to advance the File Size early on for internal reasons. When it comes time to call the Cache Manager, say at **CcCopyWrite**, the Cache Manager has to get a page ready to receive the data, and the only way to do that is to fault the page in. This now results in a page fault read back to the file system from within the write path. Fortunately Resources, such as the one synchronizing the Fcb, allow recursive acquisition, so the read proceeds fine. The File Size may already be advanced, but clearly what the file system wants to do in this case is detect that the read is beyond Valid Data Length, so that no real read is required. The file system in this case should simply map the buffer and clear it, and complete the request.

Now, for file systems that actually record Valid Data Length on disk, this field should be updated in a reliable fashion such that even if the system dies, there are still no windows where someone will get to see uninitialized data after the system reboots. This is necessary to really make the file system secure. However, because of the Lazy Writer, the file system can not easily and reliably keep track of when it is safe to advance Valid Data Length, because it cannot make any assumptions about what order the Lazy Writer will flush data to disk. Therefore, the Lazy Writer calls the file system to inform it when it is safe to update Valid Data Length. It does this by issuing a **IRP_MJ_SET_INFORMATION** Irp on the file with **SetEndOfFileInformation** as the operation code and the **AdvanceOnly** flag set. (The **AdvanceOnly** flag can only be set by the Lazy Writer. This call instructs the file system that it can safely update ValidDataLength for the file to the specified size, but only if that would make the new ValidDataLength greater than the current value

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

(someone could have done a WriteThrough or a flush in the meantime which would already advance the ValidDataLength). In some cases, such as for stream files containing file system metadata, the file system simply wishes to consider the entire file to be valid, and it never wants to get the **SetEndOfFileInformation** calls described above. For this case it may specify a **NULL** pointer for ValidDataLength in the **CcInitializeCacheMap** call for this file. This will disable the Cache Manager's ValidDataLength processing just described. For normal files, however, file systems are recommended to support a concept of ValidDataLength in their implementation.

One final note about FileSize. In general paging I/O requests (**IRP_PAGING_IO** set in the Irp Flags) are unsynchronized with File Size Changes. This is true whether these requests emanate from the Cache Manager (especially the Lazy Writer) or whether they occur from user mapped files. Fortunately the rules a file system must follow are simple. On reads, paging I/O requests must obey end of file like anyone else; thus reads extending beyond FileSize should be truncated to the nearest allocation boundary beyond FileSize, and reads totally beyond FileSize should receive **STATUS_END_OF_FILE**. Paging I/O writes are not allowed to extend AllocationSize or FileSize; they are handled similarly to reads. Paging I/O writes extending beyond end of file should be truncated to the nearest allocation boundary beyond FileSize. Paging I/O writes starting beyond FileSize should be nooped with an immediate completion with **STATUS_SUCCESS**. Complete all successful writes with the Information field of the I/O status containing the requested byte count, whether all the bytes really were transferred or not.

2.11 Resource Locking Rules

Doing a caching strategy with a mapped file model is a fairly complex problem. The file system calls the Cache Manager, the Cache Manager calls Memory Management, at which point Memory Management sometimes has to call the file system again. Generally all of this activity stays within the same file. In spite of this complexity, at the time of this writing two disk-based file systems (FAT and HPFS), the CDRom file system, the Lan Manager Redirector, and the Lan Manager Server (through the calls in the next subsection) are all completed and running reliably using the Cache Manager.

Through the experiences gained with the above implementations, a set of resource locking rules has been refined, which seems to allow for good parallelism without deadlock. These rules are as follows:

- Since most activity begins in the file system, the first rule of preventing deadlock is that resources must be acquired in the order: file system resources, Cache Manager resources, Memory Management resources.

  Since some activity begins in the Lazy Writer as it processes its work queue, and since it is necessary for the Lazy Writer to own some its resources across calls to the file systems, the Cache Manager requires some very simple callbacks to allow it to acquire file system resources first before beginning to acquire its own resources. (See Section 0.)

  Some activity also begins in Memory Management, such as in the Modified

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Page Writer, or in the servicing of MM services. In general, Memory Management attempts to own no resources at all when it calls the file system.

- o The next rule is that the file system resources must support recursive acquisition, since some Cache Manager calls that the file system makes will sometimes result in recursive calls to the file system within the same thread. It's important to note that the recursive calls are never random, but rather logical consequences of the Cache Manager call being made; otherwise recursive resource acquisition could actually be dangerous! One example of a worst-case scenario: in the process of servicing a cached write request, the file system calls CcCopyWrite, which results in a recursive call to the file system for a noncached read to fault in the page to be written, then subsequently a call for a noncached write of the page if the file object is Write Through.

  All of the file systems currently use the executive resource package, which supports single-thread exclusive access or multi-thread shared access. Both exclusive and shared access support recursive acquisition. If an exclusive user recursively requests a resource shared, this is transparently turned into a recursive exclusive acquisition (there is only one release call). Finally, a non-recursive exclusive acquisition can be converted to shared access to allow greater sharing after completing a critical section. (Code which attempts to convert shared to exclusive is almost certain to cause deadlocks.) The calls are: **ExAcquireResourceExclusive**, **ExAcquireResourceShared**, **ExReleaseResource**, and **ExConvertExclusiveToShared**.

  Note that the file systems use some of the other synchronization mechanisms available in NT, but never across calls to the Cache Manager.

- o As further assistance, the following table attempts to summarize how the Cache Manager expects the the Fcb to be acquired when it is called at its various entry points. This table was built from the actual usage in HPFS and FAT. Note that the file systems should always attempt to own no other resources exclusive (such as a resource synchronizing allocation on the volume) across calls to the Cache Manager.

  Multiple options in the table below are separated by "/". In the table E = exclusive, S = shared, 0 = unowned, and - = don't care.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Routine	Fcb Res
CcInitializeCacheMap	E/S
CcUninitializeCacheMap	E
CcExtendCachedFileSize	E
CcExtendCacheSection	E
CcFlushCache	E/0
CcPurgeFromWorkingSet	-
CcPurgeCacheSection	-
CcTruncateCachedFileSize	E
CcZeroData	E
CcRepinBcb	-
CcUnpinRepinnedBcb	0
CcIsFileCached	-
CcReadAhead	S
CcSetAdditionalCacheAttributes	E
CcCopyRead	S
CcCopyWrite	E/S
CcMdlRead	S
CcMdlReadComplete	-
CcPrepareMdlWrite	E/S
CcMdlWriteComplete	0
CcPinRead	-
CcMapData	-
CcPinMappedData	-
CcPreparePinWrite	-
CcSetDirtyPinnedData	-
CcUnpinData	-

In addition, the caller should have nothing pinned (repinned is ok) when calling **CcExtendCacheSection** or **CcUnpinRepinnedBcb**.

2.12 Network File Server Interfaces

There is not a lot to say here about how a network file server should use the Cache Manager, as this occurs primarily by virtue of the fact that the file server calls a local file system which is already using the Cache Manager. However, it is quickly worth mentioning that there are basically two ways for a Server to access cached files. Note that in any case servers will tend to open files, close files, and do all other operations except read and write by calling the same file APIs that any other local program would call.

The first alternative for reading and writing file data in a server is to also issue the normal **NtReadFile** and **NtWriteFile** operations. This is not a bad approach, as the server will still benefit from the Fast I/O operations implemented in these services. However, note that all data will be copied into and out of the cache; there is no opportunity to get at the Mdl interfaces at this level.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The second alternative assumes that the Server is running as a kernel-mode process, just like the current Lan Manager Server and all of the file systems. Running in kernel mode the server is able to directly call the FsRtl Fast I/O interfaces layers to either the Cache Manager copy interfaces or Mdl interfaces. The FsRtl interfaces are nearly identical to the respective Cc interfaces documented in this paper; the names are the same except that the Cc prefix is replaced by FsRtl. The difference is that the FsRtl interfaces perform the necessary synchronization with the file system via the Fcb resource, and they also perform a few simple checks (such as IsFastIoPossible as described in Section 0). If the FsRtl routine cannot perform the specified request, then it returns FALSE. If the Server receives FALSE from an FsRtl Fast I/O routine, then it should build the same request in the form of an Irp and queue it directly to the file system via **IoCallDriver** (see the *Windows NT I/O System Specification* and Section 0).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. File System Maintenance Functions (FSSUP)

3.1 CcInitializeCacheMap

This routine is intended to be called by File Systems only. It initializes the Cache Manager Data structures for data caching. It should be called the first time a File Stream which is to be cached is read or written, or any time the stream is about to be written and it is not already cached (FileObject->PrivateCacheMap == NULL).

The Fcb should be acquired either shared or exclusive when this routine is called.

The three size parameters passed will be captured in the Shared Cache Map, and they must be updated as described later if they change.

If a window to the file cannot be mapped in the normal system cache, then it will be mapped to the specified process, which should presumably be the file system's Fsp process.

The callbacks are described in the next subsection.

VOID
CcInitializeCacheMap (
  IN PFILE_OBJECT *FileObject,*
  IN PLARGE_INTEGER *AllocationSize*,
  IN PLARGE_INTEGER *FileSize,*
  IN PLARGE_INTEGER *ValidDataLength* **OPTIONAL,**
  IN PEPROCESS *Process,*
  IN BOOLEAN *PinAccess,*
  IN PCACHE_MANAGER_CALLBACKS *Callbacks,*
  IN PVOID *LazyWriteContext,*
  IN PVOID *CloseContext*
  );

Parameters:

FileObject - A pointer to the file object for the stream to be cached.

AllocationSize - The size of the file to be cached. This must be greater than or equal to the actual size of the file. It might be greater, for example, if the file is being created, or may be extended. If supplied as 0, it will be defaulted by the Cache Manager.

FileSize - The exact File Size of the file, beyond which it may not be read.

ValidDataLength - The initialized portion of the file, beyond which 0's must be returned if read (up to FileSize). This number also controls when the Lazy Writer should call Set Information File to advance valid data length. If the caller wants to consider all data valid and does not want callbacks, it can specify **NULL** for this pointer (please refer to Section 0).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Process - Pointer to the process to which the view should be mapped, if it cannot be mapped in system space. This should typically be the Fsp process itself.

PinAccess - FALSE if file will be used exclusively for Copy and Mdl access, or TRUE if file will be used for Pin access. (Files for Pin access are mapped entirely in one view, as it is assumed that the caller must access multiple areas of the file at once. Therefore, it is a good idea organize files for Pin Access into a number of small files.)

Callbacks - Pointer to a vector of Callbacks used by the Lazy Writer (see next subsection)

LazyWriteContext - Parameter to pass to Lazy Write and Read Ahead callbacks.

CloseContext - Parameter to pass to Close callbacks

3.1.1 Cache Manager Callbacks

The Cache Manager must set rules for locking order in order to prevent deadlocks. At a high level these rules are that first the file system is allowed to acquire its resources, then the Cache Manager is allowed to acquire resources, and finally in the worst case Memory Management may also acquire resources. These rules work perfectly well in most cases, since most activity starts in the file system to begin with. One case where there is a problem, however, is in the Lazy Writer. The Lazy Writer must own some Cache Manager resources prior to calling the file system.

To keep this case from producing deadlocks, the Cache Manager requires a set of callbacks to allow the Lazy Writer to acquire any necessary file system resources first, before it begins to acquire its own. This allows the Lazy Writer to continue to follow the locking rules, and prevent deadlocks.

To this end, **CcInitializeCacheMap** takes a pointer to a vector of callback addresses, and two different callback parameters, as defined below:

This routine is called by the Lazy Writer prior to calling **CcUninitializeCacheMap**, since this may result in a Close call to the file system. The context parameter supplied is whatever the file system passed as the CloseContext parameter when it called CcInitializeCacheMap.

```
typedef
BOOLEAN (*PACQUIRE_FOR_CLOSE) (
            IN PVOID Context,
            IN BOOLEAN Wait
            );
```

This routine releases the Context acquired above.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef
VOID (*PRELEASE_FROM_CLOSE) (
            IN PVOID Context
            );
```

This routine is called by the Lazy Writer prior to doing a write, since this will require some file system resources associated with this cached file. The context parameter supplied is whatever the FS passed as the LazyWriteContext parameter when it called **CcInitializeCacheMap**.

```
typedef
BOOLEAN (*PACQUIRE_FOR_LAZY_WRITE) (
            IN PVOID Context,
            IN BOOLEAN Wait
            );
```

This routine releases the Context acquired above.

```
typedef
VOID (*PRELEASE_FROM_LAZY_WRITE) (
            IN PVOID Context
            );
```

This routine is called by the Lazy Writer prior to doing a readahead. It also uses the LazyWriteContext parameter.

```
typedef
BOOLEAN (*PACQUIRE_FOR_READ_AHEAD) (
            IN PVOID Context,
            IN BOOLEAN Wait
            );
```

This routine releases the Context acquired above.

```
typedef
VOID (*PRELEASE_FROM_READ_AHEAD) (
            IN PVOID Context
            );
```

Finally, this is the complete callback vector, a pointer to which must be passed to **CcInitializeCacheMap**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef struct _CACHE_MANAGER_CALLBACKS {

    PACQUIRE_FOR_CLOSE AcquireForClose;
    PRELEASE_FROM_CLOSE ReleaseFromClose;
    PACQUIRE_FOR_LAZY_WRITE AcquireForLazyWrite;
    PRELEASE_FROM_LAZY_WRITE ReleaseFromLazyWrite;
    PACQUIRE_FOR_READ_AHEAD AcquireForReadAhead;
    PRELEASE_FROM_READ_AHEAD ReleaseFromReadAhead;

    } CACHE_MANAGER_CALLBACKS, *PCACHE_MANAGER_CALLBACKS;
```

3.2 CcUninitializeCacheMap

This routine uninitializes the previously initialized File Stream. This routine is only intended to be called by File Systems. It should be called when the File System receives a cleanup call on the File Object.

A File System which supports data caching must always call this routine whenever it closes a file that it is trying to delete, whether it cached the file on the given file object or not. This is because the final cleanup of a file related to truncation or deletion of the file, can only occur on the last close, whether the last closer cached the file or not. Any time **CcUninitializeCacheMap** is called on a file object for which **CcInitializeCacheMap** was never called, the call is benign.

CcUninitializeCacheMap does the following:

- o If a File Stream was initialized on this File Object, it is uninitialized (unmap any views, delete section, and delete Cache Manager structures).
- o On the last Cleanup, if the file has been deleted, the Section is forced closed. If the file has been truncated, then the truncated pages are purged from the cache.

Some times a file system may want pages of the file removed from the cache, even though the file is still open. Examples in the case of a local file system might be if the file has been truncated, a file's media has been removed from the drive. For a network file system client, examples might be if an opportunistic locking protocol dictates that a file may no longer be cached, or perhaps if a virtual circuit goes down. For this purpose **CcUnitializeCacheMap** takes a TruncateSize parameter, which, if specified, causes all pages from the specified file offset on to be purged (removed) from the cache.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
CcUninitializeCacheMap (
IN PFILE_OBJECT *FileObject,*
IN PLARGE_INTEGER *TruncateSize* **OPTIONAL**,
IN PCACHE_UNINITIALIZE_EVENT *UninitializeCompleteEvent* **OPTIONAL**
);

Parameters:

FileObject - File Object which was previously supplied to **CcInitializeCacheMap**.

TruncateSize - If specified, all pages should be purged (removed) from the cache starting at, and including, the specified address.

UninitializeCompleteEvent - If specified, this event will be set when the uninitialize is complete, since it may not be complete upon return. If the caller wishes to wait on this event, he must absolutely guarantee that he owns no resources, as this could lead to deadlocks. The format of this structure is:

```
typedef struct _CACHE_UNINITIALIZE_EVENT {
struct _CACHE_UNINITIALIZE_EVENT *Next;
KEVENT Event;
} CACHE_UNINITIALIZE_EVENT, *PCACHE_UNINITIALIZE_EVENT;
```

Returns:

FALSE - if Section was not closed. In this case, if the caller really cares, it may wish to specify and wait on the UninitializeCompleteEvent.

TRUE - if Section was closed.

3.3 CcExtendCachedFileSize

This routine must be called whenever a file has been extended to reflect this extension in the Cache Manager data structures and the underlying section. Calling this routine has a benign effect if the current size of the file is already greater than or equal to FileSize. The Cache Manager must know the correct file size to make the fast read paths work correctly.

VOID
CcExtendCachedFileSize (
IN PFILE_OBJECT *FileObject,*
IN PLARGE_INTEGER *FileSize*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileSize - Supplies the new file size for the file.

3.4 CcExtendCacheSection

This routine must be called whenever the allocation for a file has been extended to reflect this extension in the Cache Manager data structures and the underlying section. Calling this routine has a benign effect if the current allocation size of the file is already greater than or equal to NewSize. The Cache Manager must know the correct allocation size in order to insure that the underlying section is large enough.

BOOLEAN
CcExtendCachedFileSize (
    IN PFILE_OBJECT *FileObject,*
    IN PLARGE_INTEGER *NewSize*,
    IN BOOLEAN *Wait*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

NewSize - Supplies the new allocation size for the file.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Returns:

FALSE - if Wait was supplied as **FALSE**, and the extend was not possible without blocking.

TRUE - if the extend was successfully completed.

3.5 CcFlushCache

This routine may be called to flush dirty data from the cache to the cached file on disk. Any byte range within the file may be flushed, or the entire file may be flushed by omitting the FileOffset parameter.

This routine does not take a Wait parameter; the caller should assume that it will always block.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
VOID
CcFlushCache (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER FileOffset OPTIONAL,
    IN ULONG Length,
    OUT PIO_STATUS_BLOCK IoStatus OPTIONAL
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - If this parameter is supplied (not NULL), then only the byte range specified by FileOffset and Length are flushed.

Length - Defines the length of the byte range to flush, starting at FileOffset. This parameter is ignored if FileOffset is specified as NULL.

IoStatus - The I/O status resulting from the flush operation.

3.6 CcPurgeFromWorkingSet

This routine which may optionally be used to purge all of the pages of a file from the system cache or Fsp working set. The pages do not immediately leave memory, but simply become eligible for replacement.

```
BOOLEAN
CcPurgeFromWorkingSet (
    IN PFILE_OBJECT FileObject,
    IN BOOLEAN Wait
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Returns:

FALSE - if Wait was supplied as **FALSE**, and the extend was not possible without blocking.

TRUE - if the extend was successfully completed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.7 CcPurgeCacheSection

This routine forcibly purges pages from the cache, automatically uninitializing *all* file objects which have cached this file if necessary. It is meant for infrequent use when dealing with such things as removeable media. Note that this routine is called automatically if the Cache Manager is notified of a file truncation via **CcTruncateCachedFileSize**.

```
VOID
CcPurgeCacheSection (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER PurgeSize
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

PurgeSize - The offset at which the purge is to begin. If not on a page boundary, the page at PurgeSize is first flushed.

3.8 CcTruncateCachedFileSize

This routine must be called any time a local file system truncates a file. It informs the Cache Manager of the new size. If any of AllocationSize, FileSize, or ValidDataLength are larger than this number, they are reduced. Any pages beyond this point are purged from the cache.

```
VOID
CcTruncateCachedFileSize (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER NewSize
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

NewSize - Supplies the new size for the file.

3.9 CcZeroData

This routine may be called to zero a given byte range in a file. As a general service, it may even be called by file systems to zero byte ranges in files which are not cached.

Up to some reasonable amount, this routine will simply attempt to zero data in the cache, and let it be lazy written out. However, beyond a certain size, or for the entire range if the file is not cached, the pages of the file are zeroed by writing to

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

them directly on disk. Note that for files which are not cached, the caller must guarantee that the specified StartOffset is on a physical sector boundary for the underlying disk, otherwise the disk driver will return an error and this routine will raise that error status.

BOOLEAN
CcZeroData (
    IN PFILE_OBJECT *FileObject,*
    IN PLARGE_INTEGER *StartOffset,*
    IN PLARGE_INTEGER *EndOffset,*
    IN BOOLEAN *Wait,*
    OUT PIO_STATUS_BLOCK *IoStatus*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

StartOffset - Supplies the file offset at which zeroing is to begin. If the file is not cached, this offset must be on a hardware sector boundary.

EndOffset - Supplies the file offset at which zeroing is to end.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

IoStatus - Returns the I/O status from the zeroing operation.

Returns:

FALSE - if Wait was supplied as **FALSE**, and the extend was not possible without blocking.

TRUE - if the extend was successfully completed.

3.10 CcRepinBcb

This routine may be called to guarantee that the specified Bcb does not go away. This Bcb address must be one previously returned by either **CcPinRead**, **CcPreparePinWrite**, or **CcPinMappedData**. The caller must subsequently call **CcUnpinRepinnedBcb** for this Bcb. This sequence is usually done in connection with a write through file object, however it may also be done to insure that a buffer does not leave memory to facilitate possible error recovery.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
BOOLEAN
CcRepinBcb (
    IN PBCB Bcb
    )
```

Parameters:

Bcb - A previously returned pinned Bcb.

3.11 CcUnpinRepinnedBcb

This routine must be called to release a Bcb which was previously specified in **CcRepinBcb**. It releases the Bcb, optionally writing it through to disk first.

```
VOID
CcUnpinRepinnedBcb (
    IN PBCB Bcb,
    IN BOOLEAN WriteThrough,
    OUT PIO_STATUS_BLOCK IoStatus
    )
```

Parameters:

Bcb - Address of the Bcb

WriteThrough - Specified as **TRUE**, if the data represented by the Bcb should first be written through

IoStatus - Returns the I/O status of the write, if WriteThrough was specified

3.12 CcIsFileCached

This routine is the approved way to determine if a file is cached by any FileObject, whether it is cached by the input file object or not.

Note, if the caller wishes to determine if a given file object itself has been initialized for caching, he should simply test FileObject->PrivateCacheMap. If this field is not NULL, then the file object has been initialized for caching.

```
BOOLEAN
CcIsFileCached (
    IN PFILE_OBJECT FileObject
    )
```

Parameters:

FileObject - The file object in question.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Returns:

> **FALSE** - if no file object has the file cached.
>
> **TRUE** - if at least one file object has the file cached.

3.13 CcReadAhead

This routine is intended to be called by file systems, after a successful **CcCopyRead** or **CcMdlRead**. The caller essentially specifies information about the previous read. **CcReadAhead** maintains history information about a small number of recent calls for this file object, and attempts to detect if read ahead would currently be adviseable, and if so, whether or not the determined read ahead has already been performed.

If the routine decides that it should perform some read ahead, then a read ahead work request is queued off to one of the Cache Manager's worker threads, in order to not tie up the current thread.

VOID
CcReadAhead (
  IN PFILE_OBJECT *FileObject*,
  IN PLARGE_INTEGER *FileOffset*,
  IN ULONG *Length*,
  IN ULONG *StillNeed*
)

Parameters:

> *FileObject* - A file object for which **CcInitializeCacheMap** has been previously called.
>
> *FileOffset* - Byte offset in file where the last read was just performed.
>
> *Length* - The number of bytes successfully returned to the reader.
>
> *StillNeed* - If the read was **CcCopyRead**, this parameter should specify 0. If the read was **CcMdlRead**, then the caller had specified both a Length and MinimumLength that he desired, and we may therefore have given him less than Length. If so, this parameter should specify the Length requested in **CcMdlRead** minus the length we returned to him.

3.14 CcSetAdditionalCacheAttributes

This routine may be called to disable read ahead or lazy write on a file object. Disabling read ahead is always safe. The caller must guarantee that if it disables lazy write, that it will write all dirty pages eventually for the entire file by flushing. A file system should clearly not disable lazy write just because someone opens the file write through, because disabling lazywrite applies to the file itself (not a given file

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

object), and someone else may open the file without write through. Note also that write through is properly synchronized with the Lazy Writer anyway.

```
VOID
CcSetAdditionalCacheAttributes (
    IN PFILE_OBJECT FileObject,
    IN BOOLEAN DisableReadAhead,
    IN BOOLEAN DisableLazyWrite
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

DisableReadAhead - If specified as **TRUE**, read ahead will be disabled.

DisableLazyWrite - If specified as **TRUE**, lazy writing will be disabled.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Copy Interface (COPYSUP)

4.1 CcCopyRead

This routine attempts to copy the specified file data from the cache into the output buffer, and deliver the correct I/O status.

BOOLEAN
CcCopyRead (
    IN PFILE_OBJECT *FileObject*,
    IN PLARGE_INTEGER *FileOffset*,
    IN ULONG *Length*,
    IN BOOLEAN *Wait*,
    OUT PVOID *Buffer*,
    OUT PIO_STATUS_BLOCK *IoStatus*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Buffer - Pointer to output buffer to which data should be copied.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if Wait was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

4.2 CcCopyWrite

This routine attempts to copy the specified file data from the specified buffer into the Cache, and deliver the correct I/O status. If the file object has **FO_WRITE_THROUGH** set, then the data will have been written through to disk upon return.

There is one optimization that is important to note. In **CcCopyWrite**, a fast compare is made to see if the caller happens to be writing the same data that already exists in the file at that point, a common case in certain applications. On

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

the first different byte that is seen, a move of the new data into the cache begins at that point. However, if the buffer is completely the same, then the write is essentially nooped. This optimization does not occur if the buffer was already dirty anyway, or the write is beyond ValidDataLength, or the file does not support ValidDataLength (**NULL** pointer was passed to **CcInitializeCacheMap**). Given these checks, this optimization should always be safe, but the file system should be aware of this optimization none the less.

BOOLEAN
CcCopyWrite (
IN PFILE_OBJECT *FileObject*,
IN PLARGE_INTEGER *FileOffset*,
IN ULONG *Length*,
IN BOOLEAN *Wait*,
IN PVOID *Buffer*,
IN PLSN *Lsn* **OPTIONAL**,
OUT PIO_STATUS_BLOCK *IoStatus*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file to receive the data.

Length - Length of data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Buffer - Pointer to input buffer from which data should be copied.

Lsn - An optional pointer reserved for future support. Should be supplied as **NULL**.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed when *WriteThrough* = **FALSE**, otherwise the actual I/O status from the Write is returned.)

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the data was not copied.

TRUE - if the data has been copied.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Mdl Interface (MDLSUP)

5.1 CcMdlRead

This routine attempts to lock the specified file data in the cache and return a description of it in an Mdl along with the correct I/O status.

If not all of the data can be delivered, but at least *MinimumLength* can be, then all of the data currently available is still delivered.

If the caller does not want to block, then *Wait* should be supplied as **FALSE**. If *Wait* was supplied as **FALSE** and it is currently impossible to supply the minimum requested data without blocking, then this routine will return **FALSE**. However, if the minimum amount of data is immediately accessible in the cache and no blocking is required, this routine locks the data and returns **TRUE**.

If the caller supplies *Wait* as **TRUE**, then this routine is guaranteed to lock at least *MinimumLength* data and return **TRUE**. If at least *MinimumLength* is immediately accessible in the cache, then no blocking will occur, and all of the available data up to *Length* will be returned. Otherwise, a data transfer from the file into the cache will be initiated for all missing data up to *MinimumLength*, and the caller will be blocked until the data can be returned.

File system Fsd's will typically not use **CcMdlRead**, except to implement the **IRP_MN_MDL** subfunction of read.

File Server threads do not call this routine directly as that is not safe. They may call **FsRtlMdlRead**, which has essentially the same interface. They may also queue an Irp with **IRP_MN_MDL** set in the subfunction of an **IRP_MJ_READ** request. In this case they must pass *MinimumLength* in via the Irp->IoStatus.Information field. They can intercept the Irp completion via a completion routine (see the *Windows NT I/O System Specification*) and read the I/O status and get the Mdl from Irp->MdlAddress. It must then clear this field, or else not allow the Irp completion to continue.

After the caller is done with the data, it must call **CcMdlReadComplete** to free Cache Manager resources.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
CcMdlRead (
    IN PFILE_OBJECT *FileObject*,
    IN PLARGE_INTEGER *FileOffset*,
    IN ULONG *Length*,
    IN ULONG *MinimumLength*,
    IN BOOLEAN *Wait*,
    OUT PMDL **MdlChain*,
    OUT PIO_STATUS_BLOCK *IoStatus*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

MinimimumLength - Minimum data to be guaranteed on return if this routine returns **TRUE**.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

MdlChain - Returns a pointer to an Mdl chain describing the desired data.

IoStatus - Pointer to standard I/O status block to receive the status and byte length of the returned data for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if Wait was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

5.2 CcMdlReadComplete

This routine must be called after the call to **CcMdlRead**, when the Mdl is no longer required. It performs any cleanup that is necessary from the **CcMdlRead**.

Note that this routine does not assume that the calls to **CcMdlReadComplete** will occur in the same order as the calls to **CcMdlRead**, however it does assume that each call to **CcMdlRead** will eventually be followed by a call to this routine.

File systems only use this routine to implement the **IRP_MN_MDL_COMPLETE** subfunction of **IRP_MJ_READ**. Servers may generate this Irp with the *MdlChain* returned from **CcMdlRead** in Irp->MdlAddress, or they may call

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FsRtlMdlReadComplete. (Either of these options are available regardless of how they called **CcMdlRead**.)

```
VOID
CcMdlReadComplete (
    IN PFILE_OBJECT FileObject,
    IN PMDL MdlChain
    );
```

Parameters:

FileObject - Same file object pointer supplied to **CcMdlRead**.

MdlChain - Mdl chain returned from **CcMdlRead**.

5.3 CcPrepareMdlWrite

This routine attempts to lock the specified file data in the cache and return a description of it in an Mdl along with the correct I/O status. Pages to be completely overwritten may be satisfied with empty pages.

File system Fsd's will typically not use **CcMdlWrite**, except to implement the **IRP_MN_MDL** subfunction of write.

File Server threads do not call this routine directly as that is not safe. They may call **FsRtlPrepareMdlWrite**, which has essentially the same interface. They may also queue an Irp with **IRP_MN_MDL** set in the subfunction of an **IRP_MJ_WRITE** request. They can intercept the Irp completion via a completion routine (see the *Windows NT I/O System Specification*) and read the I/O status and get the Mdl from Irp->MdlAddress. It must then clear this field, or else not allow the Irp completion to continue.

After the caller is done with the data, it must call **CcMdlWriteComplete** to free Cache Manager resources.

```
BOOLEAN
CcPrepareMdlWrite (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN BOOLEAN Wait,
    OUT PMDL *MdlChain,
    OUT PIO_STATUS_BLOCK IoStatus
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

MdlChain - On output it returns a pointer to an Mdl chain describing the desired data.

IoStatus - Returns I/O status from potential read required to prepare the data.

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the pages were not delivered

TRUE - if the pages are being delivered

5.4 CcMdlWriteComplete

This routine must be called after a call to **CcPrepareMdlWrite**. The caller supplies the *ActualLength* of data that it actually wrote into the buffer, which may be less than or equal to the *Length* specified in **CcPrepareMdlWrite**.

This call does the following:

- Makes sure the data up to *ActualLength* eventually gets written. If the file object is not write through, the data will not be written immediately and *IoStatus* will simply say *ActualLength* bytes were successfully written on return (even though they were not). This strategy allows the caller to always check the I/O status, and know that if it got an error, the file object must be write through. If the file object is write through, then the data is written synchronously, and the appropriate *IoStatus* is returned.
- Unlocks the pages and deletes the *MdlChain*

File systems only use this routine to implement the **IRP_MN_MDL_COMPLETE** subfunction of **IRP_MJ_WRITE**. Servers may generate this Irp with the *MdlChain* returned from **CcPrepareMdlWrite** in Irp->MdlAddress, or they may call **FsRtlMdlWriteComplete**. (Either of these options are available regardless of how they called **CcPrepareMdlWrite**.)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
BOOLEAN
CcMdlWriteComplete (
    IN PFILE_OBJECT FileObject,
    IN ULONG ActualLength,
    IN PMDL MdlChain,
    IN BOOLEAN Wait,
    OUT PIO_STATUS_BLOCK IoStatus
    );
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

ActualLength - Length of data actually transferred.

MdlChain - Mdl chain returned from **CcPrepareMdlWrite**.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

IoStatus - Returns success, or the actual I/O status from Write Through.

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the pages were not delivered

TRUE - if the pages are being delivered

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6. Pin Interface (PINSUP)

The Pin interface is implemented specifically for file systems which are implementing a particular file structure on external media. In a sense it is a more primitive interface than the Copy or Mdl interfaces which always guarantee that data is never left locked in memory, and which automatically deal with cases where the desired data crosses page boundaries, and so on. Therefore there are a couple of special rules that users of this interface must obey.

In general, the Pin interface allows a range of bytes to be locked/pinned in memory (or in the lighter-weight case mapped), and subsequently accessed directly by virtual address. While the data is pinned or mapped system resources are being held. The Cache Manager absolutely relies on the File System to guarantee that it will free these resources by calling **CcUnpinData**.

Forgetting to unpin data is a serious error which can lead to system failure. One approach has proven to be bullit-proof in guaranteeing that file systems never forget to unpin data. By initializing all Bcb variables in a procedure to **NULL**, and nesting all calls which may fail or Raise within a try statement of a try-finally clause, all non-**NULL** Bcbs may be unpinned in the finally clause on the way out. This means they will be unpinned whether the try statement is exited normally, or whether some type of exception occurs which causes the procedure to be unwound.

There is another rule which is a bit more subtle, but for most cases not a problem. Whenever a file system maps or pins a range of bytes in one request that are present on one or more pages, it is invalid for that file system to *ever* make a subsequent request to map or pin a range of bytes in this stream that includes a page from the first request along with a page that was not included in the first request. The reason for this is somewhat due to internal details, but here is a simplified explanation. Once the Cache Manager completes the first request, he has "delivered" this data at a particular range of virtual addresses. If the second request comes along and overlaps the first request, but demands at least one additional page at the beginning or end, it is impossible in general for the Cache Manager to guarantee that it can deliver the new page(s) at contiguous virtual addresses. In the worst case the new page(s) could currently be being accessed as part of another request at a different virtual range. In reality the Cache Manager tries to avoid doing dynamic mapping, but in addition to the potential mapping problems the internal use of Bcbs also restricts overlapping requests.

6.1 CcPinRead

This routine attempts to lock/pin the specified file data in the cache. If successful (returning **TRUE**), a pointer is returned to the desired data in the cache. This routine is intended for File Systems.

If the caller subsequently modifies the data, it should call **CcSetDirtyPinnedData**.

In any case, the caller MUST subsequently call **CcUnpinData**. Naturally if **CcPinRead**, **CcMapData**, or **CcPreparePinWrite** were called multiple times for the same data, **CcUnpinData** must be called the same number of times.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The returned *Buffer* pointer is valid until the data is unpinned, at which point it is invalid to use the pointer further.

BOOLEAN
CcPinRead (
    IN PFILE_OBJECT *FileObject*,
    IN PLARGE_INTEGER *FileOffset*,
    IN ULONG *Length*,
    IN BOOLEAN *Wait*,
    OUT PVOID **Bcb*,
    OUT PVOID **Buffer*,
    OUT PIO_STATUS_BLOCK *IoStatusBlock*
)

<u>Parameters:</u>

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Bcb - On the first call this returns a pointer to a Bcb parameter which must be supplied as input on all subsequent calls for this buffer.

Buffer - Returns pointer to desired data, valid until the buffer is unpinned or freed.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

<u>Returns:</u>

FALSE - if *Wait* was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

6.2 CcMapData

This routine attempts to map the specified file data in the cache. If successful (returning **TRUE**), a pointer is returned to the desired data in the cache. Mapping data is considerably cheaper than pinning it, however mapped data may not be modified. One either needs to call **CcPinRead** (or **CcPreparePinWrite**) instead if one knows in advance that the data is to be modified, or call **CcPinMappedData** prior to modifying the data and setting it dirty. The caller must not modify the data or set it dirty before calling **CcPinMappedData**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This routine is intended for File Systems.

The caller MUST subsequently call **CcUnpinData** once with the Bcb returned from this call, or the modified Bcb returned from **CcPinMappedData** if that routine was called. Naturally if **CcPinRead**, **CcMapData**, or **CcPreparePinWrite** were called multiple times for the same data, **CcUnpinData** must be called the same number of times.

The returned *Buffer* pointer is valid until the data is unpinned, at which point it is invalid to use the pointer further.

BOOLEAN
CcMapData (
  IN PFILE_OBJECT *FileObject*,
  IN PLARGE_INTEGER *FileOffset*,
  IN ULONG *Length*,
  IN BOOLEAN *Wait*,
  OUT PVOID **Bcb*,
  OUT PVOID **Buffer*,
  OUT PIO_STATUS_BLOCK *IoStatusBlock*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Bcb - On the first call this returns a pointer to a Bcb parameter which must be supplied as input on all subsequent calls for this buffer.

Buffer - Returns pointer to desired data, valid until the buffer is unpinned or freed.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.3 CcPinMappedData

This routine attempts to pin the specified file data which may have previously only been mapped. If successful (returning **TRUE**), a pointer is returned to the desired data in the cache. The data is guaranteed to stay at the same virtual address. If **CcPinMappedData** has already been called for this data or the data was actually pinned in the first place (both cases determined from the *Bcb* IN OUT parameter), then this call is benign. Also note that Bcbs that are either pinned or mapped have to be unpinned, and a call to this routine does not mean that **CcUnpinData** has to be called an additional time, in fact it should not.

Note that *Bcb* is an IN OUT parameter, and that its value may in fact change. If so, it is the new value that must be specified to **CcSetDirtyPinnedData** or **CcUnpinData**; the caller should avoid making copies of the Bcb prior to this call.

This routine is intended for File Systems.

If the caller subsequently modifies the data, it should call **CcSetDirtyPinnedData**.

In any case, the caller MUST subsequently call **CcUnpinData**. Naturally if **CcPinRead**, **CcMapData**, or **CcPreparePinWrite** were called multiple times for the same data, **CcUnpinData** must be called the same number of times.

BOOLEAN
CcPinMappedData (
  IN PFILE_OBJECT *FileObject*,
  IN PLARGE_INTEGER *FileOffset*,
  IN ULONG *Length*,
  IN BOOLEAN *Wait*,
  IN OUT PVOID **Bcb*,
  OUT PIO_STATUS_BLOCK *IoStatusBlock*
)

<u>Parameters:</u>

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Bcb - On the first call this returns a pointer to a Bcb parameter which must be supplied as input on all subsequent calls for this buffer.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

6.4 CcPreparePinWrite

This routine attempts to lock the specified file data in the cache and return a pointer to it along with the correct I/O status. Pages to be completely overwritten may be satisfied with empty pages.

When this call returns with **TRUE**, the caller may immediately begin to transfer data into the buffers via the Buffer pointer. The buffer will already be marked dirty.

The caller MUST subsequently call **CcUnpinData**. Naturally if **CcPinRead** or **CcPreparePinWrite** were called multiple times for the same data, **CcUnpinData** (or **CcFreePinnedData**) must be called the same number of times.

The returned *Buffer* pointer is valid until the data is unpinned, at which point it is invalid to use the pointer further.

BOOLEAN
CcPreparePinWrite (
IN PFILE_OBJECT *FileObject*,
IN PLARGE_INTEGER *FileOffset*,
IN ULONG *Length*,
IN BOOLEAN *Zero*,
IN BOOLEAN *Wait*,
IN PLSN *Lsn* **OPTIONAL**,
OUT PVOID **Bcb*,
OUT PVOID **Buffer*,
OUT PIO_STATUS_BLOCK *IoStatus*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Zero - If supplied as **TRUE**, the buffer will be zeroed on return.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Lsn - An optional pointer reserved for future support. Should be supplied as **NULL**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Bcb - This returns a pointer to a Bcb parameter which must be supplied as input to **CcPinWriteComplete**.

Buffer - Returns pointer to desired data, valid until the buffer is unpinned or freed.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the pages were not delivered

TRUE - if the pages are being delivered

6.5 CcSetDirtyPinnedData

This routine declares that the data previously read via a call to **CcPinRead** has been modified. It is important to call this routine to insure that the data will eventually be written to disk in a timely manner.

VOID
CcSetDirtyPinnedData (
  IN PVOID *Bcb*,
  IN PLSN *Lsn* **OPTIONAL**,
)

Parameters:

Bcb - *Bcb* parameter returned from a call to **CcPinRead**.

Lsn - An optional pointer reserved for future support. Should be supplied as **NULL**.

6.6 CcUnpinData

This routine must be called after each call to **CcPinRead**, **CcMapData** or **CcPreparePinWrite**. It unlocks the data from the cache, enabling it to be written if it is dirty. Data will never be written while it is pinned.

VOID
CcUnpinData (
  IN PVOID *Bcb*
)

Parameters:

Bcb - Bcb parameter returned from the last call to **CcPinRead**, **CcMapData** (possibly modified by **CcPinMappedData**) or **CcPreparePinWrite**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7. Revision History

Original Draft 1.0, February 3, 1990

Revision Draft 1.1, March 5, 1990

- Minor changes plus incorporate review comments
- Addition of MmDeclareWsRoutines
- Addition of Section on further Memory Management Requirements
- Addition of entire

Revision Draft 1.2, June 15, 1990

- Complete rewrite except for the first half of Page 1, to reflect the actual implementation driven by the Design Review meeting for Draft 1.1.

Revision Draft 1.3, October 27, 1991

- Greatly expanded detail and update to describe actual implementation for PDK1.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group
NT OS/2 Coding Conventions

Author: Mark Lucovsky, Helen Custer

Revision 1.5, January 21, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Introduction 3
Module Headers 3
Function Headers 4
Header Files 6
  Header File Inclusion 6
  Header File Format 7
Naming 8
  Variable Names 9
    Initial Caps Format 9
    Unstructured Format 9
  Data Type Names 9
  Structure Field Names and Enumeration Constants 10
  Macro and Constant Names 10
Indentation and Placement of Braces 11
Constructs to Avoid 13
  Left Hand Side Typecasts 13
  Zero Length Arrays in Structures 13

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Introduction

All code written for NT OS/2 by members of the Portable Systems Group adheres to a common coding style. This style gives the system a uniform appearance that allows group members to read, modify, and maintain each other's modules without learning several different coding conventions.

The following items are standardized:

- Module headers
- Function headers and declarations
- Header file format
- Names of variables, data types, structure fields, macros, and constants
- Control structure indentation and placement of braces

Module Headers

The following prototype should appear at the beginning of each module. The source to the prototype can be found in file \nt\bak\inc\modhdr.c.

```
/*++

Copyright (c) 1989  Microsoft Corporation

Module Name:

    name-of-module-filename

Abstract:

    abstract-for-module

Author:

    name-of-author (email-name) creation-date-dd-mmm-yyyy

[Environment:]

    optional-environment-info (e.g. kernel mode only...)

[Notes:]

    optional-notes

Revision History:

    most-recent-revision-date email-name
        description
        .
        .
    least-recent-revision-date email-name
        description

--*/
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following is a sample of a completed module header:

```
/*++

Copyright (c) 1989  Microsoft Corporation

Module Name:

    pool.c

Abstract:

    This module contains the pool allocator for the NT OS/2
    executive.

Author:

    Mark Lucovsky (markl) 16-Feb-1989

Environment:

    Kernel mode only.

Revision History:

    22-Feb-1989     markl

        Modified module to conform to the new naming and coding
        standards agreed to 21-Feb-1989.

    20-Feb-1989     markl

        Added module and function headers.

--*/
```

Note that the revision history portion is not completed. Until we get further along in the project, we will not keep a revision history.

The /++ <text> --*/ construct is used by a comment extractor program that will be developed to assist in our documentation efforts.*

Function Headers

The following is a prototype function declaration. This declaration is to appear with the implementation of the function. The source to the prototype can be found in file \nt\bak\inc\prochdr.c.
Notice the following details in the function declaration:

- A form-feed character should appear one line before the "return-type" line. This convention is noted in this document with the string "<form-feed>".
- All formal arguments are preceded by one of the following macro definitions:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN Indicates that the argument is a non-modifiable input value (i.e., call-by-value semantics)

OUT Indicates that the argument is an address which refers to a variable or structure that will be modified by the function (i.e., call-by-reference semantics)

IN OUT Indicates that the argument is the address of an input variable or structure that is both read and written by the function (i.e., call-by-reference semantics)

- The OPTIONAL macro appears after a formal argument of type pointer, HANDLE, or ULONG when the function accepts either a NULL or non-NULL value. To determine whether the actual value supplied is NULL or non-NULL, the programmer must use the macro ARGUMENT_PRESENT, which takes the pointer, HANDLE, or ULONG variable as an argument and returns a value of type BOOLEAN.

- The order of the arguments in the comment block is the same as the order in which they appear in the function declaration.

- The function declaration follows:

```
<form-feed>
return-type
function-name(
    direction type-name argument-name,
    direction type-name argument-name...
    )

/*++

Routine Description:

    description-of-function

Arguments:

    argument-name - [Supplies | Returns]  description-of-argument
    .
    .
    .

Return Value:

    return-value - description-of-return-value
                                    -or-
                          None

 --*/

{
    .
    .
    .
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
}
```

The following is a sample of a completed function declaration:

```
<form-feed>
VOID
IoBuildPartialMdl(
    IN PMDL SourceMdl,
    IN PMDL TargetMdl,
    IN PVOID VirtualAddress,
    IN ULONG Length OPTIONAL
    )

/*++

Routine Description:

This routine maps a portion of a buffer as described by an
MDL.  The portion of the buffer to be mapped is specified via a virtual
address and an optional length.  If the length is not supplied, then
the remainder of the buffer is mapped.

Arguments:

    SourceMdl - MDL for the current buffer.

    TargetMdl - MDL to map the specified portion of the buffer.

    VirtualAddress - Base of the buffer to begin mapping.

    Length - Optional length of buffer to be mapped;  if zero,
               remainder.

Return value:

    None.
```

When a function is declared externally in a header file, its declaration contains only the function prototype and not the comment section. For example:

```
VOID
IoBuildPartialMdl(
    IN PMDL SourceMdl,
    IN PMDL TargetMdl,
    IN PVOID VirtualAddress,
    IN ULONG Length OPTIONAL
    );
```

Header Files

The following sections define the requirements for inclusion and format of header files.

Header File Inclusion

There are three types of header files in the NT OS/2 system:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Header files that are private to a single operating system component (the kernel or the I/O system, for example)
- A header file that is shared by the internal components of the operating system (the kernel and the executive)
- A public header file that defines external application programming interfaces (APIs) for system components outside the kernel and executive

Each component of the operating system has a private header file. The naming convention for these header files is <component-name>p.h. For example, the private header file for kernel component, ke, is called kep.h.

The NT OS/2 shared header file, \nt\private\src\ntos\inc\ntos.h, is included by each component of the executive and by the kernel, using the following statement:

```
#include "ntos.h"
```

(This file is included by a component's private include file.)

File ntos.h contains a list of #include statements, one for each operating system component. Each operating system component has a corresponding header file that defines prototypes for the functions that are shared with other components within the executive. The naming convention for these header files is <component-name>.h. For example, the header file containing shared prototypes for kernel component, ke, is called ke.h.

The public header file, \nt\sdk\inc\ntos2.h, is included by all components outside the NT OS/2 kernel and executive, using the following statement:

```
#include <ntos2.h>
```

Header File Format

Modules should be able to nest header files without causing multiple definition problems. To accomplish this, each header file should be conditionally expanded to itself, or to nothing if it has already been expanded.

In the example below, if the module pool.h was not previously included, then the macro _POOL_ is defined and the header file is expanded. Otherwise, _POOL_ is already defined and the remainder of the header file is ignored. This results in the header file being included only once.

The following header file style should be used:

```
/*++

Copyright (c) 1989  Microsoft Corporation

Module Name:

    pool.h
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Abstract:

    This module defines the NT OS/2 pool data structures and
    function prototypes.

Author:

    Mark Lucovsky (markl) 16-Feb-1989

Revision History:

--*/

#ifndef _POOL_
#define _POOL_

#include "ntdef.h"
#include "list.h"
#include "process.h"

typedef enum _POOL_TYPE {
    NonPagedPool,
    PagedPool
    } POOL_TYPE;

#endif // _POOL_
```

Note that if module list.h were shown, the conditional would appear as follows:

```
#ifndef _LIST_
#define _LIST_

//
// body
//

#endif // _LIST_
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Naming

The following sections describe the naming conventions for variables, structure fields, types, constants, and macros.

Variable Names

Variable names are either in "initial caps" format, or they are unstructured. The following two sections describe when each is appropriate.
Note that the NT OS/2 system does not use the Hungarian naming convention used in some of the other Microsoft products.

Initial Caps Format

All global variables and formal argument names must use the initial caps format. The following rules define this format:

- Words within a name are spelled out; abbreviations are discouraged.
- The first character of each word in a name is capitalized.
- Acronyms are treated as words, that is, only the first character of the acronym is capitalized.

The following list shows some sample names that conform to these rules:
NumberOfBytes
TcbAddress
BilledProcess

Unstructured Format

Local variables may appear in either the initial caps format, or in a format of the programmer's preference. The following list shows some possibilities for local variable names:
loopindex
LoopIndex
loop_index

Data Type Names

A set of primitive data types for use in the NT OS/2 system is defined in the file \nt\sdk\inc\ntdef.h. All NT OS/2 software must declare variables using these defined types rather than standard C types, where appropriate. The following are some examples of NT OS/2 types:

VOID
PVOID
QUAD
UQUAD
STRING
TIME

All new type names should be created in uppercase using typedef. Words within the name may either be packed together or separated by underscores. All types should have a corresponding typedef which

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

defines a pointer to the type. The name for the pointer is the type name with a "P" prefix.

The following example illustrates how to use typedef to create a structure type:

```
typedef struct _POOL_LIST_HEAD {
    ULONG CurrentFreeLength;
    ULONG TotalEverAllocated;
    LIST_ENTRY ListHead;
} POOL_LIST_HEAD, *PPOOL_LIST_HEAD;
```

The following example illustrates how to use typedef to create an enumerated type:

```
typedef enum _POOL_TYPE {
    NonPagedPool,
    PagedPool,
    MaxPoolType
    } POOL_TYPE;
```

Structure Field Names and Enumeration Constants

Structure field names should follow initial caps format. They should not have field name prefixes tied to a type. The following is a sample structure:

```
typedef struct _POOL_LIST_HEAD {
    ULONG CurrentFreeLength;
    ULONG TotalEverAllocated;
    LIST_ENTRY ListHead;
} POOL_LIST_HEAD, *PPOOL_LIST_HEAD;
```

As illustrated in the previous section, enumeration constants should also follow initial caps format.

Macro and Constant Names

All macros and manifest constants should have uppercase names. Words within a name may either be packed together, or separated by underscores.

The following statements illustrate some macro and manifest constant names:

```
#define PAGE_SIZE   4096
#define CONTAINING_RECORD(address, type, field) \
        ((type *)((LONG)(address) - \
                  (LONG)(&((type *)0)->field)))
```

Note: Any macro that is likely to be replaced by a function at a later time should use the naming conventions for functions.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Indentation and Placement of Braces

The following skeletal statements illustrate the proper indentation and placement of braces for C control structures. In all cases, indentations consist of four spaces each.
All control structures should routinely use braces even if there is only a single statement that will be executed.

```
<form-feed>
INT
FooBar(
    INT ArgumentOne,
    PULONG ArgumentTwo
    )

/*++

Routine Description:

    This is the routine description.

Arguments:

    ArgumentOne - Supplies the value for argument 1.

    ArgumentTwo - Supplies the address of argument 2.

Return Value:

    0 - Success

    1 - Failure

--*/

{
    //
    // Local variables are indented one tab (tabs are 4 spaces)
    //

    ULONG LocalVariable1;
    LONG Counter;

    //
    // for loops
    //    - all for loops must have braces
    //    - closing brace is at same indentation level as
    //      for statement
    //

    for ( Counter = 0; Counter < 10; Counter++ ) {

        //
        // Body of loop
        //

    }
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
//
// if statement
//
//    - All if statements should use braces
//

if ( Counter == 0 ) {

    //
    // Then statements
    //

}

//
// if then else
//

if ( Counter == 1 ) {

    //
    // Then statements
    //

} else {

    //
    // Else statements
    //

}

//
// switch statement
//

switch ( Counter ) {

case 1 :

    //
    // case 1 statements
    //
    break;

case 2 :

    //
    // case 2 statements
    //
    break;

default :

    //
    // default case
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        //
        break;

    }
}
```

Constructs to Avoid

NT OS/2 is written in portable, ANSI C. Due to differences in C compilers, there are a number of coding constructs that need to be avoided in order to promote portability.

Left Hand Side Typecasts

Some C compilers allow the cast operator on the left hand side of an assignment. This is not allowed by standard C and must be avoided in NT OS/2.

Zero Length Arrays in Structures

Zero length arrays embedded in structure definitions are not handled uniformly by all C compilers. They should not be used in NT OS/2.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Original Draft 1.0, February 21, 1989 - ml
Revision 1.1, February 23, 1989 - ml
Revision 1.2, May 5, 1989 - hkc

1. Extracted coding guidelines from exec.txt and converted text to Word.
2. Added text regarding primitive data type definitions.
3. Added text and example describing OPTIONAL arguments.
4. Added text regarding the inclusion of header files in implementation modules.
5. Style edit.

Revision 1.3, May 11, 1989 - Incorporated group comments. hkc
Revision 1.5, January 21, 1991 tonye

1. Emphasized that all control structures must use braces.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Linker/Librarian/Image Format Specification

Author: *Michael J. O'Leary*

Revision 1.3, May 31, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview 1
1.1 Design Goals 1
1.2 Constraints 1

2. Coff 1
2.1 What is Coff? 1
2.2 Why Coff? 2
2.3 Coff Structure 2
2.3.1 Coff File Layout 2
2.3.2 Coff File Header 4
2.3.3 Coff Optional Header 5
2.3.4 Coff Section Header 7
2.3.5 Coff Relocation Entry 11
2.3.6 Coff Linenumber Entry 11
2.3.7 Coff Symbol Table Entry 11
2.3.8 Coff Auxiliary Symbol Table Entry 14
2.3.8.1 Coff Symbol Table Ordering 14
2.3.9 Coff String Table 16
2.3.10 Overlays 16
2.3.11 Common Areas 16
2.3.12 16-bit Offset Definition 16

3. Fixups 16
3.1 Based Relocations 16
3.2 Relocation Types 17
3.2.1 I860 Relocation Types 17
3.2.2 386 Relocation Types 19
3.3 DLL Support 19
3.3.1 Thunks 20
3.3.2 Export Section 23

4. Image Activation 24

5. Resources 25

6. CodeView Support 25
6.1 Incremental Linking 25
6.2 Linker Command Line 26
6.3 Linker Switches 26

7. Librarian 26
7.1 Librarian Switches 27
7.2 Library File Layout 27
7.2.1 Library File Header 28
7.2.2 Library Member Header 28
7.2.3 Linker Member 29

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.2.4 Secondary Linker Member.. 29
7.2.5 Long Names Member.. 30

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This specification describes the Linker and Librarian for the NT OS/2 system. The Common Object File Format (COFF) standard with extensions needed to support Dynamic Linked Libraries (DLL's) and new languages such as C++ will be used both as the Object Module Format (OMF) produced by the compilers/assemblers and the executable image format used by the operating system to load a program.

1.1 Design Goals

- o Fastest possible image activation.
- o Minimize and localize pages that can't be shared and require fixups.
- o Able to base a DLL or image at a prefered memory location.
- o Linker is the only program that modifies or constructs images.
  - o Resource compiler will produce object fed to linker.
- o Need to easily support extensions to image format.
  - o Linker will support multiple sections in objects.

1.2 Constraints

- o Must be able to distinguish Cruiser Images vs NT images.
  - o Header must have common flags.
- o DLL support compatible with Cruiser.
  - o Support transfer of control (calls) and data references.
  - o All init routines called before program entry.
- o Must be compatible with Intel i860 assembler.
  - o Understand basic coff.
  - o Identify Intel extensions.

2. Coff

2.1 What is Coff?

Coff (Common Object File Format) is the formal definition for the structure of machine code files in the UNIX System V environment. All machine code files, whether fully linked executables, compiled applications, or system libraries, are COFF structured files. This will also become the formal definition for NT OS/2.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The COFF definition describes a complex data structure that represents object files, executable files, and archive (library) files. The Coff data structure defines fields for machine code, relocation information, symbolic information, and more. The contents of these fields are accessed by an organized system of pointers. Assemblers, compilers, linkers, and archivers manipulate the contents of the COFF data structure to achieve their particular objective.

2.2 Why Coff?

Coff was chosen over the Crusier Linear Executable Format because of the following reasons.

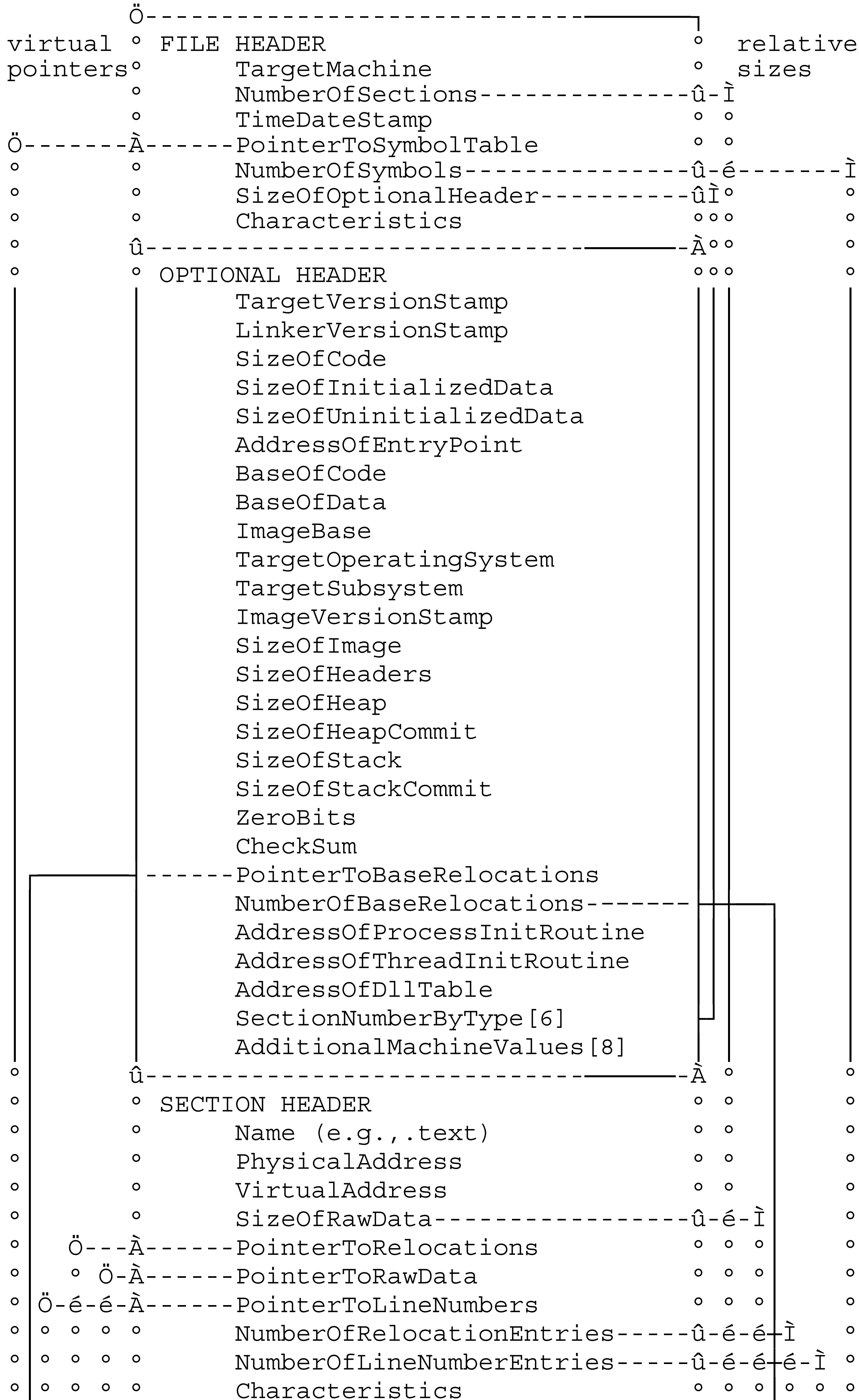
- Crusier images are not mappable.
  - No mappable image header.
  - Text and data pages are not laid out in the file such that they can be direclty mapped and paged into memory. Must grovel over a mapping table to determine page table contents.
  - Preloaded pages prohibit mapping.
  - Certain fields are not on their natural alignments.
  - Iterated data pages prohibit mapping.
- Crusier format contains 386 specifics.
- Wasted space for fields that will never be used.
  - Verify Record Table.
  - Resident Name Table.
  - Checksums.
- Fixups are by page/offset instead of by virtual address.
- Resource Compiler modifies executable image.
- Current i860 tools support COFF. We don't want to have to do another assembler.

2.3 Coff Structure

2.3.1 Coff File Layout

For NT OS/2, the following diagram shows the structure of a basic coff file. All headers must be at the beginning of the file. All other parts of the file can be in any order. An executable file will always be in the order show in this diagram.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License



Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
º│º º º û----------------------------------------À º º│º º º
º│º º º º other section header                    º º º│º º º
º│º º º û----------------------------------------À º º│º º º
º│º º º º last section header                     û-ì º│º º º
º│º º º û----------------------------------------À   º│º º º
º└┼──┼──┼──┤ base relocations                    │   º│º º º
º º º º º                                        └───┼┘º º º
º º º º û----------------------------------------À   º º º º
º º º Û-À raw data (.text)                       º   º º º º
º º º   º                                        û---ì º º º
º º º   û----------------------------------------À     º º º
º º º   º other sections raw data                º     º º º
º º º   û----------------------------------------À     º º º
º º Û---À first relocation entry                 º     º º º
º º     º      virtual address                   º     º º º
º º     º      symbol table index                º     º º º
º º     º      relocation type                   º     º º º
º º     û----------------------------------------À     º º º
º º     º last relocation entry                  û-----ì º º
º º     û----------------------------------------À       º º
º º     º other sections relocations             º       º º
º º     û----------------------------------------À       º º
º Û-----À first line number entry                º       º º
º       º      symbol table index                º       º º
º       º      line number                       º       º º
º       û----------------------------------------À       º º
º       º last line number entry                 û-------ì º
º       û----------------------------------------À         º
º       º other sections line numbers            º         º
º       û----------------------------------------À         º
Û-------À symbol table                           º         º
        º      name or string pointer            º         º
        º      virtual address                   º         º
        º      section number                    º         º
        º      type                              º         º
        º      class                             º         º
        º      number aux entries                º         º
        º                                        û---------ì
        û----------------------------------------À
        º [size] string table                    º SymPtr+NumSyms*SizeSym
        Û----------------------------------------ì
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The file header size and format is that of standard COFF.

```
typedef struct _FILE_HEADER {
    USHORT TargetMachine;
    USHORT NumberOfSections;
    ULONG  TimeDateStamp;
    ULONG  PointerToSymbolTable;
    ULONG  NumberOfSymbols;
    USHORT SizeOfOptionalHeader;
    USHORT Characteristics;
} FILE_HEADER, *PFILE_HEADER;
```

FILE_HEADER Structure:

TargetMachine —Indicates the target machine the object/image file is executable.

TargetEnvironment Flags:

COFF_FILE_TARGET_UNKNOWN —Indicates unknown target machine.

COFF_FILE_TARGET_860 —Indicates the object/image is binary compatable with the Intel i860 instruction set.

COFF_FILE_TARGET_386 —Indicates object/image is binary compatable with the Intel 386 instruction set.

COFF_FILE_TARGET_MIPS —Indicates object/image is binary compatable with the Mips instruction set.

NumberOfSections —Indicates the number of section headers contained in the file. The number of the first section is one.

TimeDateStamp —Indicates the time and date when the file was created. Number of elapsed seconds since 00:00:00 GMT, January 1, 1970.

PointerToSymbolTable —A file pointer (offset from the beginning of the file) to the start of the symbol table. The symbol table is sector aligned on disk.

NumberOfSymbols —Indicates the number of symbol table entries. Each entry is 18 bytes in length.

SizeOfOptionalHeader —Indicates the size of the optional header.

Characteristics —Indicates the characteristics of the object file.

Characteristics Flags:

COFF_FILE_RELOCS_STRIPPED —Relocation information stripped from file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

COFF_FILE_EXECUTABLE_IMAGE —No unresolved external references.

COFF_FILE_LINE_NUMS_STRIPPED —Line numbers stripped from file.

COFF_FILE_LOCAL_SYMS_STRIPPED —Local symbols stripped from file.

COFF_FILE_MINIMAL_OBJECT —Reserved.

COFF_FILE_UPDATE_OBJECT —Reserved.

COFF_FILE_BYTES_REVERSED —Bytes of machine word are reversed.

COFF_FILE_MACHINE_16BITS —16 bit word machine.

COFF_FILE_MACHINE_32BITS —32 bit word machine.

COFF_FILE_PATCH —Reserved.

COFF_FILE_NT_EXTENSIONS —If set, specifies the file contains new section headers and padded symbol table.

COFF_FILE_DLL —Image is a Dynamic Link Library.

COFF_FILE_BYTES_REVERSED_LO —Bytes of machine are reversed.

COFF_FILE_BYTES_REVERSED_HI —Bytes of machine are reversed. You can test either of the above two bits, they are in the same bit position in each short word. This allows you to identify if the coff object/image was written for a big or little endian machine.

2.3.3 Coff Optional Header

There is no standard COFF optional header size and format. NT defines the optional header as:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef struct _OPTIONAL_HEADER {
    USHORT TargetVersionStamp;
    USHORT LinkerVersionStamp;
    ULONG  SizeOfCode;
    ULONG  SizeOfInitializedData;
    ULONG  SizeOfUninitializedData;
    ULONG  AddressOfEntryPoint;
    ULONG  BaseOfCode;
    ULONG  BaseOfData;
    ULONG  ImageBase;
    USHORT TargetOperatingSystem;
    USHORT TargetSubsystem;
    ULONG  ImageVersionStamp;
    ULONG  SizeOfImage;
    ULONG  SizeOfHeaders;
    ULONG  SizeOfHeap;
    ULONG  SizeOfHeapCommit;
    ULONG  SizeOfStack;
    ULONG  SizeOfStackCommit;
    ULONG  ZeroBits;
    ULONG  CheckSum;
    ULONG  AddressOfBaseRelocations;
    ULONG  NumberOfBaseRelocations;
    PVOID  AddressOfProcessInitRoutines;
    PVOID  AddressOfThreadInitRoutines;
    ULONG  AddressOfDllTable;
    USHORT SectionNumberByTYpe[6];
    ULONG  AdditionalMachineValues[8];
} OPTIONAL_HEADER, *POPTIONAL_HEADER;
```

OPTIONAL_HEADER Structure:

TargetVersionStamp —Indicates operating system version.

LinkerVersionStamp —Indicates which version of the linker was used to build image.

SizeOfCode —Indicates the number of bytes of code.

SizeOfInitializedData —Indicates the number of bytes of initialized data.

SizeOfUnInitializedData —Indicates the number of bytes of uninitialized data.

AddressOfEntryPoint —Relative virtual address of starting point of image. This value added to the ImageBase is the virtual address of the entrypoint.

BaseOfCode —Indicates the relative virtual address (64K aligned) of the origin of the first byte of code. This value added to the ImageBase is the virtual address of the code.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BaseOfData —Indicates the relative virtual address (64K aligned) of the origin of the first byte of data. This value added to the ImageBase is the virtual address of the data.

ImageBase —Indicates the virtual address (64K aligned) of the origin of the file header.

TargetOperatingSystem —Indicates operating system and system version on which the image is executable.

TargetOperatingSystem Flags:

COFF_OPTIONAL_TARGET_OS_UNKNOWN —Indicates unknown target environment.

COFF_OPTIONAL_TARGET_OS_NTOS2 —Indicates image is targeted for NT OS/2.

TargetSubsystem —Indicates which subsystem of the operating system the image is intended to run under.

TargetSubsystem Flags:

COFF_OPTIONAL_TARGET_SUBSYSTEM_UNKNOWN —Indicates unknown subsystem.

COFF_OPTIONAL_TARGET_SUBSYSTEM_NATIVE —Indicates image runs under the native operating system. Subsystems are native images.

COFF_OPTIONAL_TARGET_SUBSYSTEM_OS2 —Indicates image is to run in the OS/2 subsystem.

COFF_OPTIONAL_TARGET_SUBSYSTEM_POSIX —Indicates image is to run in the Posix subsystem.

ImageVersionStamp —Indicates image version. To be used for backword compatibility. This stamp can be set by the user with the Version: switch.

SizeOfImage —Indicates the virtual size of the image.

SizeOfHeaders —Indicates the total size of all headers.

SizeOfHeap —Indicates the maximum size the heap is allowed to grow.

SizeOfHeapCommit —Indicates the initial heap size.

SizeOfStack —Indicates the maximum size the stack is allowed to grow.

SizeOfStackCommit —Indicates the initial stack size.

ZeroBits —Indicates how memory is to be allocated.

PointerToBaseRelocations —A file pointer to a table that is used to apply relocations to the image if the image can't be based at its desired base location. The first long word of the base table indicates the number of base table entries that follow. PointerToBaseTable will

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

be zero if the image doesn't have a base table. The base table structure is defined later in this document.

AddressOfProcessInitRoutines —TBD.

AddressOfThreadInitRoutines —TBD.

AddressOfDllTable —The relative virtual address of a table that defines DLL's. This is described later in this document.

SectionNumberByType —Is any array of interesting section numbers.

SectionNumberByType index values:

COFF_SECTION_TYPE_DEBUG —Indicates the section with contains the debug information.

COFF_SECTION_EXPORTS —Indicates the section with contains the export table.

COFF_SECTION_RESOURCE —Indicates the section with contains the resource data.

COFF_SECTION_SECURITY —Indicates the section with contains security information.

COFF_SECTION_EXCEPTION —Indicates the section with contains the exception tables.

The optional header is used only for images. If an object file contains an optional header of the proper size, it is used in the following manner:

If *TargetSubsystem* is not **COFF_OPTIONAL_TARGET_SUBSYSTEM_UNKNOWN**, then a subsystem is being defined. It tells the linker that the following sections within this file are for a particular subsystem. With this information, the linker can guarantee that different subsystem components won't be mixed together. Each library should contain one of these records.

If *AddressOfEntryPoint* is non-zero, then an entrypoint is being defined. This allows a compiler to supply the entrypoint without using the linker command line switch.

All other fields are ignored.

2.3.4 Coff Section Header

All section headers must follow the file header (or optional header if there is one).

An object or image can contain any number of sections and in any order. The linker combines any sections with the same name and with the same flags. For example, if a compiler wants to keep all constants together, then the compiler could use a section name of .const in every object that contained constants. The linker will merge these sections together (provided they also had the same flag attribute such as R/O). In some coff implementations, if a section is empty (i.e., object contains no .bss), a

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

section header still identifies the section, but would contain a zero size. For NT OS/2, this extra section header is not required.

Section names must start with a period (.). For each section, a special symbol will be defined by the linker. The period (.) will be replaced with a colon (:). This will be the next address after the section. Thus if a section is named **.text**, then the linker will create the symbol **:text**.

Grouping of sections hasn't been determined yet.

There are two styles of the section header. The first section header size and format is that of standard COFF. The second section header is an extension added to Coff. Both headers are the same size, but different format. The **COFF_OPTIONAL_NT_EXTENSIONS** flag in the file header specifies which section header the object contains. Section headers can not be mixed within one object, they must all be of one type. The image file will always have the **COFF_OPTIONAL_NT_EXTENSIONS** flag set, and thus the image will always contain new section headers.

The standard Coff section header has the following format:

```
typedef struct _OLD_SECTION_HEADER {
    UCHAR   Name[8];
    ULONG   PysicalAddress;
    ULONG   VirtualAddress;
    ULONG   SizeOfRawData;
    ULONG   PointerToRawData;
    ULONG   PointerToRelocations;
    ULONG   PointerToLinenumbers;
    USHORT  NumberOfRelocations;
    USHORT  NumberOfLineNumbers;
    ULONG   Characteristics;
} OLD_SECTION_HEADER, *POLD_SECTION_HEADER;
```

The new section header the following format:

```
typedef struct _NEW_SECTION_HEADER {
    UCHAR   Name[8];
    ULONG   NumberOfLinenumbers;
    ULONG   VirtualAddress;
    ULONG   SizeOfRawData;
    ULONG   PointerToRawData;
    ULONG   PointerToRelocations;
    ULONG   PointerToLinenumbers;
    ULONG   NumberOfRelocations;
    ULONG   Characteristics;
} NEW_SECTION_HEADER, *PNEW_SECTION_HEADER;
```

SECTION_HEADER Structure:

Name —Eight character null padded section name.

PysicalAddress —Indicates the physical address of the section. This field only exits within the old section header. Its value is never used.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VirtualAddress —Indicates the relative virtual address of the section.

SizeOfRawData —Indicates the size in bytes of the sections raw data.

PointerToRawData —A file pointer (offset from the beginning of the file) to the raw data for this sections.

PointerToRelocations —A file pointer (offset from the beginning of the file) to the relocation entries for this section. The relocation entries are sector aligned on disk.

PointerToLinenumbers —A file pointer (offset from the beginning of the file) to the line number entries for this section. The line number entries are sector aligned on disk.

NumberOfRelocations —Indicates the number of relocation entries for this section.

NumberOfLinenumbers —Indicates the number of line number entries for this section.

Characteristics —This flag represent three kinds of information:

- Section Type
- Section Content
- Section Memory Mapping

The flags determines how the linker and system loader handle the section. A section can only be of one type, one content, but can have a combination of memory flags set.

For now, all NT/OS2 objects and images will be of type **COFF_SCN_TYPE_REGULAR** except for those sections that want 16-bit offset addressing. These sections will be of type **COFF_SCN_TYPE_GROUPED**.

Section grouping is controlled by using a colon (:) in the section name. For example, if you have four objects each containing sections by the name of .DATA, .DATA:1, and .DATA:2, which all have the SAME FLAGS, then the linker will only create one section called .DATA which is a combination of all the sections but grouped in the following order:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Raw data for section .DATA
```

```
Object 1 DATA
Object 2 DATA
Object 3 DATA
Object 4 DATA
Object 1 DATA:1
Object 2 DATA:1
Object 3 DATA:1
Object 4 DATA:1
Object 1 DATA:2
Object 2 DATA:2
Object 3 DATA:2
Object 4 DATA:2
```

Further more, the linker performs grouping within a file. If a file contains multiple sections with the same group name, the linker will group all raw data with the same group name within the file together. A good example of this would be a library with many members each containing a .DATA:1 group. The linker will combine all .DATA:1 raw data extracted from the library together before it combines groups of the same name from other libraries.

If a sections name is 8 characters (without the colon), then the linker will not allow it to contain groups.

Characteristics Flags:

COFF_SCN_TYPE_REGULAR —.

COFF_SCN_TYPE_DUMMY —.

COFF_SCN_TYPE_NO_LOAD —.

COFF_SCN_TYPE_GROUPED —Used for 16-bit offset code.

COFF_SCN_TYPE_NO_PAD —Specifies if section should not be padded to next boundary before being combined with other like section.

COFF_SCN_TYPE_COPY —Reserved.

COFF_SCN_CNT_CODE —Section contains code.

COFF_SCN_CNT_INITIALIZED_DATA —Section contains initialized data.

COFF_SCN_CNT_UNINITIALIZED_DATA —Section contains uninitialized data.

COFF_SCN_CNT_OTHER —Reserved.

COFF_SCN_CNT_INFO —Section contains comments or some other type of information.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

A comment section can contain any type of information and can include relocations for this information. The first two long words of the raw data are reserved and are defined as InfoType and InfoVersion.

InfoType Flags:

COFF_SCN_INFO_UNKNOWN —Indicates unknown information.

COFF_SCN_INFO_DIRECTIVE —Indicates raw data contains linker directives such as entrypoint, full/partial/no debugging, etc. The compiler can set linker options by use of these directives. Usually the sections is also marked as discardable so this information doesn't become part of the image. InfoVersion is the linker version required to understand these directives. The current linker must have this version number or greater. The next long word is the number of directives being set, followed by the directives themselves (to be defined later). If the linker finds more than one directive of the same type (ie, two entrypoints) the linker will generated a warning and will use the first directive found.

COFF_SCN_INFO_COMPILER —Indicates raw data contains compiler information such as compiler type (i.e., C, Pascal, Fortran) and flags used. InfoVersion indicates the compiler version.

COFF_SCN_INFO_CODEVIEW —Indicates raw data contains CodeView information, and InfoVersion can either be the compiler or debugger version (to be determined later).

COFF_SCN_CNT_OVERLAY —Section contains an overlay.

COFF_SCN_CNT_DISCARD —Section contents will not become part of image. Directives to the linker will usually be marked discardable (ie, entrypoint defined by compiler).

COFF_SCN_MEM_NOT_CACHED —Section is not cachable.

COFF_SCN_MEM_NOT_PAGED —Section is not pageable.

COFF_SCN_MEM_SHARED —Section is shareable.

COFF_SCN_MEM_EXECUTE —Section is executable.

COFF_SCN_MEM_READ —Section is readable.

COFF_SCN_MEM_WRITE —Section is writeable.

2.3.5 Coff Relocation Entry

The relocation entries size and format is that of standard COFF.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef struct _RELOCATION_ENTRY {
    ULONG  VirtualAddress;
    ULONG  SymbolTableIndex;
    USHORT Type;
} RELOCATION_ENTRY, *PRELOCATION_ENTRY;
```

RELOCATION_ENTRY Structure:

VirtualAddress —Indicates the virtual address (position) in the section to be relocated.

SymbolTableIndex —Indicates the symbol table index (zero based) of the item that is referenced.

Type —Indicates the relocation type. Relocation types are defined later in this document.

2.3.6 Coff Linenumber Entry

The linenumber entries size and format is that of standard COFF.

```
typedef struct _LINENUMBER_ENTRY {
    union {
        ULONG  SymbolTableIndex;
        ULONG  VirtualAddress;
    }
    USHORT Linenumber;
} LINENUMBER_ENTRY, *PLINENUMBER_ENTRY;
```

LINENUMBER_ENTRY Structure:

SymbolTableIndex —If Linenumber is zero, indicates the symbol table index (zero based) of the function name.

VirtualAddress —If Linenumber is not zero, indicates virtual address of line number.

Linenumber —Indicates the line number relative to the start of the function.

2.3.7 Coff Symbol Table Entry

The symbol table entry size and format is that of standard COFF.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

typedef struct **_SYMBOL_TABLE_ENTRY** {
    UCHAR *Name*[8];
    ULONG *Value*;
    SHORT *SectionNumber*;
    USHORT *Type*;
    CHAR *StorageClass*;
    CHAR *NumberOfAuxiliaryEntries*;
} **SYMBOL_TABLE_ENTRY**, ***PSYMBOL_TABLE_ENTRY**;

SYMBOL_TABLE_ENTRY Structure:

Name —Symbol name. If the first four bytes are zero, then the last 4 bytes are a pointer to the symbol in the string table. The pointer technique is used if the symbol is longer than 8 bytes.

Value —Symbols value dependent on section number, storage class, and type.

SectionNumber —The section number the symbol is defined in.

SectionNumber meaning:

COFF_SYM_DEBUG —Indicates value represents special symbolic debug information.

COFF_SYM_ABSOLUTE —Indicates value is an absolute value.

COFF_SYM_UNDEFINED —Indicates that value is used as common.

COFF_SYM_DEFINED —Indicates that the symbol is defined.

Type —Symbolic type.

Type flags:

COFF_SYM_TYPE_NULL —Indicates no type.

COFF_SYM_TYPE_VOID —Indicates type void.

COFF_SYM_TYPE_CHAR —Indicates type character.

COFF_SYM_TYPE_SHORT —Indicates type short integer.

COFF_SYM_TYPE_INT —Indicates type integer.

COFF_SYM_TYPE_LONG —Indicates type long integer.

COFF_SYM_TYPE_FLOAT —Indicates type floating point.

COFF_SYM_TYPE_DOUBLE —Indicates type double word.

COFF_SYM_TYPE_STRUCT —Indicates type structure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

COFF_SYM_TYPE_UNION —Indicates type union.

COFF_SYM_TYPE_ENUM —Indicates type enumeration.

COFF_SYM_TYPE_MOE —Indicates type member of enumeration.

COFF_SYM_TYPE_UCHAR —Indicates type unsigned character.

COFF_SYM_TYPE_USHORT —Indicates type unsigned short integer.

COFF_SYM_TYPE_TYPE_UINT —Indicates type unsigned integer.

COFF_SYM_TYPE_ULONG —Indicates type unsigned long integer.

StorageClass —Storage class of the symbol.

StorageClass flags:

COFF_SYM_CLASS_EXTERNAL

COFF_SYM_CLASS_DLL_EXTERNAL

COFF_SYM_CLASS_AUTOMATIC

COFF_SYM_CLASS_REGISTER

COFF_SYM_CLASS_LABEL

COFF_SYM_CLASS_UNDEFINED_LABEL

COFF_SYM_CLASS_STATIC

COFF_SYM_CLASS_UNDEFINED_STATIC

COFF_SYM_CLASS_MEMBER_OF_STRUCT

COFF_SYM_CLASS_ARGUMENT

COFF_SYM_CLASS_STRUCT_TAG

COFF_SYM_CLASS_MEMBER_OF_UNION

COFF_SYM_CLASS_UNION_TAG

COFF_SYM_CLASS_TYPE_DEFINTION

COFF_SYM_CLASS_ENUM_TAG

COFF_SYM_CLASS_MEMBER_OF_ENUM

COFF_SYM_CLASS_REGISTER_PARAM

COFF_SYM_CLASS_BIT_FIELD

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

COFF_SYM_CLASS_BLOCK

COFF_SYM_CLASS_FUNCTION

COFF_SYM_CLASS_END_OF_STRUCT

COFF_SYM_CLASS_FILE

COFF_SYM_CLASS_SECTION

NumberOfAuxiliaryEntries —Number of auxiliary entries that further define this symbol.

2.3.8 Coff Auxiliary Symbol Table Entry

In general, auxiliary entries either implement a linked list structure within the symbol table that is used for efficient access of the symbol table data by both the linker and debugger, or contain debug/relocation information that is outside the scope of the symbol table entry structure. The following auxiliary entries are defined:

- Filename - This is the first auxiliary entry in the symbol table. The contents of the auxiliary entry is either the filename (if the name is 14 characters or less), or a pointer to the string table where larger filenames are placed. Filename may contain a path.
- Section Names - This auxiliary entry follows the symbol entry for a section name. It contains the section length, the number of relocation entries for the section, and the number of line number entries for the section. This information can also be found in the section header, but by placing the information in the auxiliary entry, the debugger can obtain all needed information directly from the symbol table.
- Tagname - To be defined.
- Function - To be defined. Will probably contain prototype information.
- Block - Include special entries such as .bb (begin block), .eb (end block), .bf (begin function), .ef (end function) and .eos (end of structure).
- Array

2.3.8.1 Coff Symbol Table Ordering

Because of symbolic debugging requirments, the order of symbols in the symbol table is very important. Symbols appear in the following sequence:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

.file filename1
.define function1
.define local var1 for function1
...
.define local varN for function1
.begin function
.block begin
...
.end block
.end function
statics
...
.file filename2
.define function1
.define local var1 for function2
... ...
statics
...
defined global symbols
undefined global symbols

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.3.9 Coff String Table

The string table is the final component of the symbolic information. If in a symbol entry, the first four characters of the symbol's name are NULL, then the last four characters represent an offset (relative to the start of the string table) into the string table where the symbol's name is stored. Symbol names are NULL-terminated, thus the symbol's name can be any length.

The first four bytes in the string table represent a long value that specifies the number of bytes in the string table. An empty string table has a length field, but the value stored there is 0.

Internal symbols generated by compilers should try to be 8 characters or less, for these are the most efficent and require the less space.

2.3.10 Overlays

- o To be defined

2.3.11 Common Areas

Common areas are defined by the symbol record containing a non-zero value, and a zero (undefined) section number. In this case, the value is the size (number of bytes) of the symbol. The linker merges symbols of the same name and allocates the largest required space in a section called .common with content of **COFF_SCN_TYPE_UNINITIALIZED_DATA**.

2.3.12 16-bit Offset Definition

When sections have the SECTION_TYPE_GROUP flag set, the linker combines sections with the same name but different content flags into one section. The combined section must be 64K or less, otherwise the linker will generate an error. A special symbol will be defined by the linker that will be the address of the middle of the section, thus signed 16-bit displacements can be used by compilers. The special symbol defined by the linker will be that of the section name but the '.' (period) will be replaced with a ';' (semi-colon).

It hasn't been determined how grouping of sections with different memory flags occur. In the worst case, they must be all of one kind, probably R/W.

3. Fixups

- o Fixups will be performed in user mode. Thus, no code is required to verify fixups are valid (in the event an image has been tampered with).
- o If the image is mapped at its specified based address, then the only runtime fixups required are those for DLL's. If the image is not mapped at the specified base address, then the fixups have to be re-applied.
- o The linker will generate thunks for calls to DLL's, thus the fixups are to read/write data, not to code. Thus no Icache flushes are necessary.
- o The linker will have a switch to indicate if fixups should occur as they are needed, or for a whole DLL at a time.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.1 Based Relocations

The base relocations are used to re-apply fixups when an image's based address is unavailable at load time. The structure of a based entry follows:

typedef struct **_BASED_RELOCATION_ENTRY** {
    ULONG *VirtualAddress*;
    ULONG *Value*;
    USHORT *Type*;
} **BASED_RELOCATION_ENTRY**, ***PBASED_RELOCATION_ENTRY**;

BASED_RELOCATION_ENTRY Structure:

VirtualAddress —Indicates the virtual address (position) in the image to be relocated.

Value —Indicates the value of the item that is referenced. This value plus the new base should replace the word located at the virtual address.

Type —Indicates the relocation type. Relocation types are defined later in this document.

3.2 Relocation Types

3.2.1 I860 Relocation Types

- COFF_REL_I860_ABSOLUTE

  This relocation is ignored.

- COFF_REL_I860_DIR32

  *(long *)Location += Addr

- COFF_REL_I860_PAIR

  Defines PairAddr.

- COFF_REL_I860_HIGH

  *(short *)Location = ((Addr + PairAddr) >> 16)

- COFF_REL_I860_LOW0

  *(short *)Location += (short)Addr

- COFF_REL_I860_LOW1

  *(short *)Location += (short)Aligned(Addr, 2)

- COFF_REL_I860_LOW2

  *(short *)Location += (short)Aligned(Addr, 4)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o COFF_REL_I860_LOW3

```
*(short *)Location += (short)Aligned(Addr, 8)
```

- o COFF_REL_I860_LOW4

```
*(short *)Location += (short)Aligned(Addr, 16)
```

- o COFF_REL_I860_SPLIT0

```
T1 = *(long *)Location
T2 = (((T1 >> 5) & 0xf800) | (T1 & 0x7ff)) + Addr
T2 = ((T2 << 5) & 0x1f0000) | (T2 & 0x7ff)
*(long *)Location = T2 | (T1 & (~0x1f07ff))
```

- o COFF_REL_I860_SPLIT1

```
T1 = *(long *)Location
T2 = (((T1 >> 5) & 0xf800) | (T1 & 0x7fe)) + Aligned(Addr,2)
T2 = ((T2 << 5) & 0x1f0000) | (T2 & 0x7fe)
*(long *)Location = T2 | (T1 & (~0x1f07fe))
```

- o COFF_REL_I860_SPLIT2

```
T1  = *(long *)Location
T2 = (((T1 >> 5) & 0xf800) | (T1 & 0x7fc)) + Aligned(Addr, 4)
T2 = ((T2 << 5) & 0x1f0000) | (T2 & 0x7fc)
*(long *)Location = T2 | (T1 & (~0x1f07fc))
```

- o COFF_REL_I860_HIGHADJ

```
*(short *)Location = ((Addr + rel1.r_symndx) >> 16)
if ((Addr + rel1.r_symndx) & 0x8000)
    *(short *)Location += 1
```

- o COFF_REL_I860_BRADDR

```
Addr = Addr - ((VirtAddr - PhysAddr) + 4 + VirtAddr
if ((Addr >= 0x4000000L) || (Addr < -0x4000000L))
    " Too Far "
```

I'll explain the previous relocation types by sample i860 code.

```
orh   h%foo,r0,r31          // COFF_REL_I860_HIGH
or    l%foo,r31,r31         // COFF_REL_I860_LOW0
ld.l  0(r31),r16
```

The first 2 instructions moves the address of the memory location labeled foo into r31. The COFF_REL_I860_HIGH type instructs the linker to extract the upper 16 bit of the address of foo for use as immediate operand in the orh instruction. Similarly, the COFF_REL_I860_LOW0 type instructs

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

the linker to extract the lower 16 bit of the address of foo for use as immediate operand in the or instruction. The final ld.l loads the memory location referenced by r31 into r16.

Alternatively, you can use

```
orh     ha%foo,r0,r3    // COFF_REL_I860_HIGHADJ, PAIR
ld.l    l%foo(r31),r16  // COFF_REL_I860_LOW0
```

to load foo into r16. The COFF_REL_I860_HIGHADJ type behaves like the COFF_REL_I860_HIGH type except that it adds 1 to the extracted upper 16 bit if bit 15 of the address value is set. This adjustment is needed because load/store arithmetic instructions sign-extend the 16-bit immediate operand. If you used

```
orh     h%foo,r0,r31    // COFF_REL_I860_HIGH
ld.l    l%foo(r31),r16  // COFF_REL_I860_LOW0
```

you will load from the wrong address when bit 15 of foo is set. Immediate operands are 0-extended in logical instructions.

```
orh     ha%foo,r0,r31   // COFF_REL_I860_HIGHADJ,  PAIR
ld.b    l%foo(r31),r16  // COFF_REL_I860_LOW0
ld.s    l%foo(r31),r16  // COFF_REL_I860_LOW1
ld.l    l%foo(r31),r16  // COFF_REL_I860_LOW2

orh     ha%foof,r0,r31  // COFF_REL_I860_HIGHADJ, PAIR
fld.l   l%foof(r31),f16// COFF_REL_I860_LOW2
fld.d   l%foof(r31),f16// COFF_REL_I860_LOW3
fld.q   l%foof(r31),f16// COFF_REL_I860_LOW4
```

The variaous COFF_REL_I860_LOW types are used to extract the lower 16 bits of a constant or and address label. The linker verifies alignment of the immediate offsets (Intel i860 Programmer Reference Manual section 5.2 programming notes) because the lower bits of the immediate are used to encode the operand length. See appendix B of the Intel i860 Programmers Reference Manual for the instruction format.

COFF_REL_I860_LOW1 verifies alignment of the immediate to 2 byte boundary.
COFF_REL_I860_LOW2 verifies alignment of the immediate to 4 byte boundary.
COFF_REL_I860_LOW3 verifies alignment of the immediate to 8 byte boundary.
COFF_REL_I860_LOW4 verifies alignment of the immediate to 16 byte boundary.

```
orh     ha%foo,r0,r31   // COFF_REL_I860_HIGHADJ, PAIR
st.b    r16,l%foo(r31)  // COFF_REL_I860_SPLIT0
st.s    r16,l%foo(r31)  // COFF_REL_I860_SPLIT1
st.l    r16,l%foo(r31)  // COFF_REL_I860_SPLIT2
```

The COFF_REL_I860_SPLIT types are used by the st instruction (fst uses the COFF_REL_I860_LOW fixups). They verify the alignment of the immediate as well as split the immediate over bit 20..16 and 10..0 of the instruction. The alignment is needed because bit 0 and bit 28 are used to encode operand length.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

COFF_REL_I860_SPLIT1 verifies alignment of immediate to 2 byte boundary.
COFF_REL_I860_SPLIT2 verifies alignment of immediate to 4 byte boundary.

```
     br         foo
     nop
foo: nop                        // COFF_REL_I860_BRADDR
```

The COFF_REL_I860_BRADDR type is used to fixup a br to an address label. The linker computes the offset of the target label relative to the current PC + 4.

3.2.2 386 Relocation Types

- COFF_REL_I386_ABSOLUTE
- COFF_REL_I386_DIR16
- COFF_REL_I386_REL16
- COFF_REL_I386_DIR32
- COFF_REL_I386_REL32

3.3 DLL Support

- An executable image which is a DLL will:
  - Have an export section which contains the ordinals, function names, and function address of each exported routine.
  - May contain init code if AddressOfEntryPoint != 0.
- An executable image which uses a DLL will
  - Have a Dll Descriptor table for each DLL used. These tables will be grouped together and the optional header will contain the address of the first table.
  - Thunks for the DLL that will be snaped at load time.

3.3.1 Thunks

The best way to describe thunks is show an example. The following example is i860 code.

Suppose we had the following Definition file:

```
GetVersion=DosCalls.GetVersion
GetMachineMode=DosCalls.GetMachineMode
GetMode=VioCalls.GetMode
Foo=DosCalls.128
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

and the following user code:

```
call GetVersion
call GetMode
call GetMachineMode
call Foo
```

The image would end up contain the following code:

```
call thunk1
call thunk2
call thunk3
call thunk4

thunk1:
    br DosCallsThunkRoutine
    ld.c fir,r31
    .word relative address of GetVersionThunkData


thunk2:
    br VioCallsThunkRoutine
    ld.c fir,r31
    .word relative address of GetModeThunkData


thunk3:
    br DosCallsThunkRoutine
    ld.c fir,r31
    .word relative address of GetMachineModeThunkData


thunk4:
    br DosCallsThunkRoutine
    ld.c fir r31
    .word relative address of Ordinal128ThunkData


DosCallsThunkRoutine:
    ld.l 0(r31),r30
    add r30,r31,r30
    ld.l 0(r30), r29
    bri r29
    nop

VioCallsThunkRoutine:
    ld.l 4(r31),r30
    add r30,r31,r30
    ld.l 0(r30), r29
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
bri r29
nop
```

Notice that DosCallsThunkRoutine and VioCallsThunkRoutine are identical. The reason for this is purely for debugging reasons. With different thunk routines, the user can set a breakpoint at the thunk routine for a specific DLL or a profiler could show which functions within which DLL are being called. The ideal situation would be to only generate one thunk routine if debugging isn't enabled, otherwise generate a thunk routine per DLL. However, I haven't figured out a way to do this yet, so until then, each DLL will have its own thunk routine.

Thunk data has the following format:

typedef struct **_THUNK_DATA** {
    PTHUNK_BY_NAME *Function;*
} **THUNK_DATA**, ***PTHUNK_DATA**;

THUNK_DATA Structure:

Function —Specifies either an ordinal number or a pointer to THUNK_BY_NAME structure. If it is an ordinal number, it will have a value less than 64K.

typedef struct **_THUNK_BY_NAME** {
    ULONG *Hint;*
    UCHAR *Name[1];*
} **THUNK_BY_NAME**, ***PTHUNK_BY_NAME**;

THUNK_BY_NAME Structure:

Hint —A hint value that can be used to reference into the ExportNames in the EXPORT_SECTION_DATA.

Name —The functions name.

Thus by example, we have:

```
DosCallsThunkData:

GetVersionThunkData:
      .word pointer to hint & "GetVersion"


GetMachineModeThunkData
      .word pointer to hint & "GetMachineMode"


Ordinal128ThunkData:
      .word 128
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
VioCallsThunkData:

GetModeThunkData:
      .word pointer to hint & "GetMode"
```

The DLL descriptor is defined as:

typedef struct **_DLL_DESCRIPTOR** {
    ULONG *Characteristics;*
    PUCHAR *Name;*
    PVOID *FirstThunk;*
} **DLL_DESCRIPTOR**, ***PDLL_DESCRIPTOR**;

DLL_DESCRIPTOR Structure:

Characteristics —TBD.

Name —A pointer to the name of the DLL.

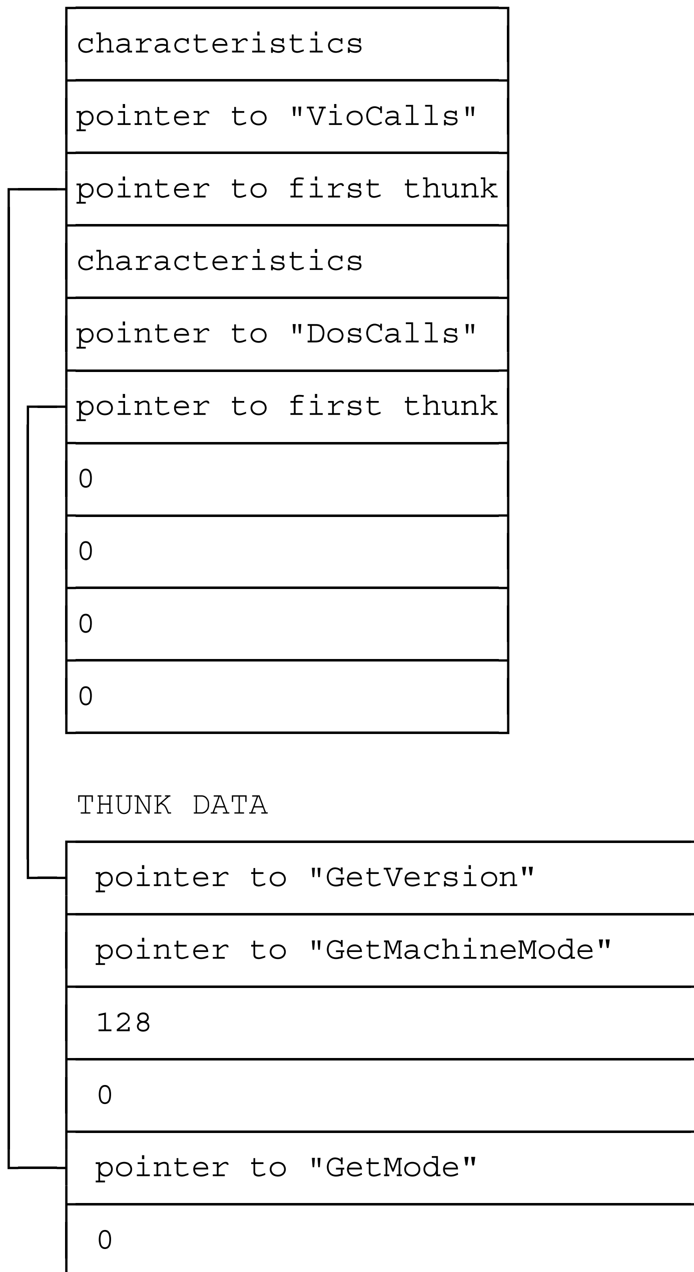
FirstThunk —A pointer to the first thunk for this DLL.

The linker places all DLL descriptors contigously in the image file. An empty DLL descriptor (both fields are zero) is appended to the list of DLL descriptors.The PointerToDLLTable in the optional headers points to the first DLL descriptor.

The purpose of the DLL descriptor is that once a snap occurs, it is possible to snap all thunks for the DLL at once. The linker places all the thunks for a particular DLL contiguously. It also appends an additional thunk data record to the list. This record will have both function_ordinal and function_name set to zero. This signifies the end of the DLL thunk data.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Thus, by example we have:



The linker doesn't know if a function is within a DLL and it doesn't have to. The thunk and thunk data will be extracted from a library that was created by the librarian from a definition file.

3.3.2 Export Section

The export section will be the first section header of an image that is flaged as a DLL. The raw data of the section has the following format:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef struct _EXPORT_SECTION_DATA {
    ULONG Characteristics;
    PSZ   DllName;
    ULONG VersionStamp;
    ULONG Base;
    ULONG NumberOfOrdinals;
    ULONG NumberOfNames;
    PVOID *AddressOfOrdinalFunction;
    PEXPORT_NAME_TABLE ExportNames;
} EXPORT_SECTION_DATA, *PEXPORT_SECTION_DATA;
```

EXPORT_SECTION_DATA Structure:

Characteristics —TBD.

DllName —A pointer to the name of the DLL.

VersionStamp —TBD.

Base —TBD.

NumberOfOrdinals —Indicates the number of ordinal functions.

NumberOfNames —Indicates the number of named functions.

AddressOfOrdinalFunction —A virtual address of the ordinal function.

ExportNames —A pointer to the function exported by name.

```
typedef struct _EXPORT_NAME_TABLE {
    PSZ ExportedName;
    ULONG  Ordinal;
} EXPORT_NAME_TABLE, *PEXPORT_NAME_TABLE;
```

EXPORT_NAME_TABLE Structure:

DllName —A pointer to the name of the Function.

Ordinal —The ordinal assigned to the function.

4. Image Activation

- Image headers, section headers, inport/export lists and debug information must all be mappable.
- Based images and DLL's.
- Sections are aligned on sector boundaries and are mapped on 64K virtual addresses.
- The only kernel memory resident structures is the information to resolve a virtual page to a disk location.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Psedo code for activating FOO.EXE.

```
Activate("FOO.EXE");


Activate (Image_Name)
{
  Handle = CreateSection(Image_Name, ..., ...);
  Image_Base = MapView(Handle, ..., ...);
  if (Image_Base->Optional_Header.PreferredImageBase != Image_Base){
     perform_local_fixups();
  }
  Load_DLL(Image_Base);
}


Load_DLL (Image_Base)
{
  If (ImageBase->Section_Header[0].Name == '.export') {
     while (Fetch_Next_DLL_Name() != NULL) {
        DLL_Handle = CreateSection(DLL_Name, ..., ...);
        DLL_Base = MapView(DLL_Handle, ..., ...);
        if (DLL_Base->Optional_Header.PreferredImageBase != DLL_Base) {
           perform_local_fixups();
        }
        Load_DLL(DLL_Base);
        if (Image_Base->Optional_Header.EntryPoint) {
          call (Image_Base->Optional_Header.EntryPoint());
        }
        perform_DLL_fixups();
}
```

5. Resources

Resources are used for internationalzation. For example, if all the error messages of an image are in a resource, then the object containing the resource can be replaced with a new resource object that contains the error messages in another language.

- Bitmaps, Fonts, Icons and Strings can all be resources.
- The resource compiler will not modify the executable images as is done today in OS/2. Instead, the resource compiler will produce either assembler or c language code that can be compiled and then linked with the retain flag set so it can be incrementally linked later.
- Resouces will be combined into one section.
    - The resource section will have a reserved name. Currently this name is .resrc.
    - The section flag will be marked as **COFF_SCN_CNT_INITIALIZED_DATA**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The current OS/2 implib program will be incorporated into the linker. It will read a definition file and produce a library which contains the thunk code for DLL entry points.

6. CodeView Support

CodeView information will reside in a section with content being **COFF_SCN_TYPE_INFO**. The Linker does not know about the internal structure of the CodeView information. The section can contain relocation entries for the information.

- How duplicate debug information might be discarded hasn't been decided yet.

6.1 Incremental Linking

Incremental linking is used to replace specific parts of an image file. This is how you change resources.

The linker will be able to incrementally link objects provided the retain switch was used before incremental linking is desired. The linker will retain the needed relocation entries for each object that refers to a specific section. The linker replaces the old section with the new section and re-applys the fixups. NOTE: If the size of the section grows, and can't fit in the padded space left from sector aligning, it hasn't been decided if the linker will move everything or just return an error indicating full linking must occur.

Incremental linking is accomplished by linking an executable image with 1 or more objects.

6.2 Linker Command Line

The linker can except switches, objects, libraries, and the definition file in any order on the command line. Only one definition file can be specified. The linker processes the object files (in order) before processing the libraries.

6.3 Linker Switches

- Debug:[None,Full,Partial]
- Def:*Filename*
- Dll
- Map:[*Filename*]
- Base:*Address* (64K aligned)
- Entry:*SymbolName*
- Force
- Include:*SymbolName*
- Out:*Filename*
- Stack:*Size*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Version:*Number*
- Retain=[All,*ObjectName*]:[All,*SectionName*]
- Fixup=*DllLibraryName*:[All,1by1,None]

7. Librarian

- The librarian will be imbedded into the linker or will at least share a DLL.
- Multiple objects of the same name will NOT be allowed in the same library.
- Multiple symbols of the same name will NOT be allowed in the same library.

The librarian has two functions:

The first function of the librarian is to simply merge files together. When the librarian builds a library, a member header is created for each file that is a member of the library. This allows removal of each individual file from a library. Any file can become a member of a library. When the files being added to a library are not COFF objects, the librarian acts like a simple file merger. You can merge an unlimited number of files together into one large file. This file will not be considered a valid library for linking purposes. A valid library to be used by the linker is created by merging only COFF objects together.

When a library contains only COFF objects, the librarian performs its second function, which is to build a symbol table for all defined external functions within the library. This symbol table is called the linker member, because it allows the linker to perform fast lookup on defined functions within the library. Once the linker member is created, only COFF objects can be added to the library.

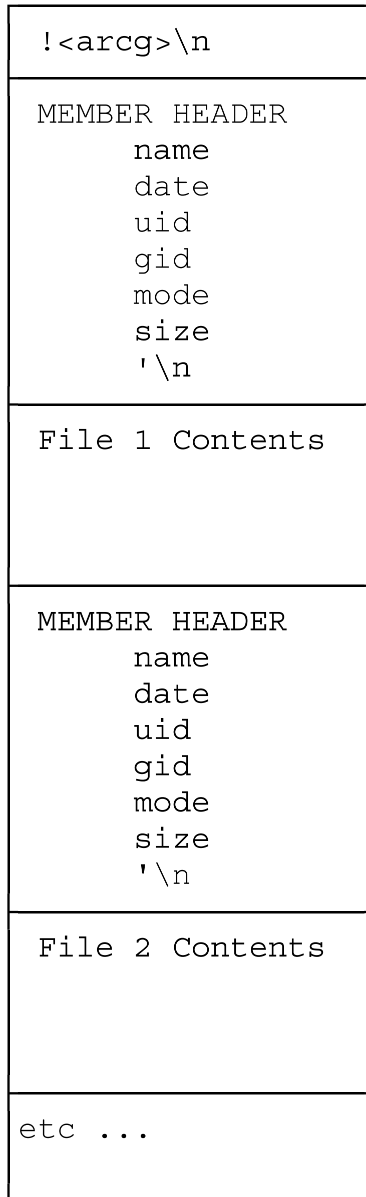
7.1 Librarian Switches

- Remove:Membername
- Def:Filename
- List

Membername is the name of the file. The linker member name is backslash (/).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.2 Library File Layout



7.2.1 Library File Header

A library file always starts with the 8 characters **!<arch>\n** where \n is a newline character.

7.2.2 Library Member Header

The library member header size and format is that of standard COFF archive files.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef struct _MEMBER_HEADER {
    CHAR Name[16];
    CHAR Date[12];
    CHAR UserID[6];
    CHAR GroupID[6];
    CHAR Mode[8];
    CHAR Size[10];
    CHAR EndHeader[2];
} MEMBER_HEADER, *PMEMBER_HEADER;
```

<u>MEMBER_HEADER Structure:</u>

Name —Is the file name of the member. It is terminated with a backslash (/) character, followed by spaces if needed to fill out the rest of the character array. The member name is stored this way only if the file name is less than 16 characters long. If the file name is 16 characters or more (path name is included), the the member names begins with a backslash (/) character, followed by ascii digits which are used as an offset into the long name table (described below).

Date —Is the members creation data as an ASCII string of decimal characters.

UserID —To be defined.

GroupID —To be defined.

Mode —To be defined.

Size —Defines the size of the member in bytes. The size can be used to find the header of the next member.

EndHeader —Contains the string **`\n** (grave accent followed by a newline character).

NOTE: A member header always starts on an even-byte boundary. A newline character (\n) is used for filling if the members contents ends on an odd-byte boundary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.2.3 Linker Member

If the file contains a COFF object, then a linker member is built by the librarian, and is the first member of the archive file. The linker member is sorted by member header offsets. The linker member is standard coff and is constructed in the following manner:

```
+------------------------------+
| MEMBER HEADER                |
|      name "/         "       |
|      date                    |
|      uid                     |
|      gid                     |
|      mode                    |
|      size                    |
|      '\n                     |
+------------------------------+
| number of symbols            |
+------------------------------+
| member header offset         |
| for symbol name1             |
+------------------------------+
| member header offset         |
| for symbol name2             |
+------------------------------+
| .                            |
| .                            |
| .                            |
+------------------------------+
| member header offset         |
| for symbol nameN             |
+------------------------------+
| symbol name1                 |
| symbol name2                 |
|                              |
|                              |
| symbol nameN                 |
+------------------------------+
```

7.2.4 Secondary Linker Member

A second linker member is built by the NT librarian. This is not standard, but most existing tools should ignore the second linker member. The second linker member is sorted by symbols names. The second linker member is constructed in the following manner:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
+-----------------------------+
| MEMBER HEADER               |
|      name "/        "       |
|      date                   |
|      uid                    |
|      gid                    |
|      mode                   |
|      size                   |
|      '\n                    |
+-----------------------------+
| number of offsets           |
+-----------------------------+
| member header 1 offset      |
+-----------------------------+
| member header 2 offset      |
+-----------------------------+
| .                           |
| .                           |
| .                           |
+-----------------------------+
| number of symbols           |
+-----------------------------+
| member offset index         |
| for symbol name1            |
+-----------------------------+
| member offset index         |
| for symbol name2            |
+-----------------------------+
| .                           |
| .                           |
| .                           |
+-----------------------------+
| member offset index         |
| for symbol nameN            |
+-----------------------------+
| symbol name1                |
| symbol name2                |
|                             |
|                             |
| symbol nameN                |
+-----------------------------+
| full member 1 filename      |
| full member 2 filename      |
|                             |
| full member N filename      |
+-----------------------------+
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.2.5 Long Names Member

The NT linker builds a long name table if any of the file names being added to the library are longer than 15 characters. This is not standard COFF, but is part of the new System V ABI. The long name table is constructed in the following manner:

```
MEMBER HEADER
     name "//       "
     date
     uid
     gid
     mode
     size
     '\n
-------------------------
asciiz strings
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Original Draft 1.0, November 06, 1989

Revision 1.1, January 10, 1990

Revision 1.2, Febuary 26, 1990

Revision 1.3, May 31, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Debug Architecture

Author: *Mark Lucovsky*

Revision 1.1, May 8, 1990
Original Draft February 15, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview.. 1
1.1 Debug Event Flow 1

2. Debug Architecture Partitioning.. 2
2.1 Debug Event Generation.. 3
2.1.1 Event Generation Message Formats.. 3
2.1.1.1 Exception.. 4
2.1.1.2 CreateThread 4
2.1.1.3 CreateProcess 5
2.1.1.4 ExitThread.. 5
2.1.1.5 ExitProcess 5
2.1.1.6 MapSection... 6
2.1.1.7 UnMapSection.. 6
2.2 Event Propagation.. 6
2.2.1 Emulation Subsystem APIs for Event Propagation 7
2.2.1.1 DbgSsInitialize 7
2.2.1.2 UiLookupRoutine... 8
2.2.1.3 SubsystemKeyLookupRoutine.. 8
2.2.1.4 KmApiMsgFilter 9
2.2.1.5 DbgSsHandleKmApiMsg.. 10
2.2.2 Event Propagation Message Formats 10
2.2.2.1 Exception.. 11
2.2.2.2 CreateThread 11
2.2.2.3 CreateProcess 11
2.2.2.4 ExitThread.. 11
2.2.2.5 ExitProcess 12
2.2.2.6 MapSection... 12
2.2.2.7 UnMapSection.. 12
2.3 Coordinate Debugger and Debuggee 12
2.3.1 Dbg Server Data Structures... 12
2.3.1.1 Subsystem Structure... 12
2.3.1.2 User Interface Structure 13
2.3.1.3 Application Process Structure 14
2.3.1.4 Application Thread Structure 15
2.3.2 Dbg Server Responses to Debug Event Propagation 16
2.3.2.1 Exception.. 16
2.3.2.2 CreateThread 16
2.3.2.3 CreateProcess 17
2.3.2.4 ExitThread.. 17
2.3.2.5 ExitProcess 18
2.3.2.6 MapSection... 18
2.3.2.7 UnMapSection.. 18
2.4 User Interface Interactions with the Dbg Server... 18
2.4.1 DbgUiConnectToDbg 19

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.4.2 DbgUiWaitStateChange 19
  2.4.2.1 State Change Record 20
  2.4.2.1.1 DbgCreateThreadStateChange 20
  2.4.2.1.2 DbgCreateProcessStateChange 21
  2.4.2.1.3 DbgExitThreadStateChange 22
  2.4.2.1.4 DbgExitProcessStateChange 22
  2.4.2.1.5 DbgExceptionStateChange 22
  2.4.2.1.6 DbgBreakpointStateChange 22
  2.4.2.1.7 DbgSingleStepStateChange 22
  2.4.2.1.8 DbgMapSectionStateChange 23
  2.4.2.1.9 DbgUnMapSectionStateChange 23
2.4.3 DbgUiContinue 23
  2.4.3.1 DbgExitThreadStateChange 24
  2.4.3.2 DbgExitProcessStateChange 24

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This specification describes the Debug Architecture found in NT OS/2. The Debug Architecture consists of the following:

- o Dbgk executive component. This component is responsible for generating debug events and sending a message through a processes debug port when a debug event for the process occurs.

- o The debugger user interface (DebugUi) is an application that provides the human interface for a debugger. An instance of DebugUi exists for each application being debugged.

- o Dbg user-mode subsystem (Dbg server). This component acts as a debug event coordinator, ensuring that debug events occuring in a process are made available to its controlling DebugUi.

- o DbgSs APIs. This set of APIs allows an Emulation Subsystem to participate in the NT OS/2 Debug Architecture. While not required, It is expected that Emulation Subsystems pick up debug events coming from the Dbgk component, add information to these events, and then forward the event to the Dbg subsystem.

- o DbgUi APIs. This set of APIs allow a DebugUi to communicate with the Dbg server so that it may receive notification of outstanding debug events, and respond to received debug events.

1.1 Debug Event Flow

Before going any further, the following example of a debug event illustrates the typical interaction between the components listed above.

- o The OS/2 subsystem (OS2SS) is controlling an application process. The process was started as a debugged process through the **SbCreateForeignSession** API. Upon reciept of the process by OS2SS, a debug port was assigned to the process using **NtSetInformationProcess**. OS2SS owns the processes debug port and receives all messages arriving at this port. The application process contians a single thread. At some point in the thread's lifetime, a reference to inaccessible memory is made. This generates an access violation exception triggering a potential debug event.

- o The exception dispatcher in the NT OS/2 executive calls into the Dbgk component (at its **DbgkForwardException** entrypoint) to report the access violation. Dbgk determines whether or not the process has an associated DebugPort. In this case, the process does have a DebugPort, so a DBGKM_EXCEPTION message is formatted. All threads (except for the current thread) in the process are frozen using **KeFreezeThread**. The current thread sends the exception message through its DebugPort and awaits a reply.

- o OS2SS receives the exception message. Since the message type stored in the message header is LPC_DEBUG_EVENT, OS2SS calls **DbgSsHandleKmApiMsg** passing the address of the message. **Exception** messages requires no additional information from the OS2SS, so the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

message is forwarded to the Dbg server for processing. The message is sent as a datagram so that OS2SS does not have to burn a thread while waiting for a reply.

- o The Dbg server receives the exception message. Using the Client Id from the the original exception message, Dbg locates an internal per-thread data structure. The exception message is captured into this data structure. A state change database entry for the application is recorded. The DebugUi for the thread is located, and its debug state change semaphore is signaled (this semaphore is shared between a DebugUi and the Dbg server).

- o At this point in time it is important to note that while three threads have participated so far, only one thread remains blocked. The blocked thread is the application thread that caused the access violation. This thread is waiting for a reply to its original exception message.

- o At some point in time, the thread's DebugUi will call the **DbgUiWaitStateChange** API. This API waits on the debug state change semaphore. When the semaphore becomes signaled, the DebugUi formats a DBGUI_WAIT_STATE_CHANGE message and sends the message to the Dbg server.

- o Upon receipt of the DBGUI_WAIT_STATE_CHANGE message, the Dbg server scans the state change database for the DebugUi. Finding a state change record, the Dbg server populates the DBGUI_WAIT_STATE_CHANGE message and replies to the DebugUi.

- o The DebugUi returns from its call to **DbgUiWaitStateChange**. A state change type of **DbgExceptionStateChange** is is returned, along with the Client Id of the thread that originally caused the access violation. Using this information, the DebugUi can take appropriate action. This may include reading and writing the threads registers using **NtGetContextThread**/NtSetContextThread or reading and writing the processes virtual memory using **NtReadVirtualMemory**/NtWriteVirtualMemory.

- o Once the DebugUi is done servicing the access violation, it can continue the thread's execution by calling **DbgUiContinue**. This API simply formats a DBGUI_CONTINUE message and sends it to the Dbg server.

- o Upon receipt of the continue message, the Dbg server locates the target thread, assures that it is waiting to be continued, and that an appropriate continue status was passed. Dbg server then sends a continue datagram to the OS2SS. After this is complete, a reply is generated to the DebugUi which is then free to wait for further debug events.

- o Upon receipt of the continue datagram by DbgSs DLL code runing in the OS2SS, the continue status is examined and appropriate callouts are made. The DLL code then generates a reply to the original DBGKM_EXCEPTION message.

- o Upon receipt of this reply, the thread begins executing in the DbgKm component. All of the threads in its process are unfrozen and the thread returns to the exception dispatcher

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2. Debug Architecture Partitioning

The NT OS/2 Debug Architecture partitions the work involved in debugging into a number of stages.

2.1 Debug Event Generation

Debug event generation is done in the Dbgk component of the NT OS/2 executive. For each debug event, the following occurs:

- The process in which the debug event is occuring in is located.
- If the process has a DebugPort, then all threads in the process are frozen.
- A DBGKM_APIMSG is formated to indicate the type of debug event. This message is sent through the processes DebugPort using **LpcRequestWaitReplyPort**.
- Upon receipt of the reply, all threads in the process are unfrozen.

Debug events are generated for a number of reasons:

- **Exception**. When a thread whose process has a DebugPort encounters an exception, a debug event is generated.
- **CreateThread**. When a thread begins executing in a process being debugged, a debug event is generated before the thread gets a chance to execute in kernel mode.
- **CreateProcess**. When the first thread in a process being debugged begins executing, a debug event is generated before the thread gets a chance to execute in kernel mode.
- **ExitThread**. When a thread exits in a process being debugged, a debug event is generated. This occurs as soon as the system detects that the thread is exiting and has updated the exit status for the thread.
- **ExitProcess**. When the last thread in a process being debugged exits, a debug event is generated. This occurs as soon as the the status of the process has been updated. Note that when the last thread in a process exits, an exit thread debug event is not generated.
- **MapSection**. When a process being debugged maps a view of a section backed by an image file, a debug event is generated.
- **UnMapSection**. When a process being debugged un-maps a view of a section backed by an image file, a debug event is generated.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.1 Event Generation Message Formats

Event generation messages are always sent in the context of the thread reporting the event; therefore, the client id stored in the message header can be used to determine the thread reporting the event. Event generation messages consist of the following standard header:

```
typedef struct _DBGKM_APIMSG {
    PORT_MESSAGE h;
    DBGKM_APINUMBER ApiNumber;
    NTSTATUS ReturnedStatus;
    union u;
} DBGKM_APIMSG, *PDBGKM_APIMSG;
```

DBGKM_APIMSG Structure:

h —Supplies the standard LPC port message. The ClientId field of this structure supplies the client id of the thread reporting the debug event. The message type field (h.u2.s2.Type) is LPC_DEBUG_EVENT.

ApiNumber —Supplies the *ApiNumber* for this message. The *ApiNumber* is used to indicate the type of event being generated.

ReturnedStatus —Returns the continuation status for the debug event.

u —Supplies the type specific event information.

2.1.1.1 Exception

If the *ApiNumber* is **DbgKmExceptionApi**, then *u*.Exception supplies a DBGKM_EXCEPTION message. The format of this message follows:

```
typedef struct _DBGKM_EXCEPTION {
    EXCEPTION_RECORD ExceptionRecord;
    BOOLEAN FirstChance;
} DBGKM_EXCEPTION, *PDBGKM_EXCEPTION;
```

DBGKM_EXCEPTION Structure:

ExceptionRecord —Supplies the exception record describing this exception.

FirstChance —Supplies a variable that if TRUE, indicates that this is the first time this debug event is being reported for this thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.1.2 CreateThread

If the *ApiNumber* is **DbgKmCreateThreadApi**, then *u*.CreateThread supplies a DBGKM_CREATE_THREAD message. The format of this message follows:

typedef struct **_DBGKM_CREATE_THREAD** {
    ULONG *SubSystemKey*;
    PVOID *StartAddress*;
} **DBGKM_CREATE_THREAD**, ***PDBGKM_CREATE_THREAD**;

DBGKM_CREATE_THREAD Structure:

SubSystemKey —This field is initialized to 0.

StartAddress —Supplies the initial starting address for the thread. This is really advisory, since anyone with THREAD_SET_CONTEXT access to the thread may change this and superseed the value of this field.

2.1.1.3 CreateProcess

If the *ApiNumber* is **DbgKmCreateProcessApi**, then *u*.CreateProcess supplies a DBGKM_CREATE_PROCESS message. The format of this message follows:

typedef struct **_DBGKM_CREATE_PROCESS** {
    ULONG *SubSystemKey*;
    HANDLE *Section*;
    DBGKM_CREATE_THREAD *InitialThread*;
} **DBGKM_CREATE_PROCESS**, ***PDBGKM_CREATE_PROCESS**;

DBGKM_CREATE_PROCESS Structure:

SubSystemKey —This field is initialized to 0.

Section —Supplies a handle to the section object that describes the initial address space of the process. If this field is NULL, then no handle exists. The handle is valid in the sending processes handle table.

InitialThread —Supplies a description of the first thread to execute in the process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.1.4 ExitThread

If the *ApiNumber* is **DbgKmExitThreadApi**, then *u*.ExitThread supplies a DBGKM_EXIT_THREAD message. The format of this message follows:

typedef struct **_DBGKM_EXIT_THREAD** {
    NTSTATUS *ExitStatus*;
} **DBGKM_EXIT_THREAD**, ***PDBGKM_EXIT_THREAD**;

DBGKM_EXIT_THREAD Structure:

ExitStatus —Supplies the exit status of the exiting thread.

2.1.1.5 ExitProcess

If the *ApiNumber* is **DbgKmExitProcessApi**, then *u*.ExitProcess supplies a DBGKM_EXIT_PROCESS message. The format of this message follows:

typedef struct **_DBGKM_EXIT_PROCESS** {
    NTSTATUS *ExitStatus*;
} **DBGKM_EXIT_PROCESS**, ***PDBGKM_EXIT_PROCESS**;

DBGKM_EXIT_PROCESS Structure:

ExitStatus —Supplies the exit status of the exiting process.

2.1.1.6 MapSection

If the *ApiNumber* is **DbgKmMapSectionApi**, then *u*.MapSection supplies a DBGKM_MAP_SECTION message. The format of this message follows:

typedef struct **_DBGKM_MAP_SECTION** {
    HANDLE *SectionHandle*;
    PVOID *BaseAddress*;
    ULONG *SectionOffset*;
    ULONG *ViewSize*;
} **DBGKM_MAP_SECTION**, ***PDBGKM_MAP_SECTION**;

DBGKM_MAP_SECTION Structure:

SectionHandle —Supplies a handle to the section mapped by the process. The handle is valid in the context of the sending process.

BaseAddress —Supplies the base address of where the section is mapped in the processes address space.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SectionOffset —Supplies the offset in the section where the processes mapped view begins.

ViewSize —Supplies the size of the mapped view.

2.1.1.7 UnMapSection

If the *ApiNumber* is **DbgKmUnMapSectionApi**, then *u*.UnMapSection supplies a DBGKM_UNMAP_SECTION message. The format of this message follows:

```
typedef struct _DBGKM_UNMAP_SECTION {
    PVOID BaseAddress;
} DBGKM_UNMAP_SECTION, *PDBGKM_UNMAP_SECTION;
```

DBGKM_UNMAP_SECTION Structure:

BaseAddress —Supplies the base address of where the section being un-mapped is in the processes address space.

2.2 Event Propagation

Event propagation occurs after a a thread receives a debug event message on a processes DebugPort. Upon receipt of the message, the thread adds any necessary information, and forwards the message to the Dbg server.

Event propagation occurs within an Emulation Subsystem. In order to minimize thread blocking in the subsystems, an asynchronous protocol is used to propagate debug events. The event propagation protocol occurs as follows:

- The event is generated. The thread generating the event reports the event using **LpcRequestWaitReplyPort** against its processes DebugPort. The thread remains blocked until a reply is received.

- The subsystem receives the debug event message. After processing the message, it determines whether or not to propagate the message to the Dbg server. If it does not propagate the message, then it must reply to the thread reporting the event.

- To propagate the message, a copy of the message is made, and is sent as a datagram to the Dbg server. After receiving the message, and receiving a "continue" from the controlling debugger user interface, the Dbg server sends a continue datagram back to the subsystem.

- A dedicated thread in the subsystem receives the continue datagram, locates the associated saved debug event message, and replies to the thread reporting the event.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.1 Emulation Subsystem APIs for Event Propagation

2.2.1.1 DbgSsInitialize

An Emulation Subsystem initializes itself so that it can participate in the NT OS/2 debug architecture using the following API:

NTSTATUS
DbgSsInitialize(
    IN HANDLE *KmReplyPort*,
    IN PDBGSS_UI_LOOKUP *UiLookUpRoutine*,
    IN PDBGSS_SUBSYSTEMKEY_LOOKUP *SubsystemKeyLookupRoutine* **OPTIONAL**,
    IN PDBGSS_DBGKM_APIMSG_FILTER *KmApiMsgFilter* **OPTIONAL**
)

Parameters:

KmReplyPort —Supplies a handle to the port that the subsystem receives DbgKm API messages on.

UiLookupRoutine —Supplies the address of a function that will be called upon receipt of a process creation message. The purpose of this function is to identify the client id of the debug user interface controlling the process.

SubsystemKeyLookupRoutine —Supplies the address of a function that will be called upon receipt of process creation and thread creation messages. The purpose of this function is to allow a subsystem to correlate a key value with a given process or thread.

KmApiMsgFilter —Supplies the address of a function that will be called upon receipt of a DbgKm Api message. This function can take any action. If it returns any value other than DBG_CONTINUE, **DbgSsHandleKmApiMsg** will not process the message. This function is called before any other call outs occur.

Return Value:

SUCCESS() —Initialization complete.

!SUCCESS() —Failure occured while connecting to the Dbg server

This function is called by a subsystem to initialize portions of the debug subsystem dll. The main purpose of this function is to set up callouts that are needed in order to use **DbgSsHandleKmApiMsg**, and to connect to the Dbg server.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.1.2 UiLookupRoutine

The *UiLookupRoutine* is called during the propagation of create process debug events. Its function is to locate the client id of the debugger user interface that controls the process whose creation is being reported.

NTSTATUS
(***PDBGSS_UI_LOOKUP**)(
    IN PCLIENT_ID *AppClientId*,
    OUT PCLIENT_ID *DebugUiClientId*
)

<u>Parameters:</u>

AppClientId —Supplies the client id of the application thread reporting the create process debug event.

DebugUiClientId —Returns the client id of the debugger user interface that controls the application process.

<u>Return Value:</u>

STATUS_SUCCESS —The application is being debugged, and the client id of its debugger user interface has been returned.

!SUCCESS() —The application is not known as being debugged. The create process debug event will not be propagated to the debug server. A reply is generated and sent to the thread reporting the debug event.

2.2.1.3 SubsystemKeyLookupRoutine

The *SubsystemKeyLookupRoutine* is called during the propagation of create process and create thread debug events. Its function is to allow a subsystem to associate a key value with the process or thread whose creation is being reported. Examples of this are OS/2 might want to associate a TID with each thread, and a PID with each process. The subsystem key value is informational only.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
(***PDBGSS_SUBSYSTEMKEY_LOOKUP**)(
    IN PCLIENT_ID *AppClientId*,
    OUT PULONG *SubsystemKey*,
    IN BOOLEAN *ProcessKey*
)

Parameters:

AppClientId —Supplies the client id of the application thread reporting the create process or create thread debug event.

SubsystemKey —Returns the subsystem key value to associate with the new process or thread.

ProcessKey —Supplies a flag which if TRUE indicates that a subsystem key for the process is needed; otherwise, a subsystem key for the thread is needed. Note that during the propagation of a create process debug event, this function is called twice. It is called once with *ProcessKey* set to TRUE, and once with *ProcessKey* set to FALSE.

Return Value:

STATUS_SUCCESS —A subsystem key was found and returned.

!SUCCESS() —A subsystem key was not located. This does not affect the propagation of the create process or create thread debug events. The *SubSystemKey* fields in the propagated messges will remain 0.

2.2.1.4 KmApiMsgFilter

The *KmApiMsgFilter* routine is called prior to debug event message propagation. The main purpose of this API is to allow a subsystem an opportunity to monitor debug events and to cancel the propagation of events.

NTSTATUS
(***PDBGSS_DBGKM_APIMSG_FILTER**)(
    IN OUT PDBGKM_APIMSG *ApiMsg*
)

Parameters:

ApiMsg —Supplies the DBGKM_APIMSG that is about to be propagated.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

DBG_CONTINUE —Message propagation will continue. Note that since this callout occurs before the other callouts, event propagation can still be cancelled (by the *UiLookupRoutine*).

Others —The message will not be propagated. No reply is generated for the debug event.

2.2.1.5 DbgSsHandleKmApiMsg

The **DbgSsHandleKmApiMsg** is called by a subsystem whenever a debug event message arrives. This is typically done in the subsystem's main API loop whenever a message arrives whose type is LPC_DEBUG_EVENT.

VOID
DbgSsHandleKmApiMsg(
    IN PDBGKM_APIMSG *ApiMsg*
)

Parameters:

ApiMsg —Supplies the debug event message to propagate to the debug server.

A number of callouts are performed prior to propagating the message:

- For all messages, the *KmApiMsgFilter* is called (if it was supplied during **DbgSsInitialize**). If this returns anything other that DBG_CONTINUE, the message is not propagated by this function. The caller is responsible for event propagation, and for replying to the thread reporting the debug event.

- For create process messages, the *UiLookupRoutine* is called. If a success code is returned than message is propagated. Otherwise, a reply is generated to the thread reporting the debug event.

- For create process and create thread messages, *SubsystemKeyLookupRoutine* is called. Failure does not effect propagation. It simply inhibits the update of the messages *SubSystemKey* field.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.2 Event Propagation Message Formats

Event propagation messages are sent as datagrams to the Dbg server. Event propagation messages consist of the following standard header:

typedef struct **_DBGSS_APIMSG** {
    PORT_MESSAGE *h*;
    DBGKM_APINUMBER *ApiNumber*;
    NTSTATUS *ReturnedStatus*;
    CLIENT_ID *AppClientId*;
    PVOID *ContinueKey*;
    union *u*.
} **DBGSS_APIMSG**, ***PDBGSS_APIMSG**;

DBGSS_APIMSG Structure:

h —Supplies the standard LPC port message.

ApiNumber —Supplies the *ApiNumber* for this message. The *ApiNumber* is used to indicate the type of event being propagated.

ReturnedStatus —Used to store the continuation status for the event.

AppClientId —Supplies the client id of the application thread reporting the debug event. This comes directly from the header of the associated DBGKM_APIMSG.

ContinueKey —Supplies the continue key, that must be returned from the Dbg server in order to cause a reply to be generated for the thread that is reporting the debug event.

u —Supplies the type specific event propagation information.

2.2.2.1 Exception

If the *ApiNumber* is **DbgSsExceptionApi**, then *u*.Exception supplies a DBGKM_EXCEPTION message. The message contains the same information as it did at the time the event was generated.

2.2.2.2 CreateThread

If the *ApiNumber* is **DbgSsCreateThreadApi**, then *u*.CreateThread supplies a DBGKM_CREATE_THREAD message. The message contains the same information as it did at the time the event was generated. If a *SubsystemKeyLookupRoutine* was called and returned success, then the *SubSystemKey* field is modified appropriately.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.2.3 CreateProcess

If the *ApiNumber* is **DbgSsCreateProcessApi**, then *u*.CreateProcess supplies a DBGSS_CREATE_PROCESS message. The format of this message follows:

typedef struct **_DBGSS_CREATE_PROCESS** {
    CLIENT_ID *DebugUiClientId*;
    DBGKM_CREATE_PROCESS *NewProcess*;
} **DBGSS_CREATE_PROCESS**, ***PDBGSS_CREATE_PROCESS**;

DBGSS_CREATE_PROCESS Structure:

DebugUiClientId —Supplies the client id of the processes debugger user interface.

NewProcess —Supplies the original contents of the DBGKM_CREATE_PROCESS message at the time the event was generated. If a *SubsystemKeyLookupRoutine* was called and returned success, then the *SubSystemKey* field is modified appropriately.

2.2.2.4 ExitThread

If the *ApiNumber* is **DbgSsExitThreadApi**, then *u*.ExitThread supplies a DBGKM_EXIT_THREAD message. The message contains the same information as it did at the time the event was generated.

2.2.2.5 ExitProcess

If the *ApiNumber* is **DbgSsExitProcessApi**, then *u*.ExitProcess supplies a DBGKM_EXIT_PROCESS message. The message contains the same information as it did at the time the event was generated.

2.2.2.6 MapSection

If the *ApiNumber* is **DbgSsMapSectionApi**, then *u*.MapSection supplies a DBGKM_MAP_SECTION message. The message contains the same information as it did at the time the event was generated.

2.2.2.7 UnMapSection

If the *ApiNumber* is **DbgSsUnMapSectionApi**, then *u*.UnMapSection supplies a DBGKM_UNMAP_SECTION message. The message contains the same information as it did at the time the event was generated.

2.3 Coordinate Debugger and Debuggee

The purpose of the Dbg server is to coordinate debug events occuring in the applications being debugged (debugee) with the requests for event notification coming from the applications debug user interface. In order to facilitate this type of coordination, the Dbg server maintains a set of data structures that bind a debugger with its debuggess, and that bind debuggees with their controlling subsystem.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The data structures used to do the bindings are created and modified based on the receipt of propagated debug event messages, connections to the Dbg server, and the receipt of wait and continue messages from debuggers.

The following sections describe the data structures maintained by the Dbg server, and the actions that causes the data structures to be created and modified.

2.3.1 Dbg Server Data Structures

2.3.1.1 Subsystem Structure

A subsystem structure exists for each subsystem connected to the Dbg server. A subsystem connects to the Dbg server as part of **DbgSsInitialize**. The connection port used during the connection has a security descriptor that limits access to only those processes that form part of the NT OS/2 TCB.

```
typedef struct _DBGP_SUBSYSTEM {
        CLIENT_ID SubsystemClientId;
        HANDLE CommunicationPort;
        HANDLE SubsystemProcessHandle;
} DBGP_SUBSYSTEM, *PDBGP_SUBSYSTEM;
```

DBGP_SUBSYSTEM Structure:

SubsystemClientId —Contains the client id of the subsystem thread that initially connects to the Dbg server.

CommunicationPort —Contains a handle to the communication port used to send continue datagrams back to the subsystem.

SubsystemProcessHandle —Contains a handle to the subsystem process. This handle has PROCESS_DUP_HANDLE access to the process.

2.3.1.2 User Interface Structure

A user interface structure is maintained for each debugger user interface (DebugUi) connected to the Dbg server. Debugger user interface's connect to the Dbg server as part of their initialization process. The connection must be made before the user interface starts any applications that need to be debugged.

The user interface structure is the key coordination data structure. All of the application processes and threads controlled by a user interface are linked off of the data structure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

typedef struct **_DBGP_USER_INTERFACE** {
    CLIENT_ID *DebugUiClientId*;
    HANDLE *CommunicationPort*;
    HANDLE *DebugUiProcess*;
    HANDLE *StateChangeSemaphore*;
    RTL_CRITICAL_SECTION *UserInterfaceLock*;
    LIST_ENTRY *AppProcessListHead*;
    LIST_ENTRY *HashTableLinks*;
} **DBGP_USER_INTERFACE**, ***PDBGP_USER_INTERFACE**;

DBGP_USER_INTERFACE Structure:

DebugUiClientId —Contains the client id of the thread that connects to the Dbg server. During the processing of a propagated create process debug event, the *DebugUiClientId* field of a DBGSS_CREATE_PROCESS structure is matched against this field. Once a match is found, the process is bound to the user interface.

CommunicationPort —Contains a handle to the communication port used to reply back to requests from the user interface.

DebugUiProcess —Contains a handle to the user interface process. The handle has PROCESS_DUP_HANDLE access to the process. This handle is used to duplicate object handles into and out of the user interace.

StateChangeSemaphore —Contains a handle to a semaphore shared between the Dbg server and the user interface. The semaphore is signaled each time a propagated debug event is available to be picked up by the user interface. The user interface waits on this semaphore. When a wait is satisfied, the user interface can call into the Dbg server to receive notification of debug events.

UserInterfaceLock —Contains a critical section lock to gaurd the user interface and associated structures.

AppProcessListHead —Contains a list head where processes being debugged by the user interface are linked.

HashTableLinks —Contains a set of link words use to quickly locate a user interface by client id.

2.3.1.3 Application Process Structure

An application process structure is maintained for each process accepted by the Dbg server. For a process to be accepted, a process creation debug event must propagated to the Dbg server, and the *DebugUiClientId* field of the DBGSS_APIMSG must match the client id of a user interface connected to Dbg. Once this occurs, an application process structure is

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

typedef struct **_DBGP_APP_PROCESS** {
    LIST_ENTRY *AppThreadListHead*;
    LIST_ENTRY *AppLinks*;
    LIST_ENTRY *HashTableLinks*;
    CLIENT_ID *AppClientId*;
    PDBGP_USER_INTERFACE *UserInterface*;
    HANDLE *HandleToProcess*;
} **DBGP_APP_PROCESS**, ***PDBGP_APP_PROCESS**;

DBGP_APP_PROCESS Structure:

AppThreadListHead —Contains the list head for all application threads that form this process.

AppLinks —Contains the link words that are used to link the process to its user interface.

HashTableLinks —Contain a set of link words to quickly locate an application process by its UniqueProcess portion of its client id.

AppClientId —Contains the client id of the processes initial thread. Only the UniqueProcess portion of the client id is used.

UserInterface —Contains a pointer to the processes user interface.

HandleToProcess —Contains a handle to the application process. When the application process structure is initially created, a handle is created in the context of the Dbg server. The handle has PROCESS_VM_READ and PROCESS_VM_WRITE access. When the user interface waits for and receives notification of the new process debug event, the handle is duplicated into the user interface and closed in the Dbg server. This field is then modified so that the handle value is a handle in the context of the user interface. When the user interface issues a continue after an exit process debug event, the handle to the process is closed and is no longer available to the user interface.

2.3.1.4 Application Thread Structure

An application thread structure is maintained for each thread accepted by the Dbg server. For a thread to be accepted, a thread creation debug event must propagated to the Dbg server, and the process that the thread is part of must have been previously accepted by Dbg.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

typedef struct **_DBGP_APP_THREAD** {
    LIST_ENTRY *AppLinks*;
    LIST_ENTRY *HashTableLinks*;
    CLIENT_ID *AppClientId*;
    DBG_STATE *CurrentState*;
    DBG_STATE *ContinueState*;
    PDBGP_APP_PROCESS *AppProcess*;
    PDBGP_USER_INTERFACE *UserInterface*;
    HANDLE *HandleToThread*;
    PDBGP_SUBSYSTEM *Subsystem*;
    DBGSS_APIMSG *LastSsApiMsg*;
} **DBGP_APP_THREAD**, ***PDBGP_APP_THREAD**;

DBGP_APP_THREAD Structure:

AppLinks —Contains link words that link an application thread to its process.

HashTableLinks —Contain a set of link words to quickly locate an application thread by its client id.

AppClientId —Contains the client id of the thread.

CurrentState —Contains the Dbg server maintained state for the thread. The thread can be in three state clasess. If a debug event has been propagated to and has been accepted by the Dbg server, the state class is "state change available". Once a user interface has received notification of the state change, the class becomes "continue pending". When a user interface issues a continue to a thread whose state class is "continue pending", the threads state class returns to "idle".

ContinueState —Contains the saved Dbg server state for the thread at the time the thread transitions from "state change available" to "continue pending".

AppProcess —Contains a pointer to the threads process.

UserInterface —Contains a pointer to the threads user interface.

HandleToThread —Contains a handle to the application thread. When the application thread structure is initially created, a handle is created in the context of the Dbg server. The handle has THREAD_GET_CONTEXT and THREAD_SET_CONTEXT access. When the user interface waits for and receives notification of the new thread debug event, the handle is duplicated into the user interface and closed in the Dbg server. This field is then modified so that the handle value is a handle in the context of the user interface. When the user interface issues a continue after an exit thread debug event, or after an exit process debug event, the handle to the thread is closed and is no longer available to the user interface.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Subsystem —Contains a pointer to the thread's subsystem. This is used to locate the subsystem to send a continue datagram to when the thread's user interface issues a continue.

LastSsApiMsg —This field contains the last DBGSS_APIMSG for the thread. This message is valid while a thread is in the "state change available" state class. Portions of this message are made available to the user interface when it receives notification of the occurance of a debug event.

2.3.2 Dbg Server Responses to Debug Event Propagation

The Dbg server responds to propagated debug events by creating or modifying its user interface, application process, or application thread data structures.

Debug events are are propagated to the Dbg server by a subsystem that is connected to Dbg and is identified by its subsystem structure. Upon receipt of a propagated debug event message, Dbg determines whether or not to accept the message. If the message is accepted, then the message is captured into the appropriate application thread structure, the thread's state is changed to the "state change available" class, and the appropriate user interface's *StateChangeSemaphore* is signaled.

The following sections describe the actions taken by the Dbg server upon receipt of a propagated debug event message:

2.3.2.1 Exception

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the exception message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are modified to either **DbgBreakpointStateChange**, **DbgSingleStepStateChange**, or DbgExceptionState change base on the **ExceptionCode** field of the *ExceptionRecord*.

2.3.2.2 CreateThread

The application process structure that the thread is part of is located. If the process can not be found , the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the create thread message is accepted and the following occurs:

- An application thread structure for the thread is allocated.
- The *CurrentState* and *ContinueState* fields are initialized to **DbgCreateThreadStateChange**.
- The *AppProcess* field is set to point to the thread's process, the *UserInterface* field is initialized to point to the thread's user interface, the *AppClientId* field is initialized, and the thread is linked to its process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- A handle to the thread is created in the context of the Dbg server. If this operation succeeds, then the *HandleToThread* field is initialized to the value of the handle; otherwise, it is initialized to NULL.

2.3.2.3 CreateProcess

The user interface whose client id is specified in the message is located. If the user interface can not be located (the user interface has not connected to Dbg), the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the create process message is accepted and the following occurs:

- An application process structure for the process is allocated.
- The *UserInterface* field is initialized to point to the process' user interface, the *AppClientId*.UniqueProcess field is initialized.
- The process is linked to its user interface.
- A handle to the process is created in the context of the Dbg server. If this operation succeeds, then the *HandleToProcess* field is initialized to the value of the handle; otherwise, it is initialized to NULL.
- An application thread structure for the thread described in the create process message is allocated.
- The *CurrentState* and *ContinueState* fields are initialized to **DbgCreateProcessStateChange**.
- The *AppProcess* field is set to point to the thread's process, the *UserInterface* field is initialized to point to the thread's user interface, the *AppClientId* field is initialized, and the thread is linked to its process.
- A handle to the thread is created in the context of the Dbg server. If this operation succeeds, then the *HandleToThread* field is initialized to the value of the handle; otherwise, it is initialized to NULL.

2.3.2.4 ExitThread

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the exit thread message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are set to **DbgExitThreadStateChange**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.3.2.5 ExitProcess

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the exit process message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are set to **DbgExitProcessStateChange**.

2.3.2.6 MapSection

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the map section message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are set to **DbgMapSectionStateChange**.

2.3.2.7 UnMapSection

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the un-map section message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are set to **DbgUnMapSectionStateChange**.

2.4 User Interface Interactions with the Dbg Server

A debug user interface has three main interactions with the Dbg server.

- Connecting to the Dbg server
- Waiting for debug event state changes to occur
- Continuing an application thread

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.4.1 DbgUiConnectToDbg

A user interface can connect to the Dbg server using **DbgUiConnectToDbg**.

NTSTATUS
DbgUiConnectToDbg(**VOID**)

Return Value:

SUCCESS() —A connection between the user interface and the Dbg server has been made. The DbgUi dll has been initialized.

!SUCCESS() —The connection to Dbg did not occur. The user interface can not use services provided by the Dbg server.

This routine makes a connection between the calling user interface and the Dbg server. If the routine is successful, a communications port is created to link the user interface with the Dbg server. A user interface data structure is created and initialized in the Dbg server. A shared state change semaphore is created between the Dbg server and the user interface. The Dbg server is granted SEMAPHORE_ALL_ACCESS to the semaphore. This allows it to signal the semaphore at the appropriate times. A handle to the semaphore is duplicated to the user interface. The handle is granted SYNCHRONIZE access to the semaphore. This allows the user interface an opportunity to wait on the semaphore. The semaphore becomes signaled when a propagated debug event is available which transitions one of the user interface's threads into the "state change available" state.

2.4.2 DbgUiWaitStateChange

A user interface can wait for a state change to occur in one of its threads using **DbgUiWaitStateChange**.

NTSTATUS
DbgUiWaitStateChange(
    OUT PDBGUI_WAIT_STATE_CHANGE *StateChange*
)

Parameters:

StateChange —Supplies the address of state change record that will contain the state change information.

Return Value:

STATUS_USER_APC —A user mode APC occured which caused this call to abort without retreiving state change information.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

STATUS_ALERTED —The thread was alerted while waiting for a state change to occur. No state change information was retreived.

DBG_NO_STATE_CHANGE —The state change semaphore was signaled, but the Dbg server has no state change information to return. This error can only happen if a user interface bypasses the DbgUi APIs and attempts to communicate directly with the Dbg server.

DBG_UNABLE_TO_PROVIDE_HANDLE —A state change occurred that required a handle to be duplicated into the user interface. For some reason, a handle could not be provided. All other portions of the state change reporting were successful.

STATUS_SUCCESS —A state change occured. Valid state change information was returned.

OTHERS —Refer to object management error codes.

This function causes the calling user interface to wait for a state change to occur in one of it's application threads. The wait is ALERTABLE. A state change occurs when an application thread changes its state to the "state change available" class. If a user interface makes a successful call to this function while one of its threads is in the "state change available" class, then the threads state is set to "continue pending", and a state change record is formatted and returned to the caller. Once a state change has been reported for a thread, its user interface is responsible for continuing the thread at the appropriate time.

2.4.2.1 State Change Record

A state change record has the following format:

typedef struct **_DBGUI_WAIT_STATE_CHANGE** {
    DBG_STATE *NewState*;
    CLIENT_ID *AppClientId*;
    union *StateInfo*;
} **DBGUI_WAIT_STATE_CHANGE**, ***PDBGUI_WAIT_STATE_CHANGE**;

DBGUI_WAIT_STATE_CHANGE Structure:

NewState —Supplies the new state of the thread reporting the state change.

AppClientId —Supplies the client id of the thread reporting the state change.

StateInfo —Supplies the per-state change type description that describes the state change.

2.4.2.1.1 DbgCreateThreadStateChange

The state change of **DbgCreateThreadStateChange** is reported whenever a state change is reported due to the propagation of a create thread debug event. The major side effect of this state change is that the user interface is given a handle to the thread reporting the state change. The handle is granted

THREAD_GET_CONTEXT and THREAD_SET_CONTEXT access to the thread. This allows the user interface to use NtReadContextThread and NtWriteContextThread to read and write the thread's registers.

StateInfo for this type of state change is as follows:

typedef struct **_DBGUI_CREATE_THREAD** {
    HANDLE *HandleToThread*;
    DBGKM_CREATE_THREAD *NewThread*;
} **DBGUI_CREATE_THREAD**, ***PDBGUI_CREATE_THREAD**;

DBGUI_CREATE_THREAD Structure:

HandleToThread —Supplies a handle to the thread identified by this state change. A value of NULL indicates that the handle is not valid and that an informational status code of DBG_UNABLE_TO_PROVIDE_HANDLE was returned.

NewThread —Supplies the description of the new thread as formatted by the subsystem during debug event propagation.

2.4.2.1.2 DbgCreateProcessStateChange

The state change of **DbgCreateProcessStateChange** is reported whenever a state change is reported due to the propagation of a create process debug event. The major side effects of this state change are:

- The user interface is given a handle to the process of the thread reporting the state change. The handle is granted PROCESS_VM_READ and PROCESS_VM_WRITE access to the process. This allows the user interface to use **NtReadVirtualMemory** and **NtWriteVirtualMemory** to read and write the processes virtual memory.

- The user interface is given a handle to the section that forms the initial address space for the process reporting the state change. The handle is granted SECTION_ALL_ACCESS access to the section. This allows the user interface to map a view of the section to locate the symbol table and other section information.

- The user interface is given a handle to the thread reporting the state change. The handle is granted THREAD_GET_CONTEXT and THREAD_SET_CONTEXT access to the thread. This allows the user interface to use NtReadContextThread and NtWriteContextThread to read and write the thread's registers.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef struct _DBGUI_CREATE_PROCESS {
        HANDLE HandleToProcess;
        HANDLE HandleToThread;
        DBGKM_CREATE_PROCESS NewProcess;
} DBGUI_CREATE_PROCESS, *PDBGUI_CREATE_PROCESS;
```

DBGUI_CREATE_PROCESS Structure:

HandleToProcess —Supplies a handle to the process identified by this state change. A value of NULL indicates that the handle is not valid and that an informational status code of DBG_UNABLE_TO_PROVIDE_HANDLE was returned.

HandleToThread —Supplies a handle to the thread identified by this state change. A value of NULL indicates that the handle is not valid and that an informational status code of DBG_UNABLE_TO_PROVIDE_HANDLE was returned.

NewProcess —Supplies the description of the new process as formatted by the subsystem during debug event propagation. The **Section** field of this structure is modified to contain a handle that is valid in the user interfaces context. A value of NULL indicates that the handle is not valid and that an informational status code of DBG_UNABLE_TO_PROVIDE_HANDLE was returned.

2.4.2.1.3 DbgExitThreadStateChange

The state change of **DbgExitThreadStateChange** is reported whenever a state change is reported due to the propagation of an exit thread debug event. There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.4 DbgExitProcessStateChange

The state change of **DbgExitProcessStateChange** is reported whenever a state change is reported due to the propagation of an exit process debug event. There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.5 DbgExceptionStateChange

The state change of **DbgExceptionStateChange** is reported whenever a state change is reported due to the propagation of an exception debug event where the exception code is anything other than STATUS_BREAKPOINT or STATUS_SINGLE_STEP. There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.4.2.1.6 DbgBreakpointStateChange

The state change of **DbgBreakpointStateChange** is reported whenever a state change is reported due to the propagation of an exception debug event where the exception code is STATUS_BREAKPOINT . There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.7 DbgSingleStepStateChange

The state change of **DbgSingleStepStateChange** is reported whenever a state change is reported due to the propagation of an exception debug event where the exception code is STATUS_SINGLE_STEP . There are no side effects of this state change.

StateInfo for this type of state change is as that same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.8 DbgMapSectionStateChange

The state change of **DbgMapSectionStateChange** is reported whenever a state change is reported due to the propagation of a map section debug event. The major side effects of this state change are:

- o The user interface is given a handle to the section being mapped by reporting the state change. The handle is granted SECTION_ALL_ACCESS access to the section. This allows the user interface to map a view of the section to locate the symbol table and other section information.

2.4.2.1.9 DbgUnMapSectionStateChange

The state change of **DbgUnMapSectionStateChange** is reported whenever a state change is reported due to the propagation of an un-map section debug event. There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.4.3 DbgUiContinue

A user interface can continue a thread that previously reported a state change using **DbgUiContinue**.

NTSTATUS
DbgUiContinue(
    IN PCLIENT_ID *AppClientId*,
    IN NTSTATUS *ContinueStatus*
)

Parameters:

AppClientId —Supplies the address of the ClientId of the application thread being continued. This must be an application thread that previously notified the caller through **DbgUiWaitStateChange** but has not yet been continued.

ContinueStatus —Supplies the continuation status to the thread being continued. Valid values for this are DBG_EXCEPTION_HANDLED, DBG_EXCEPTION_NOT_HANDLED, DBG_TERMINATE_THREAD, DBG_TERMINATE_PROCESS, or DBG_CONTINUE.

Return Value:

STATUS_SUCCESS —Successful call to **DbgUiContinue**

STATUS_INVALID_CID —An invalid ClientId was specified for the *AppClientId*, or the specified Application was not waiting for a continue.

STATUS_INVALID_PARAMETER —An invalid continue status was specified.

Continuing an application thread has a number of side effects. In some cases data structures inside of the Dbg server are modified or event deallocated. This is all dependent upon the *ContinueState* of the thread being continued. A number of standard actions occur during a continue regardless of the thread's *ContinueState*:

- A check is made to ensure that the *ContinueStatus* is valid, that the thread is known to Dbg, and that the thread is in the "continue pending" state.
- Perform and *ContinueState* dependent side effects.
- Format a continue datagram and send it to the thread's subsystem. This is then picked up by the subsystem which uses the continue key to reply to the original DBGKM_APIMSG which generated the debug event. Once the reply is received the thread which genrated the original debug even can continue execution.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following sections describe the *ContinueState* dependent side effects of **DbgExitThreadStateChange** and **DbgExitProcessStateChange** state changes. No other state change types have side effects.

2.4.3.1 DbgExitThreadStateChange

Continuing a thread whose continue state is **DbgExitThreadStateChange**, causes the Dbg server to deallocate its application thread structure. If a handle to the thread was successfully duplicated into the user interface, the handle is closed. Once a user interface continues a thread in this state, it can no longer read and write the thread's registers.

2.4.3.2 DbgExitProcessStateChange

Continuing a thread whose continue state is **DbgExitProcessStateChange**, causes the Dbg server to deallocate its application process structure. If a handle to the process was successfully duplicated into the user interface, the handle is closed. Since this also implies that an application thread has exited, the thread's application thread structure is deallocated. If a handle to the thread was successfully duplicated into the user interface, the handle is closed.

Once a user interface continues a thread in this state, it can no longer read and write the thread's registers, or read and write the processes virtual memory.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Driver Model Specification

Author: *Darryl E. Havens*

Revision 1.2, July 20, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction........ 1

2. Overview........ 1

3. Driver Model Description........ 5
   3.1. Time-Out Handling........ 9
   3.2. Power Recovery........ 10
   3.3. Canceling I/O........ 12
   3.4. Driver Layering........ 12

4. File System Description........ 14
   4.1. IFS Design........ 16
   4.2. Mapped File I/O........ 17
   4.3. File Caching........ 18
   4.4. Splitting Transfers........ 18
      4.4.1. FSP Model........ 18
      4.4.2. FSD Parallel Model........ 20
      4.4.3. FSD Serial Model........ 20
   4.5. Mounting and Volume Verification........ 20

5. Network Service Description........ 22

6. I/O Completion........ 22

7. Error Logging and Handling........ 24
   7.1. Error Logging Facility........ 24
   7.2. Error Ports........ 25

8. Terminal I/O Considerations........ 26
   8.1. Unsolicited Input........ 26
   8.2. Subsystem Input........ 26

9. I/O Data Structures and Objects........ 27
   9.1. I/O Request Packet Description........ 27
   9.2. Volume Parameter Block........ 28
   9.3. File Object........ 28
   9.4. Driver Object........ 29
   9.5. Device Object........ 29
   9.6. Controller Object........ 30
   9.7. Adapter Object........ 30

10. I/O System APIs........ 30
   10.1. IoAbortInvalidRequest........ 32
   10.2. IoAllocateAdapterChannel........ 32

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.3. IoAllocateController 34
10.4. IoAllocateErrorLogEntry 35
10.5. IoAllocateIrp 35
10.6. IoAllocateMdl 36
10.7. IoAsynchronousPageWrite 37
10.8. IoAttachDeviceByName 38
10.9. IoBuildAsynchronousFsdRequest 38
10.10. IoBuildFspRequest 39
10.11. IoBuildPartialMdl 40
10.12. IoBuildSynchronousFsdRequest 41
10.13. IoCallDriver 42
10.14. IoCancelThreadIo 43
10.15. IoCheckDesiredAccess 43
10.16. IoCheckFunctionAccess 44
10.17. IoCheckShareAccess 45
10.18. IoCompleteRequest 46
10.19. IoCreateController 46
10.20. IoCreateDevice 46
10.21. IoCreateFile 47
10.22. IoCreateStreamFile 49
10.23. IoDeallocateAdapterChannel 50
10.24. IoDeallocateController 50
10.25. IoDeallocateIrp 50
10.26. IoDeleteController 51
10.27. IoDeallocateMdl 51
10.28. IoDeleteDevice 52
10.29. IoDeregisterFileSystem 52
10.30. IoDetachDevice 52
10.31. IoFlushAdapterBuffers 53
10.32. IoGetAttachedDevice 53
10.33. IoGetCurrentIrpStackLocation 54
10.34. IoGetNextIrpStackLocation 54
10.35. IoGetRelatedDeviceObject 55
10.36. IoGetRequestorProcess 55
10.37. IoInitializeDpcRequest 55
10.38. IoInitializeTimer 56
10.39. IoIsOperationSynchronous 57
10.40. IoMakeAssociatedIrp 57
10.41. IoMapTransfer 58
10.42. IoPageRead 59
10.43. IoQueryInformation 60
10.44. IoRegisterFileSystem 60
10.45. IoRemoveShareAccess 61

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.46. IoRequestDpc 61
10.47. IoSendMessage 62
10.48. IoSetCompletionRoutine 62
10.49. IoSetShareAccess 64
10.50. IoStartNextPacket 64
10.51. IoStartPacket 65
10.52. IoStartTimer 66
10.53. IoStopTimer 66
10.54. IoSynchronousPageWrite 66
10.55. IoUpdateShareAccess 67
10.56. IoWriteErrorLogEntry 68

11. I/O System Folklore 68
11.1. Rules for Completing an I/O Request 68
11.2. Accessing Another Driver 70
11.3. Generating Packets 70
11.4. Direct vs. Buffered vs. Neither I/O 71
11.5. Building Virtually Discontiguous Buffers 74
11.6. I/O Services Synchronization 74

12. Revision History 76

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

The **Windows NT** I/O system provides system programmers the features necessary to write their own device drivers for those devices that **Windows NT** does not support as part of its regular SDK. The driver interface is designed to allow these programmers to write all device drivers in a high-level language. **Windows NT** provides all of the necessary include files to write these drivers in C.

This specification describes the basic flow of control of an I/O request from the requestor's call, through the I/O system, through the device driver, and back through the I/O system to the requestor. It does not attempt to exhaustively enumerate all of the error conditions nor does it attempt to specify how every type of device is to be dealt with in this design.

This specification also describes the basic driver model, how file systems and network systems fit into that model, and then describes the data structures and I/O APIs used to support the model.

For background information about the I/O system API used by code external to the I/O system, please see the *Windows NT I/O System Specification*.

2. Overview

This section presents an overview of the sequence of operations that take place when an I/O operation is requested.

When a user invokes the **NtCreateFile** or the **NtOpenFile** service, the system attempts to translate the name of the supplied file specification that is to be accessed. If the name successfully translates to a device object, then the system passes the remainder of the file specification, if any, to the parse routine for the object.

All I/O services begin by performing the following operations, except as noted. Arguments and addresses are captured as appropriate.

o - The caller's arguments are probed for read access by the previous mode.

o - The file handle is translated, referenced, and checked for validity. If the handle is valid, then it is set to the Not-Signaled state. This obviously does not occur on an open or a create operation.

o - The event object handle, if specified, is translated, referenced, and set to the Not-Signaled state, if valid.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - The caller's I/O status block is alignment checked and probed for write access by the previous mode.

o - The caller's buffers, if any, are probed for the appropriate access by the previous mode. This only occurs at this point if it is known that the I/O being performed is *buffered I/O*[1].

o - On an open or create call, the file name is parsed to determine the device for which the operation is destined.

o - All other parameters specific to device-independent services are checked for accessibility and validity.

Once the above checks have been made and it is determined that the caller's device-independent parameters are valid, an *I/O Request Packet* **(IRP)** is allocated and the parameters are marshalled into it. Some parameters, such as the length of a buffer, are simply copied. Others, such as the handle for an event, are put into the IRP as pointers to objects rather than as handles to them. Also at this point, any user buffers that need to be locked into memory are probed and locked. This causes a *Memory Descriptor List* **(MDL)** to be built that describes the pages that are locked. The address of the MDL is also stored in the IRP.

The IRP is then handed to the driver's entry point according to the major function code of the request. This routine is given a pointer to one of its device objects (the one that the request is for) and a pointer to the IRP. It is the driver's responsibility to validate the remainder of the parameters and then, if valid, to start the I/O operation. Drivers can use the I/O system routine, **IoStartPacket**, to pass the packet to the start I/O routine. This function gives the IRP directly to the driver's start I/O routine if the device is not busy and sets the driver's busy flag. If the device is already busy, then it simply queues the packet to the driver's request queue. All synchronization of the queue is handled in this routine via the use of a *device queue*, a kernel-provided object designed just for this purpose.

Regardless of whether the I/O actually gets started, the packet simply gets queued for later processing, or an error of some kind occurred, the driver's major function routine returns to the I/O system indicating whether or not everything up to this point has been successful. The I/O system then returns to complete the original user call. If the operation was successfully started or queued, then it is up to the

---

[1] *Buffered I/O* refers to I/O being performed to an intermediary buffer. *Direct I/O* refers to I/O being performed directly on the original buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

caller to synchronize itself with the completion of the I/O operation, unless one of the synchronous options has been specified.

If an error does occur and the I/O operation was neither started nor queued for later processing by the driver, then the operation is considered to be in error and will never be "completed". That is, an error status code is returned to the caller indicating that the operation failed. The file object will not be set to the Signaled state, nor will the event, if one was specified. The APC routine is not executed and the state of the I/O status block is undefined.

The start I/O routine must synchronize access to the device so that the *Interrupt Service Routine* **(ISR)** cannot access the device at the same time as the start I/O routine. Three items of interest must be considered. The start I/O routine must synchronize with the following:

o - Interrupts occurring on the current processor

o - The same start I/O routine executing on another processor

o - Access to the device with power failures

These synchronization issues will be discussed in detail later. Note that this type of synchronization is particularly interesting when dealing with controllers that service multiple devices. It is possible to write a device driver that depends on the state of a busy flag to synchronize access to the device, provided that no unsolicited input interrupt can be taken from the device. That is, for devices of this type, the start I/O routine will never be invoked when an interrupt can occur because the busy flag would already be set. Of course, synchronization with powerfail interrupts must still be dealt with by the driver.

Once the I/O operation completes on the device, the device requests an interrupt. The interrupt will be taken when the processor's *Interrupt Request Level* **(IRQL)** is at a lower level than that of the requesting interrupt. The interrupt dispatcher then uses the *interrupt object*, created when the driver was initialized, to invoke the device driver's ISR. The ISR is executed at the same priority level that the device interrupted. This means that no other devices at the same interrupt request level can be serviced until the current interrupt servicing has been completed.

Device drivers are written to perform as little work as possible in their interrupt service routines. Notice also that access to the device needs to be synchronized with the remainder of the driver. Because **Windows NT** supports multiprocessor systems, other parts of the device driver could be accessing the device or some common data base from another processor. (They could not be doing so on the current processor because part of synchronizing with the ISR is raising the current

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IRQL to the level that the device interrupts, and this blocks the device from interrupting on the current processor.) A per-device spin lock is used to perform this synchronization among the processors. The steps a device driver takes in order to synchronize with the interrupt service routine are as follows:

1. Save the current IRQL.

2. Raise the IRQL to the device interrupt priority level.

3. Obtain the spin lock for the device.

4. Raise IRQL to block powerfail interrupts and check for power failure. If a failure has occurred, do not perform the next step.

5. Manipulate device registers.

6. Release spin lock.

7. Lower IRQL back to saved IRQL.

Notice that many drivers will simply raise the IRQL to block power failure interrupts during step 2, hence saving extra time by not setting the IRQL twice. The main reason that drivers might wish to use the above steps as written is that they might want to do more than simply manipulate device registers at raised IRQL. It is much better to do the work between steps 3 and 4 at device IRQL than it is at powerfail IRQL.

Programmers developing device drivers for **Windows NT** need not concern themselves with the particulars of how to synchronize access between the ISR and other parts of the device driver. The kernel provides a synchronization mechanism explicitly designed to aid in writing device drivers. When the device driver is being initialized, it creates an entity called an *interrupt object*. An interrupt object allows the device driver to describe to the system what IRQL its ISR should be associated with. This object can then be used in calls to another kernel-provided routine, **KeSynchronizeExecution**. This routine provides the above access synchronization, except for synchronization with power recovery interrupts.

Power recovery is handled through the use of two other kernel-provided objects, the *power status object* and the *power notify object*. The former object provides drivers with the ability to specify a Boolean variable that should be set in the event that the power has failed and then come back on with the contents of dynamic memory preserved. This allows drivers to provide the synchronization in step 4 above. The latter object provides drivers with the ability to register a routine that should be

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

invoked should a power recovery occur. This gives the driver a chance to reinitialize its device, handle any problems that may have occurred if an operation was currently in progress, perform cleanup operations, etc.

Once the driver ISR has completed its interrupt service processing, most of the time it will need to do more work at a lower priority level, such as start another I/O operation or "complete" the current operation. This can be done by requesting the execution of a *Deferred Procedure Call* **(DPC)**. The DPC queue, a kernel-provided mechanism, allows the device driver to request the execution of a routine at a later time at a lower IRQL. The I/O system provides a set of routines that allow the device driver to use this mechanism. The following steps provide this functionality to the driver:

> o - The device driver, in its initialization routine, initializes the DPC in its device object to specify the address of the routine that is to be executed when the DPC is requested. This is done through the use of the **IoInitializeDpcRequest** routine.
>
> o - When the interrupt service routine wishes to request that a DPC routine be executed later, it invokes the **IoRequestDpc** routine. This routine uses the kernel-provided routine to insert the DPC into the system's DPC queue.
>
> o - The DPC routine is executed after other higher level interrupts have been dismissed and the DPC queue is being processed. This may happen on any processor, including a different processor than the one on which the original interrupt occurred.

When the DPC queue interrupt occurs, at DISPATCH_LEVEL, the system-provided ISR examines the DPC queue to determine if there is any work to be performed. If so, then it removes a DPC entry from the queue and processes it. Processing consists of invoking the specified routine with a pointer to the DPC entry itself as well as the parameters specified in the call to **IoRequestDpc**.

It is at DISPATCH_LEVEL IRQL that the driver performs the majority of its work. It is here that the driver determines whether any errors have occurred, performs its error logging if needed, cleans up its context for the operation and determines whether there are more operations to do and starts them. All work that does not absolutely have to be performed in the ISR at device IRQL is done in this DPC routine.

The system provides a routine, **IoStartNextPacket**, that removes the next packet from the front of the device queue and returns a pointer to it to the caller, making the next IRP in the queue the "current" packet. If the device queue was empty, then

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

the function returns a null pointer. This function assumes that the device busy flag is already set and will actually bugcheck the system if it is not set. If there was no other packet in the queue, then the busy flag is cleared.

Once the driver has completed its own processing, it then invokes the system routine, **IoCompleteRequest**, to complete the I/O operation. I/O completion, in general, consists of the following operations:

o - Unlock any buffers that were locked down for DMA I/O.

o - Copy any buffered data from system buffers into the user's buffer.

o - Copy the status and state information from the IRP into the user's I/O status block.

o - If an event was specified, set it to the Signaled state and dereference it.

o - If no event was specified, set the file object to the Signaled state.

o - Dereference the file object.

o - If an APC was requested, queue it to the target process.

Copying the data and the I/O status block information must be performed in the context of the user's process so that its address space is accessible. Of course, signaling the event cannot occur until these two operations have completed since doing so causes a race condition.

The most important operation to complete as soon as possible is to unlock any buffers that have been locked into memory. Therefore, the first operation that the completion routine performs is to unlock the caller's buffers. This is done by invoking a memory management routine that unlocks the pages.

Once any buffers that need to be unlocked have been unlocked the completion routine queues a special kernel mode APC to the target process. When this APC executes, it finishes the tasks to be performed. It copies any buffered data that must be copied and copies the I/O status information into the requestor's I/O status block. It also sets either the file object or the event to the Signaled state and dereferences them, and queues the caller's APC, if one was specified. If the caller did not specify an APC then the I/O request packet is deallocated.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. Driver Model Description

Device drivers in **Windows NT** are loaded either at system initialization or dynamically using a special device driver loader program. This loader is executed from a directory that can be protected from non-privileged but malicious users who might try to load bogus code into the privileged part of the system, thereby compromising the system. The device driver image files themselves are also loaded, by default, from a protected directory on the boot device so that non-privileged users cannot overwrite them.

The image file format for a device driver is no different than other normal programs in the system. It is an executable program with a transfer address.

The components of a driver that the system is most interested in are as follows:

o - The *initialization routine*. This routine is invoked once when the driver is loaded. It is responsible for any initialization that the driver must perform including its own data, the device, the controller, etc.

This routine creates the objects (see the discussion below) that the driver needs in order to be used by other drivers in the system, by user application programs, etc.

This routine is specified as the transfer address of the device driver.

This routine is also responsible for filling in the addresses of the other routines in the driver object. This allows the I/O system to locate the various entry points for the driver.

o - The *major function routines*. These routines specify the entry points in the driver for each of the major function codes that can be specified in an I/O request packet. They are invoked when the driver is called with an IRP. It is their responsibility to perform any parameter checking, etc. If the state of the packet is acceptable, then the routine starts the I/O operation.

o - The *start I/O routine*. This routine is invoked to actually start the I/O operation on a device. It is invoked either because the device is not busy and a request is ready to be performed, or because the current request is finished and now the next request can be started. Its responsibility is to actually start the request on the device.

o - The *unload routine*. This routine is responsible for cleaning up any data structures that the driver has, deallocating any pool, and closing any objects

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

that it has opened. The system then frees the driver's code and data space, and marks it as gone from the system.

o - The *cancel routines*. These routines are invoked when a packet is to be canceled, and the packet is in such a state that simply setting its cancel flag will not cause the packet to be examined by the device driver. There is a cancel entry point that cooresponds to each state that a packet can be in. When a packet is marked for cancellation, the appropriate cancel routine is invoked to cancel the request.

As can be seen from this initial overview, device drivers do not really have a full context in the sense that a thread has a context. That is, they do not have their own address space or set of registers. A driver's context is the objects that it owns and the IRPs that it has access to via its device queue.

Device drivers execute in one of four different contexts depending on what part of the driver is executing and why. These contexts are as follows:

1. In the context of the user client thread. User threads request I/O by invoking the user APIs described in the *Windows NT I/O System Specification*. These routines validate the I/O operation, including probing the device-independent parameters and copying them into the IRP. The routines then invoke the device driver at its various function entry points to check the device-dependent parameters and to start the packet. If the device is already busy performing some operation, then the request may simply be queued to the driver.

The driver receives the request from the system in the form of an IRP. This packet describes the user's parameters. The driver may then use the I/O subroutines at this point to aid in getting the operation started. These routines are discussed in a later section of this document.

2. In the context of the ISR. When a device interrupts the processor, the processor executes a system routine that invokes the driver's interrupt service routine. In some machine architectures this may happen directly. In others, it may take place through a common interrupt dispatcher that gains control and then passes control to the appropriate device driver(s). The kernel sets up the necessary data structures for the given architecture so the correct control flow takes place.

The ISR must synchronize itself with the device driver start I/O routine. This is done through the use of the interrupt object and a routine, provided by the kernel, **KeSynchronizeExecution**. This synchronization object is necessary

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

so that the driver's start I/O and interrupt service routines do not attempt to access the device at the same time.

3. In the context of the DPC interrupt. Once an ISR has completed the absolutely minimal amount of work required to satisfy the device interrupt, it requests that it be able to perform the remainder of the work needed to process the interrupt at a lower IRQL. This is done by requesting that it be invoked at its DPC routine at DISPATCH_LEVEL.

The kernel's DISPATCH_LEVEL interrupt service routine scans the queue of DPCs to be executed and invokes them in order. The driver's DPC routine should perform the majority of its work servicing the device in this routine. It should deal with checking for errors, making error log entries if appropriate, setting the device to a known state if needed, completing any handshaking required by the device, etc.

This routine is also responsible for starting the next I/O operation on the device if there are any in the device queue. This can be done by invoking the **IoStartNextPacket** function. This function checks, in a synchronized manner, for another IRP in the device queue. If one is found, the routine dequeues it and returns the address of the packet. The device driver can then perform whatever actions are necessary to get the operation started on the device.

The DPC routine is also responsible for completing the current I/O request. This is done by invoking the **IoCompleteRequest** function. This function actually completes the request by copying data, unlocking buffers, setting events, etc.

The DPC routine is probably the most important and certainly the most complicated part of a simple device driver. It is here where most of the time in the driver will be spent.

4. In the context of the "driver process". Some drivers may actually have a full driver process associated with them. Drivers that have associated processes, as will be seen more clearly later, actually have two parts: 1) The standard driver part described above, and 2) a process that has one or more threads, each complete with a virtual address space, general register set, etc. The **Windows NT** file system and network service drivers are built using this model.

Drivers deal primarily with several different system objects that are specific to drivers. These objects are as follows:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - **Driver object.** Driver objects are created by the driver loader when the driver is initially loaded into the system. They are used by the system to determine where the entry points are for the driver, as well as for locating all of the controller and device objects that the driver is servicing. It also keeps track of where the code for the driver is loaded, its size, etc.

Device driver writers themselves do not generally manipulate the driver object itself. However, the object is used indirectly to locate entry points, etc., when calling between drivers or when the I/O system is calling the driver.

o - **Device object.** A device object is created by a driver for each device or device partition that the driver is to service. The collection of all of these objects, therefore, describes all of the devices in the system. Device objects contain information to allow the I/O system to manage I/O operations on the device, device characteristics, queue headers, etc. They also contain a device-specific area that the device driver can define. This allows the driver to maintain device-dependent context besides the IRPs with which it deals.

o - **Device queue.** A device queue is a kernel-provided object that allows the driver to synchronize a list of operations that need to be performed. Briefly, it provides the driver with a queue of IRPs to be executed, queue and dequeue routines, and a busy flag. Access to these items is synchronized by the kernel.

Device drivers do not normally manipulate the device queue directly; rather, they use the I/O system-provided routines to perform this work for them.

o - **Interrupt object.** Interrupt objects are created by the driver when associating itself with an interrupt vector, so that its interrupt service routine can be executed when the device interrupts. This is known as **connecting** to an interrupt. Interrupt objects, which are provided by the kernel, also provide the driver with a synchronization mechanism so that the driver can synchronize the execution of various critical sections of code with the driver's interrupt service routine.

o - **Controller object.** When the driver's initialization routine is invoked, it can create a controller object to represent the physical controller for its devices. Controller objects are created by device drivers to allow synchronization of requests bound for devices through the controller to which the devices are physically attached.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - **Adapter object.** Adapter objects are created by the I/O system when the system is initialized. An adapter object represents the system's mapping hardware. This object allows mapping registers to be allocated and set up so that scatter/gather operations can be performed to and from DMA devices. Device drivers use the adapter object in I/O system calls to allocate and initialize these mapping registers. In some systems, a device driver, via its device object, may be placed into a queue awaiting enough contiguous mapping registers to perform its transfer.

This object is also used to represent the channels on buses on some systems. The I/O system uses the adapter object to synchronize access to the channels by queueing the requesting device object to the adapter object.

Windows NT takes a "layered" approach to its driver design. That is, one driver may be layered on top of another device driver. This allows functionality to be added to the I/O system in such a way that device drivers themselves can be smaller and common code can be used for various types of devices.

For example, a device driver writer might want to implement a SCSI driver in one of two ways:

o - *The layered approach.* Using this approach, the writer would write a "port" driver and a "class" driver. The port driver would handle managing the controller itself and determining which device could be serviced based on requests being sent to it. The requests, of course, would be given to it in the form of an IRP, as for any driver.

The class driver would handle the devices themselves. For example, one class driver might drive the hard disks that are attached to the SCSI controller. Another class driver might handle a floppy disk, or a tape drive.

This approach is used in **Windows NT** to implement some of its drivers, including such drivers as the various file system drivers and network drivers. Requests, for example, can be given to a file system driver that modifies the request, and then gives it to the device driver itself. This is analogous to having the file system driver be a class driver and the device driver being a port driver.

o - *The controller approach.* This approach allows the driver writer to use the controller object to provide synchronization rather than have a separate device object and device driver to do this. For many simple controllers and devices, this approach works best.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

> When a device driver which is using this approach wishes to start an I/O operation, it must allocate the controller object, start the I/O operation and then either deallocate the controller object if it is no longer needed (for example, a disk seek), or keep the object allocated until the operation is complete. When the operation is complete, such a driver deallocates the controller so that the next operation can be started.

Which of these two approaches a device driver developer uses is discretionary. Some more complex operations, such as a file system, clearly require something more than a controller object. A SCSI driver could be done using either approach with a fair amount of efficiency.

3.1. Time-Out Handling

Some device drivers must be concerned with operations timing out, whether because the device itself has timed out or because the user has asked that an operation be completed in a certain amount of time and it did not finish within the allocated time.

The **Windows NT** kernel provides driver writers with a means to timing operations. It does this through an object termed a *timer object*. A timer object can be initialized by calling the **KeInitializeTimer** kernel function. This routine initializes a timer object that can then be set to an expiration time using the **KeSetTimer** function. Expiration times may either be expressed in an absolute or a delta time format. The set timer routine also optionally accepts a *Deferred Procedure Call* **(DPC)** object address. This allows the specified routine to be executed when the timer expires. Finally, the kernel provides the **KeCancelTimer** function that allows a timer to be canceled.

The I/O system provides a simple interface to the structures and routines provided by the kernel. The interface allows the device driver to specify the address of a routine to be executed once every second. This routine is provided with a context pointer that it uses to access its own data structures. One of these structures can contain a counter that is decremented each time the routine is invoked. If the counter is ever decremented to zero, then the operation is considered to have timed out. Whenever a new operation is started on the device, the counter can be set to the number of seconds that the operation has before it should time-out. If no operation is in progress, then the counter can be set to some predetermined value, such as minus one. This allows the timer routine to determine that no operation is being performed and it should not modify the counter.

The I/O system provides a device driver object with a built-in timer with the above functionality. The routines used to support this interface are as follows:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - The **IoInitializeTimer** routine. This function accepts the address of the routine that is to be executed and the address of a device object. The **IoInitializeTimer** routine initializes the device object's timer for use when it is started.

o - The **IoStartTimer** routine. This routine starts the timer; that is, it causes the routine specified in the above call to be invoked once every second. The driver-specified routine will be invoked with the context parameter that it specified in the call to the **IoInitializeTimer** routine.

o - The **IoStopTimer** routine. This routine stops the timer; that is, it will stop invoking the driver's timer routine once every second. The **IoStopTimer** routine is generally only invoked if the driver is being unloaded from the system.

3.2. Power Recovery

Drivers must also deal with the possibility of power failures and recoveries. When a power failure interrupt occurs, the kernel saves enough state so that when the power is restored the system can be restarted from wherever it was executing when the power failed. Special considerations must be made for device drivers because of the hardware associated with them.

The concerns here are three-fold:

1. Device drivers must be given the chance to reinitialize the devices that they are servicing. This allows the devices to be put back into a well known state so that operations may once again be requested. Controllers also need to be reinitialized and placed into well known states.

2. Device drivers must deal with operations that were currently in progress when power was lost. For most devices, this probably means restarting the I/O request once the device has been reinitialized.

3. Device drivers must deal with synchronization of power recovery interrupts and of executing code that touches the device itself. That is, the device driver should ensure that once it is ready to actually tell the device to begin a transfer, it does this atomically. This means locking out power failure interrupts for a short period of time. If a power failure interrupt occurs between the time that the device driver synchronized itself with its start I/O routine (at device interrupt request level - DIRQL) and the time that it was ready to begin writing device registers, then it would need to abort the operation and not tell the device to perform it. This is because if the power

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

failed while writing device registers and then came back up, the device could perform an erroneous operation if the transfer were restarted.

Likewise, the device driver needs to synchronize with power failures occurring while the device driver is in its device interrupt service routine or in a DPC routine that touches the device registers. In both of these cases the driver needs to determine whether or not a power failure has occurred so that it does not inadvertently "complete" an I/O request based on unpredictable device register contents.

For these reasons, the **Windows NT** kernel provides device drivers with two different objects that can be used to deal with power failures. These two objects are as follows:

o - **Power Notify Object.** This object allows the driver to establish a DPC that is to be invoked whenever the system's power is restored. This allows the device driver to be asynchronously notified when the power has been restored by having the system call one of the driver's routines. This routine should deal with the first two concerns above. It should handle the device/controller reinitialization and it should restart or fail the "current" request.

The kernel provides a routine to initialize a power notification object **(KeInitializePowerNotify)**, a routine to insert the power notification object into a queue so that the associated routine can be invoked **(KeInsertQueuePowerNotify)**, and a routine to remove the object from a queue **(KeRemovePowerNotify)**.

o - **Power Status Object.** This object and its associated functions are provided to deal with the third concern. This object provides the driver with a flag that can be tested to determine whether or not the power has failed. The flag is set to TRUE if the power has failed and recovered; otherwise it is set to FALSE.

The kernel provides routines to initialize a power status object **(KeInitializePowerStatus)**, a routine to register the power status object **(KeInsertPowerStatus)**, and a routine to unregister the power status object **(KeRemovePowerStatus)**.

The model for recovering from a power fail interrupt is to use the power notification object and two power status objects. The two status objects should be used as follows:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. The first power status object should be used to indicate that the power has failed. This power status object will hereafter be termed the *power failed* object.

2. The second power status object should be used to indicate whether or not the power notification routine has finished its processing of the power recovery. This power status object will hereafter be termed the *power recovery routine not done* object.

When the power fails, both of the power status objects will be set to TRUE. This indicates to the driver that the power has failed and the power recovery routine has not finished its processing. At well-defined points in the driver, both of these status objects should be checked to determined whether either is TRUE. If **either** of the objects is TRUE, then the driver should not continue processing of requests. That is, the driver should not allow new requests to be processed. For example, the driver's start I/O routine, interrupt service routine, and dispatch routines should check to ensure that no power failure processing is occurring in the driver. This is done by raising the current IRQL to POWER_LEVEL and check the logical OR of the two status objects. If either is TRUE, then processing should not continue for the device.

The power notification routine performs four functions:

1. The first step is to set the power failed object to FALSE. This indicates that the driver is aware that the power has failed and recovered.

2. The second step is to perform any device initialization that is necessary to allow normal operations to continue on the device.

3. The third step is to set the power recovery routine not done object to FALSE. This indicates that all power recovery for the device has been completed and operations can be continued normally.

4. The final step is to restart the current packet for the device, if one was active. The *CurrentIrp* field of the device object is a pointer to the packet that was currently being processed by the driver.

3.3. Canceling I/O

I/O operations can be canceled in one of two ways:

1. All of the I/O for a thread can be canceled by invoking the **IoCancelThreadIo** subroutine. This routine scans the IRP list for a thread and cancels the I/O operation represented by each IRP in the list. This

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

function is useful in thread rundown and termination. It is invoked from kernel mode in the executive and is not available to general users.

2. All pending operations issued by the calling thread for a file handle can be canceled by invoking the **NtCancelIoFile** service. This service performs the same operation as above except that only the thread I/O for the file associated with the specified file handle is canceled. All other I/O is unaffected.

\\ *This functionality in the I/O system is currently being redesigned. This design will be complete and implemented for the next version of this specification.* \\

3.4. Driver Layering

As mentioned earlier, drivers in the **Windows NT** I/O system may be layered. This allows one driver to communicate with another driver by simply calling it and passing it a pointer to an IRP. This feature allows the system programmer to add functionality to the system in many broadly or narrowly focused ways.

Consider, for example, a file system and a simple disk driver. The file system is represented by a file system driver and the disk driver is represented by a device driver. Both have device objects which are named so that they can be referenced from outside themselves. The file system sees the disk that the device driver presents as a stream of 512-byte blocks that are referred to by numbers, 0 through *n*.

In this simple case, when the user attempts to read part of a file from the file system, the file system accepts the request and changes it into a request to the disk driver for whatever blocks on the disk that need to be read.

IRPs are set up to handle exactly this type of layering. Each packet contains a fixed portion that contains information about the original request, which thread the request belongs to, event pointers, etc. The remainder of the packet is an array of structures that is treated as a "stack". That is, each layer in the chain of drivers owns one of the structures in the array. In our example then, the disk driver would own "stack location" number one and the file system would own "stack location" number two.

These stack locations contain the following information:

o - A major and minor function code. These function codes are used to tell the driver what type of operation to perform and how to interpret the remainder of the information in the structure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - Parameters. These are the parameters that the driver uses, based on the function codes, to perform the specified operation. Up to four separate 32-bit longwords of arguments can be passed, as well as an 8-bit flags byte.

o - I/O system information. There is also some information that is maintained by the I/O system that describes information about the driver.

As each layered driver is invoked down the chain, the "current stack location" is adjusted so that it points to the proper structure for the next driver. As each driver is invoked, it may decide that it would like to be invoked on certain conditions once the I/O operation is complete. For example, the file system performing the read on behalf of the user may wish to know if an error occurred during the read operation so that it knows that there is a potentially bad block in the file.

A driver can register a "completion routine" through the use of the **IoSetCompletionRoutine** function. This function accepts the following parameters:

o - A pointer to the IRP that is currently being worked on.

o - The address of a routine to be invoked when all of the layers below the current driver have completed the request.

o - A context parameter which can be used by the driver for whatever extra information it requires.

o - Three flags which indicate whether the completion routine is to be invoked if the operation is successful, if it completes with an error, or if the operation is being canceled.

Once the drivers below the current driver, in our example the disk driver, completes the operation, the file system's completion routine is invoked, if one was specified. It is invoked with the following parameters:

o - A pointer to the driver's device object, which contains the information about the volume or device on which the I/O operation is taking place.

o - A pointer to the IRP. The current stack location in the packet is the one that belongs to the current driver.

o - The context pointer that was passed in the above function call.

If the driver does not wish to be invoked at all, then it need not call this I/O function.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Layering of drivers also yields the ability for the system programmer to insert layers of drivers between each other to provide different functionality to the system. In the previous scenario, for example, a driver can be inserted between the file system and the disk driver which interfaces to many disks rather than to a single disk. It could then present all of the disks to the file system as a single large disk with lots of blocks, still referenced as 0 to *n*. Note that since this driver is completely file-system-independent, it could be used with any file system type. So, this single driver could be used as an enhancement that provided tightly coupled volume sets to several different file systems without ever touching the sources of the file system driver.

4. File System Description

File systems in **Windows NT** are executed as layered drivers with the special exception that they have an actual process associated with them. These processes are referred to as *file system processes*, or **FSPs**. Having these processes gives file systems the ability to not only be able to execute in the context of a requesting thread, or to execute without concern as to what thread they are executing in, but it also gives them the ability to perform such operations as waiting on an object without causing the client thread to wait.

File system processes are like most other user processes in the system except that they run entirely in kernel mode. These types of processes are referred to as *kernel processes* in **Windows NT**. They can make direct calls to I/O functions and to their own file system drivers **(FSDs)**. They can also allocate system pool, etc.

The communication between an FSD and an FSP occurs through a *communication region* that is set up by the FSD during its initialization. This communication region contains a queue, a spin lock to protect the queue, and an event. The spin lock and the queue are used in conjuction to provide the functionality of an interlocked queue. The event, generally autoclearing, is used to notify the FSP when an item has been placed into the queue.

The queue provides a location for the FSD to place IRPs that need more processing than the FSD itself is capable of performing in its limited context. Other shared data used between the FSD and the FSP may be contained in this region. How it is structured is up to the file system writer. The communication region need not be statically allocated within the image. It can be allocated as part of the device extension when the device object is created.

To synchronize access to the queue, the FSD performs the following steps:

o - Inserts the IRP into the queue using the **ExInterlockedInsertTailList** function.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - Sets the event to the Signaled state.

The FSP synchronizes access to the queue by performing the following steps:

o - Wait for the event to be set to the Signaled state and reset it if it is not an autoclearing type event.

o - Remove the entry from the head of the queue using the **ExInterlockedRemoveHeadList** function.

o - Loop, removing entries from the head of the queue until there are no more entries remaining.

It should be noted that it is possible for the FSP to be awakened for an entry in the queue that it has already removed. Therefore, it must be able to handle the situation where there are no entries in the queue to be processed.

Using this queueing method allows the FSD to pass information to the FSP. FSPs, on the other hand, can communicate with FSDs by simply invoking routines in the FSD. Because the FSD is thread-context-independent, this is basically no different than the FSD being invoked by the system to perform a request on behalf of a user thread.

The two parts of the file system, the FSD and the FSP, both reside in the same image file on the disk. When this image is loaded, the loader maps the entire image into system space. The FSD, in its initialization, sets up the data structures and communication region that it will use to communicate with the FSP and creates the FSP.

Both parts of the file system share data in the communication region described above. Also in the communication region, or perhaps pointed to by it, are the data structures that allow the FSD to determine which files have been opened, what data is in the cache, etc.

Among this shared data is also a description of how blocks in files are mapped to blocks on the disk itself. The data structure supported by the **Windows NT** I/O system to describe this is termed a *Map Control Block* **(MCB)**. Not all file systems need to use MCBs, as the on-disk structure may already be well structured enough to describe the mapped blocks efficiently.

An MCB describes the extents of a file in an array of structures composed of the following two longwords of information:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - Virtual Block Number (VBN). The starting block within the file that the extent represents.

o - Logical Block Number (LBN). The starting block on the disk where the extent resides.

A header on the data structure describes the maximum number of entries in the MCB, the current number of valid entries in the MCB, and the block number of the extent in the file which represents the end of the file. It should be noted that this structure also works well for describing sparse files and can be searched with a binary search.

If the FSD attempts to translate a file block number into a disk block number and the entry required to perform the mapping is not currently in the MCB, then the FSD must have the FSP perform a *window turn* operation on the MCB. That is, the FSP must adjust the entries in the MCB so that the file extent to be mapped is actually described in the MCB entries. The FSP is needed for this operation because the amount of shared data necessary to allow the FSD to perform this operation would be too large for some file systems. The FSP is also needed so that no implicit wait operations are required in the context of the client thread.

\\ *The set of interface routines for providing this functionality has yet to be designed. Since it is required fairly soon in the implementation of the project, it will be provided in the next revision of this document.* \\

4.1. IFS Design

Multiple file systems may be active in **Windows NT** at any given time. These file systems might be servicing multiple devices or they might be servicing different partitions on the same device.

When a file system is loaded, it registers itself as a resident file system that is eligible to be invoked when a volume is to be "mounted". This is accomplished by creating a device object whose type is *FILE_DEVICE_FILE_SYSTEM* and invoking the **IoRegisterFileSystem** function with the returned device object. This function inserts the device object onto the list of active file systems.

The loader sees a file system driver the same as it sees a device driver, as a simple program. The transfer address of a file system driver is its initialization routine.

The entry points that the I/O system is interested in for a file system driver are as follows:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - *Initialization routine*. This FSD routine initializes the file system's data structures, allocates and initializes the communication region, creates a device object for the file system, creates the FSP with at least one thread executing, and registers itself as a file system.

When the FSP is created, it is passed the address of the device object. Its responsibility is to initialize itself and set up any other structures that may be used between the two components, if any.

The FSD routine must also fill in the driver object routine address fields with its entry points so its routines can be located.

o - *Major function routines*. These routines correspond to the major function codes in an IRP. They are selectively given control when the IRP is handed to the driver. The file system supplies a routine entry point for each of the major functions that it implements. All others are defaulted by the I/O system to a routine that returns an error code indicating that the request is not implemented by the driver.

The major function routines are responsible for validating the parameters in the I/O request packet and determining what should be done to perform the request. This might mean copying data from the file system's cache, or performing a window turn on the file, etc. If more processing is required, then the FSD might give the packet to its FSP.

o - *Unload routine*. This routine is invoked by the system when the file system is being unloaded. Its responsibilities are to let the FSP know that it should clean up and exit, and then to clean up any data structures the FSD has, deallocate its communication region, unregister itself as an active file system, and close its objects. The system then frees the file system's code and data space and marks it as gone from the system.

o - The *cancel routines*. These routines are invoked when a packet is to be canceled and the packet is in a state such that simply setting its cancel flag will not cause the packet to be examined by the driver. There is a cancel entry point that cooresponds to each state that a packet can be in. When a packet is marked for cancellation, the appropriate cancel routine is invoked to cancel the request.

Once the media in a device has been recognized by an active file system, a record is kept of which file system owns the media. When a request for that device is issued,

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

the system first gives the request to the file system for processing. This file system can then give the request to the device driver, if required.

The structure that is used to keep track of which file system is currently servicing a device is called a *volume parameter block* **(VPB)**. This structure contains the volume label and serial number that the file system returned when the volume was mounted. If a request is made for a volume that is no longer in the drive, then the system hard error routine will be invoked to request that the user place the appropriate media back into the drive so the operation can continue.

A file system may request that it be permitted to perform post-processing work in the FSD after the device driver has completed the I/O operation. This is accomplished by using the its stack location in the I/O request packet, as described in an earlier section of this document on *Driver Layering*.

4.2. Mapped File I/O

> \\ *This section is somewhat out-of-date. The current revision, 1.1, has cleaned up the obsolete statements made in previous versions which were very old. As the design evolves, this section will be updated to reflect the latest thinking on this area.* \\

It is possible in **Windows NT** for users to perform I/O to a file by simply reading and writing memory. A set of library routines are provided to support the interfaces that set this up for the caller. Mapped I/O can be set up by calling the following functions:

o - **NtCreateFile(**FH, ...**);**

o - **NtCreateSection(**SH, ..., FH**);**

o - **NtMapViewOfSection(**SH, ...**);**

Once the file has been created or opened (via **NtCreateFile** or **NtOpenFile**), a section is created to the file. The user then creates the section for the file by invoking the **NtCreateSection** service. When this service is invoked, it must be invoked with the maximum size that the file will ever grow to before the section is closed.

> \\ *This will need to be fixed. The section must be able to grow if people are to realistically use sections for files. An editor, for example, cannot determine how large a file will grow when the user initially opens it.* \\

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The section is then mapped using the **NtMapViewOfSection** service. This service allows the user access to the file actually backing up the section. Reads and writes to the file occur implicitly by simply reading or writing the memory that is mapped by the view to the section.

Likewise, the user may still use the file handle returned from the **NtCreateFile** (or **NtOpenFile**) service to perform other operations on the file such as reading and writing it. In this case, the file system simply maps a view to the section for the file into the system virtual address space and then reads or writes the data from/to its memory.

There is a design note, **Windows NT Mapped I/O Design Note**, that further discusses this mapped I/O model and other models that were considered. See this document if there are questions regarding mapped I/O.

4.3. File Caching

Caching in the file system is provided through the *Cache Manager*. This component of the system provides file system drivers with the ability to map files into the system virtual address space so that they can be managed by the file system through the use of the memory manager. More information on the Cache Manager can be found in the *Windows NT Cache Design Note*.

The amount of I/O system-specific support required to provide this functionality to file systems is minimal. In fact, the only feature required is the addition of *stream file objects*. Stream file objects allow a file system to represent various parts of an on-disk structure that are not in proper files as files in the I/O system. That is, a file is cached through the memory manager by creating a section that is backed by the file itself. Once the section is created, the file system simply needs to map a view to the part of the file that it requires access to, and then touch that virtual address space. This causes a pagefault to occur for the mapped file. The memory manager then performs a read or write operation to the file based on the type of access the file system made to the memory location.

In order to allow EAs or ACLs for files to be cached, the file system needs a way to describe the portions of the disk structure that contain this data. Since this data may not be in the file itself, a virtual file concept is used to describe and map the data as if it were part of a normal file. Stream file objects are used to represent these abstractions.

A stream file object can be created to represent any part of the on-disk structure that the file system needs access to through the cache. Creating a stream file object is accomplished by calling the **IoCreateStreamFile** function. This function creates

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

a stream file and returns a pointer to it. The pointer can then be used by the file system to create a memory section as it would for a normal file.

4.4. Splitting Transfers

Sometimes a user requests a file transfer that spans multiple extents in the file. The FSD can determine this by attempting to map the offset in the file via the MCB for the file. When this occurs, the file system needs some way of splitting the request into several different transfers. This section describes the three options that a file system writer has available.

All of the following models begin with a request (IRP), being passed to the appropriate major function routine in the FSD. Once this routine recognizes that the I/O request requires multiple transfer requests to the device, it uses one of the following models. Each model is described using a scenario where the operation is a read and there is no caching involved in the transaction to keep it as simple as possible. Other scenarios may be extrapolated from the models without much effort, so they are not exhaustively elaborated here.

4.4.1. FSP Model

The FSP model uses the FSP to split a transfer into multiple device requests. The FSP splits the transfer by performing the following steps:

o - The FSD places the IRP into the interlocked work queue of the shared communication region and sets the event associated with the queue to the Signaled state for the FSP.

o - The FSP's wait on the event is now satisfied so it removes packets from the interlocked work queue.

o - It determines the number of individual requests that are required to satisfy the original request and fills in a counter location in the original IRP with that number.

o - For each individual contiguous transfer from the device, the FSP performs the following steps:

- Allocates an IRP,

- Fills in the IRP with the information to describe the partial request,

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Marks the IRP as an associated IRP using the **IoMakeAssociatedIrp** function,

- Fills in the starting LBN for this extent,

- Fills in the length of this extent,

- Allocates and initializes an MDL for the next part of the requestor's buffer using the **IoAllocateMdl** and **IoBuildPartialMdl** functions.

- Calls the appropriate device driver to perform the function using the **IoCallDriver** function.

Each time a requested transfer completes, the device driver completes the IRP which causes the system's I/O completion routine to be invoked. This routine sees that the IRP is an associated IRP and decrements the counter in the original IRP. The final I/O status is also formed from each I/O request packet.

Once all of the IRPs are complete, denoted by the original counter going to zero, the completion code "completes" the original request. Notice that until the original request is actually completed, the completion code need not context switch back to the original requestor. Likewise, no context switches to the FSP are required.

This model causes the various pieces of the transfer to be processed in a quasi-parallel fashion because the FSP can queue multiple transfer requests to the device driver. This allows the device driver to process the packets more quickly.

A variation on this model is to use an algorithm where the FSP simply queues one request for each extent, one at a time to the driver. The FSP then requests that it be notified explicitly when the request completes. This allows the FSP to keep state information about the original IRP rather than setting the associated IRP pointer in the new IRPs that it queued.

This variation would work best if the FSP used APC routines in its own threads to synchronize the I/O operations, thereby using fewer system resources by not dedicating a thread to each request.

4.4.2. FSD Parallel Model

This model is similar to the FSP model except that, rather than have the FSD context switch to the FSP to let it allocate the IRPs and queue them to the device driver, the FSD would perform all of the steps itself. It would perform the allocation and initialization of the IRPs while still executing in the context of the user's thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

While this model works well for some small number of situations, it further complicates the FSD code and causes more state information to be kept in the FSD.

This model is still considered quasi-parallel however, because it allows the file system to queue multiple requests to the device driver without having to wait for any one operation to complete.

4.4.3. FSD Serial Model

This model is similar to the FSD model except that rather than have the FSD allocate, initialize, and queue multiple request packets, it simply allows the FSD to reuse the original request packet over and over again until the entire transfer request has been satisfied.

This is done by storing the original request information in the packet as context information and then changing the MDL, length, and starting LBNs according to the next extent of the file.

The stack location completion routine for the FSD in the IRP can be filled in so that it gets a chance to execute when the request has been completed by the device driver. When this routine is invoked, the FSD sees that this is a multiple-transfer IRP, fills in the next request information, and gives it back to the device driver again. This continues until the request has been completed.

Once the entire original request has been satisfied, the FSD calls the normal completion code to complete the operation.

This model has the advantage that it requires the least amount of processing overhead. No IRPs are being allocated and initialized from scratch. On the other hand it is potentially slower because it is serialized. If the I/O were being done to multiple spindles, such as a striped file, this method could actually lower the throughput of the I/O system even though the amount of processing overhead is less.

4.5. Mounting and Volume Verification

Windows NT supports multiple file systems running at the same time. This imposes three basic requirements on **Windows NT**:

> o - Automatic media recognition. When the media in a device is first accessed, the system must be able to determine which file system is supposed to deal with that media. That is, when a user makes a request such as an

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

open, read, write, etc., operation on a device that needs the support of a file system, the system must be able to determine which file system is to handle the request.

o - Supporting removable media. Because the system supports removable media, such as floppy disks, it must also be able to dynamically change its idea about which file system is currently supporting the device.

For example, if the user switches the media in a floppy drive, the system must be able to determine whether the on-disk file structure on the new media is the same as on the previous media. If it is not, then **Windows NT** must determine which file system understands the new media and associate the new file system with that media.

o - Supporting multiple partitions. Finally, the system must be able to support multiple partitions on hard drives. This means that it must also be able to support cases when each partition is a different on-disk structure.

These three requirements mean that when a device is first accessed or when a device is being accessed whose media may have changed, the system must locate the appropriate file system for the device. This is done by keeping track of the media that the system thinks is currently in the drive. The structure used to do this is a called a *volume parameter block* **(VPB)**. This structure keeps the volume label and the volume serial number of the media. It also keeps track of the file system's device object associated with the volume.

Initially, the VPB associated with a drive is blank; that is, it has no name and a zero (an invalid value) for its serial number. When a volume is first touched, the system realizes that the information in the VPB is invalid and begins invoking registered file systems to determine which one recognizes the structure on the media. It does this by passing the FSD an IRP with a function of "mount".

The file system generally performs the following steps to determine whether or not it should successfully complete the mount request, thereby "owning" the volume on the device:

o - It begins by giving the IRP to a thread in the FSP which can perform I/O to the device.

o - The FSP reads whatever blocks are required to determine whether or not it recognizes the on-disk structure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - It then either completes the request with an error if the structure was unrecognizable, or it continues.

o - If the FSP continues, the FSP creates a device object which will be used to hold its context for this volume and initializes the file system-specific data structures in the newly created device object.

o - The FSP then fills in the device object field of the VPB with the address of the newly created device object.

o - The FSP then creates a thread to handle this volume, passing it the address of the newly created device object.

o - Finally, the FSP completes the original mount I/O request packet with a success status, indicating that the media format was recognized.

If the file system does not recognize the structure on the media as its own, then the system then continues through the list of registered FSD's looking for a file system that recognizes the media. If none is found, then the RAW file system, the last in the list, takes over the media and treats it as non-formatted media. This file system provides a way of reading and writing the device. It recognizes all media.

Upon completion of a mount, the I/O system mount code checks the status of the operation, and if it is successful, it associates the VPB with the device. This is accomplished by having a pointer in the device object that contains the address of the VPB. The VPB currently being pointed to represents the system's idea of the file system structure in the drive.

5. Network Service Description

\\ *This section is currently under design review in the Windows NT Network Group. The current thinking is that there will be four to five layers: Redirectors, Transports, Networks, Datalinks, and Physical links. Each layer will be able to be invoked from the layer above it or directly by the user with no differences. There can be any number of drivers at any layer. Further, some layers may be subsumed by another layer.*

This section will also discuss how names are resolved.

More information on Networks will be available in the next release of this specification. \\

6. I/O Completion

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

I/O completion consists of a routine which drivers invoke and a special kernel APC routine internal to the I/O system. This section presents how each is used and the functions that each performs.

Once a driver has finished all of its processing for an I/O request, it invokes the **IoCompleteRequest** function to actually complete the I/O request. The device driver normally invokes this routine when it is at DISPATCH_LEVEL in its DPC queue routine.

IoCompleteRequest begins by checking the IRP stack to determine if there are any other drivers that need to be notified that the I/O completed. If there are, then it invokes each driver's completion routine if one was specified in the stack. The function determines whether or not there are more drivers that need to be notified by comparing the count of stack entries to the current entry number. If they are different, then there is another driver to be notified.

The routine determines whether or not to invoke the driver's completion routine based on whether the cancel I/O flag is set, the success or failure of the status code in the I/O status block of the IRP, and whether the routine is to be called for each case depending on what was specified in the call to **IoSetCompletionRoutine**.

The algorithm that I/O completion uses to determine whether the routine should be invoked is as follows:

```
if (Irp.CancelIo AND InvokeOnCancel)
                    OR
   ((NT_SUCCESS(Irp.IoStatus.Status)) AND InvokeOnSuccess)
                    OR
   ((!NT_SUCCESS(Irp.IoStatus.Status)) AND InvokeOnError)
        /* invoke the completion routine */
```

A driver may not wish to be invoked for any of the above reasons. If this is the case, then it need not take any action since the I/O system guarantees that the flags are zeroed in the stack when the driver is initially invoked.

Once all of the device drivers have been invoked, the I/O completion routine checks to see if any pages were locked into memory for this operation. This is done by querying the *MdlAddress* field of the IRP. If any MDLs are associated with the IRP the I/O system walks the list and unlocks all of the pages associated with each of the buffers described by the MDLs.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Once any pages that were locked are unlocked a special kernel mode APC is initialized and queued to the target thread using the kernel APC interface routines. If the thread is in an appropriate state, then it will be scheduled to execute. The address of the routine to be executed as the special kernel APC is part of the I/O completion code.

Once the special kernel mode APC begins execution in the context of the target thread, it performs the following steps to complete the I/O request:

o - All buffered data, if any, is copied from the system space buffers into the user's buffers. If the system space buffers were temporary buffers allocated to transfer this information, then they are deallocated.

Buffered data here also refers to any data that is OUT in any API, with the exception of the I/O status block. That is, any data in the output buffer interface such as the **NtQueryDirectoryFile** service, for example, is copied into the user's buffers.

o - Any MDLs used to describe the user's buffers are deallocated.

o - The I/O status information in the IRP is copied to the user's I/O status block.

o - The file object is set to the Signaled state, if no event was specified, and it is dereferenced.

o - The specified event for this request, if any, is set to the Signaled state and dereferenced.

o - If no APC was requested by the I/O initiator, then the IRP is deallocated.

o - If an APC was requested, an APC object is initialized and queued to the thread. The IRP is deallocated once the APC has been removed from the thread queue by the kernel.

Some system features cause special considerations in how I/O completion works. IRPs that have been linked together by a file system, for example, are handled differently to gain better performance and because the file system isn't generally interested in the completion notification unless the operation fails. Many times it is only concerned when all of the operations have completed and sometimes even then it is not interested directly.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Paging I/O also requires special consideration during I/O completion. Such considerations include the following:

o - No pagefaults may occur on any path of pagefault code during I/O completion.

o - No special kernel mode APCs may be taken in the context of the target thread because this could cause another pagefault to occur. Moreover, APCs are blocked anyway. Therefore, the general mechanism for completion cannot be used.

o - Caching in the file system is disabled for paging I/O.

o - No APC is delivered to the target process to "complete" the paging I/O request. This can all be done without actually entering the thread's context. The pager's I/O status block is in non-paged memory in system space and will never incur a pagefault. It is always visible regardless of what thread context the system happens to be in when the I/O completion code is executing.

o - The pages that were being paged in do not need to be unlocked by the I/O system since they will be unlocked by the pager.

7. Error Logging and Handling

Windows NT provides both special error handling for users, and an error logging facility for drivers. This section explains how both work and how they should be used from the driver's point of view.

7.1. Error Logging Facility

Error logging in **Windows NT** is supported by the following components:

o - I/O support routines. These routines provide drivers with an interface to the error process. Communication with this process occurs through the use of datagrams sent to its port that contain the information that the driver would like to write to the error log file. These routines allow the driver to allocate a datagram, fill it in, and send it to the error log process's port.

o - The I/O system thread. This thread removes error log buffer entries from the pending queue and sends them to the error log process as datagrams. It then frees the entries back into the buffer pool. This thread uses the standard interlocked work queue and event method of synchronization used by FSD/FSP drivers.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - The error log process. This process maintains a port to which error log datagrams can be sent. It is the responsibility of this process to take the datagrams and write their contents to the error log file. This process is also responsible for maintaining the file itself. An old file can be opened, an old file can be closed and a new one created, or a new file may be created. Finally, it is the responsibility of this process to write time stamps to the file. These are written in such a way that if no actual error log entries are written between time stamps, then only one time stamp is entered in the file. This saves disk space by minimizing the amount of data actually written to the file.

o - The error format utility **(EFU)**. The EFU has the responsibility of reading error entries out of the error log file and displaying the contents in a form that is understandable by service personnel.

Because there are many different types of devices in the system and some will be supported by device drivers other than the standard drivers that are part of the **Windows NT** operating system, device driver writers can write error log buffer translation routines that the EFU can invoke when an entry is found for the specified device.

Each error log entry contains a header that specifies the type of entry being formatted. This header also contains a device type identifier field that is filled in by the driver through the I/O support routines. This is a unique name that is declared by the device driver. When the EFU discovers the entry, it invokes the entry in the DLL that corresponds to the entry to format it. All of the images for formatting the entries are contained in the *error* directory in the **Windows NT** directory tree. The names of the images are the same as the device type identifier field. That is, a ".DLL" is appended to the device type identifier field and that dynalink library is invoked to format the entry.

The EFU passes the DLL routine a pointer to the entry to be formatted as well as the address of a routine that can be invoked to output whatever information needs to be output for formatting purposes.

The name of the entry point in the DLL for the EFU to invoke is **FormatEntry**.

The I/O support routines related to error logging are as follows:

o - **IoAllocateErrorLogEntry** - This routine allocates an error log entry for the device driver. The header information is automatically filled in by the routine. The driver can then place the error information into the entry. If there are no available error log entries in the system, then a null pointer is returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - **IoWriteErrorLogEntry** - This routine sends the specified error log entry to the error log process to be written to the error log file.

\\ *There also needs to be an API like NtCloseErrorLogFile that users with the appropriate privilege can call to have the error log process close the current error log file and optionally open a new one.* \\

7.2. Error Ports

Users may wish to be notified when a device operation is going to fail because the wrong disk is in a drive or because of some other error that requires intervention on behalf of the user. The OS/2 subsystem also requires this functionality in order to emulate the "fail on error" flag.

Windows NT provides this type of functionality by giving the user the option of providing an "error port" on a create or open to a file or device. This port allows the I/O subsystem to RPC to the user when an error occurs to let him determine what he would like to do about it. That is, if the wrong volume is in a device, for example, an RPC message is sent to the user's port in order to determine what he'd like to do about the situation. The user may return one of the following:

o - Retry - Retry the operation. In this case, the caller presumably communicated the problem to the user and asked that the correct volume be placed in the device. The operation is simply retried by sending the IRP back through the system.

o - Abort - Abort the operation. In this case, the I/O request is simply aborted and the operation is completed with the appropriate error status.

The I/O system uses this common model with the system hard error thread. This thread has a globally known port that receives an RPC when an error such as the above is encountered. The hard error routine does exactly what was described above in order to have the situation resolved.

8. Terminal I/O Considerations

This section addresses those special requirements needed by terminal devices, especially dealing with unsolicited input and conventions used by subsystems to perform terminal input operations that they need to support.

8.1. Unsolicited Input

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The operating system emulation subsystems, the Session Manager (SM), and the Terminal Logon Process need to be able to be notified when an unsolicited input occurs on a terminal which, by definition, is not logged into the system. To do this, there is an **NtDeviceIoControlFile** service that allows a user with sufficient privilege to specify a port to be given a message when an unsolicited input occurs on a terminal.

The Terminal Logon Process can therefore open the terminal, issue this service and then close the terminal, and not keep any more state until something happens on the terminal.

When someone enters a termination or interrupt character on the terminal, the terminal driver will use the **IoSendMessage** routine to send a message to the specified port. This routine allocates and queues a kernel mode APC to the I/O completion thread. The routine specified as the APC will be a routine in the I/O system that translates the name of the port and sends a datagram message to it specifying the name of the terminal device on which the unsolicited input occurred. The type of the message sent is **PORT_TERMINAL_INPUT**.

The Terminal Logon Process can then open the terminal and begin performing the standard login sequence.

8.2. Subsystem Input

The model used for terminal input by subsystems that are emulating different APIs varies, but the following recommendations help system performance and, in some cases, provide the subsystem with an easier task:

o - Subsystems should attempt to perform all terminal input operations on buffers that are long enough to contain the average terminal line size.

o - Subsystems should avoid performing single character I/O to terminals or other devices when possible.

o - Subsystems may wish to have one dedicated thread that performs terminal input. This thread can process the data by placing it into internal queues and emulating the PM keyboard model of I/O. This allows the subsystem to have one interface to these types of devices.

o - Subsystems should make use of the device I/O control service to set the termination characters for a device to those characters generally recognized by the API they are trying to emulate. They should also include those characters that are considered out-of-band characters for their API. This

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

allows the subsystem to scan characters written to its buffer from the terminal driver and determine when these character have been seen.

9. I/O Data Structures and Objects

This section gives a brief overview of the data structures and objects used in the **Windows NT** I/O system. It describes what the data structures are, how they are used, and in some cases a description of the major fields in the data structures.

All of the data structures in this section contain a type field which is used by the I/O system for robustness. This allows the system to check to ensure that a pointer really points to the type of structure to which it is supposed to point.

9.1. I/O Request Packet Description

The I/O Request Packet (IRP) is the primary data structure used in the I/O system to pass information from system services to drivers, from drivers to other drivers, and from drivers back to the I/O system. IRPs are always allocated from some part of nonpaged memory since they are sometimes accessed at raised IRQL.

An IRP is allocated with an array of structures that are associated with each of the drivers in the I/O system hierarchy required to complete the specific I/O request. Each of these structures, called "stack locations", contain function and parameter information for each driver. (For more information see the section in this document on *Driver Layering*.)

The primary fields of an IRP include the following:

o - File object. This field points to the file object that the request is being performed on.

o - MDL. This field points to the MDL(s) associated with the I/O operation. The MDL(s) describe the buffer or buffers being used for the operation in terms of both their virtual addresses and physical page numbers. MDLs also describe the length of the buffer.

o - User service-independent parameters. These fields contain information about the standard parameters to I/O system services. An example is a field that contains a pointer to the referenced event object that the user specified in a service call.

o - Thread. This field contains a pointer to the thread that originally requested the I/O operation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - I/O status. This field contains the final status of the operation. It is copied into the user's I/O status block variable when the operation is complete.

o - Flags. This field describes to the various drivers and I/O system subroutines the type of operation that is represented by the IRP. For example, *IRP_INPUT_OPERATION* is a flag that indicates that this is an input operation and buffer copying during I/O completion needs to take place between a system allocated buffer and the user's buffer.

o - IRP stack location management. These fields keep track of which stack location in the IRP stack is the current location and how many locations are in the stack.

o - Device queue entry. This structure is used to queue to IRP to the device object device queue.

o - APC entry. This structure is used to allow the IRP to be used as an APC. It may either represent the special kernel APC used in I/O completion or the caller's APC routine.

9.2. Volume Parameter Block

A Volume Parameter Block (VPB) is used to keep track of the volume in a specific device. It also associates the volume with a specific file system that is currently managing the volume.

The major fields of a VPB are as follows:

o - Volume label. This field stores the name of the volume. This is useful when a removable volume is accessed but the media has been taken out of the drive. It gives the system a way to refer to the volume that the user can understand.

o - Volume serial number. This field stores the serial number of the volume. This is useful when two or more volumes mounted in the system have the same name. The serial number uniquely identifies the volume. A serial number of zero is invalid.

o - Real device object. This field is a pointer to the device object of the physical device itself on which the media is currently mounted.

o - Device object. This field is a pointer to the file system's device object for the volume. It is set by the file system after a successful mount operation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

9.3. File Object

A file object is the object used to represent files in **Windows NT**. These objects are created by the object management parse routine for device objects in the I/O system when a file is being opened or created. They represent the actual file itself.

The major fields of a file object are as follows:

o - Device object. This field is a pointer to the device object on which the file resides. This field is interrogated by the I/O system to determine which driver should be invoked when the file is being accessed through I/O system services. If the device object has a VPB associated with it, then the device driver for the VPB's file system device object is invoked rather than the real device's driver.

o - VPB. This field is a pointer to the Volume Parameter Block for the volume that the file resides on, if any.

o - File system context. These fields are reserved for use by the file system. They are undefined for I/O system use.

o - File name. This field contains the volume-relative name of the file.

o - Synchronization objects. These fields are used to control caller synchronization to the file. They allow the caller to wait on the file handle, for example.

9.4. Driver Object

A driver object is used to represent the driver code and data. It is used to keep track of the entry points for the driver as well as where the driver is currently loaded. Driver objects are used by the I/O system and the Configuration Manager.

The major fields of a driver object are as follows:

o - Entry points. These fields keep track of the routine entry points in the driver.

o - Device object. This field is a list head of all of the device objects that are being serviced by this device driver. This list represents the number of reasons why the driver cannot be unloaded from the system. The driver cannot be unloaded from the system until all of its device object have been deleted.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - Driver object. This field is a pointer to the next driver object in the system. It is used to link all of the drivers in the system together.

o - Driver base. This field contains the base system virtual address of the driver itself. It is used when the driver is being unloaded.

o - Driver size. This field contains the size of the driver. It is used when the driver is being unloaded.

9.5. Device Object

A device object is a permanent object used to represent a physical, logical, or virtual device. The collection of all device objects represents all known devices in the system. Device objects are used by system services and drivers.

The major fields of a device object are as follows:

o - Reference count. This field represents the number of reasons why this particular device object cannot be deleted.

o - Driver object. This field points to the driver object for this device. It is used by the I/O system to locate the entry points to the driver so that it may be invoked by system services and other drivers.

o - Device objects. These fields link this device object to other device objects for this driver and for devices attached to this device.

o - Current IRP. This field is a pointer to the IRP that this driver is currently working on if it is busy.

o - Timer information. These fields are used by the I/O system to implement the timers for the driver.

o - Device extension. This field is a pointer to the driver-specific extension to the device object.

o - Device type. This field specifies the type of device that the object represents.

o - Device queue. This structure is used to allow IRPs to be queued to the device using kernel synchronization and worked on.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

9.6. Controller Object

A controller object represents a hardware controller. It is used by drivers to synchronize access to the controller by various devices.

The major fields of a controller object are as follows:

o - Wait queue. This structure allows device objects to be queued and dequeued to/from the controller using kernel-provided synchronization.

9.7. Adapter Object

An adapter object represents a hardware bus adapter or DMA controller. It is used to synchronize access to the hardware.

The major fields of an adapter object are as follows:

o - Channel information. These fields describe the channels on the adapter.

o - Map register information. These fields describe the map registers in the adapter.

o - Map register allocation control. These fields keep track of the allocation of map registers in the adapter.

o - Wait queue. This structure allows device objects to be queued and dequeued to/from the adapter using kernel-provided synchronization.

10. I/O System APIs

The APIs described in this section are used by various components of the I/O system. Some are used by executive services, some are used by file systems, some are used by I/O subroutines, and some are used by drivers. All of the functions must be called from kernel mode.

Some of these functions are implemented as separate procedures, some as inline routines, and some as C language macros.

This section describes the following APIs:

IoAbortInvalidRequest - Abort an invalid I/O Request Packet.
IoAllocateAdapterChannel - Allocate adapter channel and execute routine.
IoAllocateErrorLogEntry - Allocate error log entry.
IoAllocateIrp - Allocate I/O Request Packet.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IoAllocateMdl - Allocate a Memory Descriptor List.
IoAsynchronousPageWrite - Write page data to the paging file asynchronously.
IoAttachDeviceByName - Attach two device objects.
IoBuildAsynchronousFsdRequest - Build asynchronous I/O Request Packet for an FSD.
IoBuildFspRequest - Build I/O Request Packet for an FSP.
IoBuildPartialMdl - Build a partial Memory Descriptor List.
IoBuildSynchronousFsdRequest - Build synchronous I/O Request Packet for an FSD.
IoCallDriver - Invokes a driver at its major function entry point.
IoCancelThreadIo - Cancel all I/O for a thread.
IoCheckDesiredAccess - Check desired access against granted access.
IoCheckFunctionAccess - Check function access against granted access.
IoCheckShareAccess - Check shared access request to a file.
IoCreateController - Create a controller object.
IoCompleteRequest - Complete an I/O request.
IoCreateDevice - Create a device object.
IoCreateFile - Create/open a file.
IoCreateStreamFile - Create a stream file object.
IoDeallocateAdapterChannel - Deallocate an adapter channel
IoDeallocateController - Deallocate a controller.
IoDeallocateIrp - Deallocate an I/O Request Packet.
IoDeallocateMdl - Deallocate a Memory Descriptor List.
IoDeleteController - Delete a controller object.
IoDeleteDevice - Delete a device object.
IoDeregisterFileSystem - Deregister driver as an active file system.
IoDetachDevice - Detach two device objects.
IoFlushAdapterBuffers - Flush adapter buffers to memory or device.
IoGetAttachedDevice - Get pointer to highest level attached device.
IoGetCurrentIrpStackLocation - Get pointer to IRP stack location.
IoGetNextIrpStackLocation - Get pointer to next IRP stack location.
IoGetRelatedDeviceObject - Get device object related to specified file object.
IoGetRequestorProcess - Get process of I/O requestor.
IoInitializeDpcRequest - Initialize a DPC for later posting.
IoInitializeTimer - Initialize a one-second timer.
IoIsOperationSynchronous - Determine whether an I/O operation is synchronous.
IoMakeAssociatedIrp - Allocate and initialize an associated IRP.
IoMapTransfer - Map an I/O transfer in DMA controller.
IoPageRead - Build a page read request packet for the pager.
IoQueryInformation - Query information about a file.
IoRegisterFileSystem - Register driver as an active file system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IoRemoveShareAccess - Remove the share access information when a file is closed.
IoRequestDpc - Request a DPC routine execution.
IoSendMessage - Send terminal input message to port.
IoSetCompletionRoutine - Set completion routine and context in IRP stack.
IoSetShareAccess - Set share access information for a file open.
IoStartNextPacket - Start the next I/O Request Packet, if any.
IoStartPacket - Start the current I/O Request Packet if device not busy.
IoStartTimer - Start a one-second timer.
IoStopTimer - Stop a one-second timer.
IoSynchronousPageWrite - Write page data to the paging file synchronously.
IoUpdateShareAccess - Update share access information for a file.
IoWriteErrorLogEntry - Queue error log entry to be written to log file.

10.1. IoAbortInvalidRequest

A driver may abort an invalid I/O Request Packet using the **IoAbortInvalidRequest** function:

```
VOID
IoAbortInvalidRequest(
        IN PIRP Irp
        );
```

Parameters:

Irp - A pointer to the IRP that represents the I/O request.

The **IoAbortInvalidRequest** function is invoked to abort a request represented by an IRP when the packet specifies an invalid operation. For example, this routine is invoked when an attempt is made to read from a nonexistent sector on a disk device. This function causes the entire operation to be aborted. That is, the initiator's I/O operation is not completed normally, if the request has not already been successfully queued at a driver level above the current driver that invoked the function.

This function must be invoked at DISPATCH_LEVEL.

Once this function has been invoked the IRP is no longer accessible to the driver that made the call.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.2. IoAllocateAdapterChannel

A driver may allocate an adapter object channel and cause its execution routine to be invoked using the **IoAllocateAdapterChannel** function:

VOID
IoAllocateAdapterChannel(
  IN PADAPTER_OBJECT *AdapterObject,*
  IN PDEVICE_OBJECT *DeviceObject,*
  IN ULONG *NumberOfMapRegisters,*
  IN PDRIVER_CONTROL *ExecutionRoutine,*
  IN PVOID *Context*
  );

Parameters:

AdapterObject - A pointer to the adapter object that represents the adapter channel to be allocated.

DeviceObject - A pointer to the device object that represents the device on which the I/O is to be performed.

NumberOfMapRegisters - Supplies the number of map registers to be allocated. If zero, only the adapter is allocated and no map registers are allocated.

ExecutionRoutine - The address of a routine to be executed once the adapter channel has successfully been allocated.

Context - A context parameter to pass to the *ExecutionRoutine* when it is invoked.

The routine specified by the *ExecutionRoutine* parameters has the following type definition:

typedef
IO_ALLOCATION_ACTION
(*PDRIVER_CONTROL) (
  IN PDEVICE_OBJECT *DeviceObject,*
  IN PIRP *Irp,*
  IN PVOID *MapRegisterBase,*
  IN PVOID *Context*
  );

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

DeviceObject - A pointer to the device object specified as the *DeviceObject* parameter in the call to the **IoAllocateAdapterChannel** function.

Irp - A pointer to the current I/O Request Packet that the device is working on.

MapRegisterBase - A pointer to the base address of the first map register allocated. If no map registers were allocated, then the pointer's value is null.

Context - A pointer to be used as context for the routine. This value of this parameter is the same as the *Context* parameter in the call to the **IoAllocateAdapterChannel** function.

The **IoAllocateAdapterChannel** function allocates an adapter object for the channel specified by the *AdapterObject*, waiting if necessary. If the caller requested a set of map registers, then the number of registers are also allocated. The *DeviceObject* is potentially queued to the adapter object wait queue if the channel is busy, or if the number of requested map registers cannot be immediately granted.

Once the adapter channel, and potentially the map registers, have been allocated, the driver's *ExecutionRoutine* is invoked.

The execution routine may return a value that indicates whether or not the channel and/or map registers are to remain allocated to the device. If they are, then the driver must subsequently invoke the appropriate routine to explicitly deallocate them.

This function is used by device drivers that service devices that are referenced through a bus adapter or a DMA controller.

10.3. IoAllocateController

A driver may allocate a controller object and cause its execution routine to be invoked using the **IoAllocateController** function:

VOID
IoAllocateController(
    IN PCONTROLLER_OBJECT *ControllerObject,*
    IN PDEVICE_OBJECT *DeviceObject,*
    IN PDRIVER_CONTROL *ExecutionRoutine,*
    IN PVOID *Context*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

);

Parameters:

ControllerObject - A pointer to the controller object that represents the physical device controller to be allocated.

DeviceObject - A pointer to the device object that represents the device on which the I/O is to be performed.

ExecutionRoutine - The address of a routine to be executed once the controller has successfully been allocated.

Context - A context parameter to pass to the *ExecutionRoutine* when it is invoked.

The routine specified by the *ExecutionRoutine* parameters has the following type definition:

```
typedef
IO_ALLOCATION_ACTION
(*PDRIVER_CONTROL) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID MapRegisterBase,
    IN PVOID Context
    );
```

Parameters:

DeviceObject - A pointer to the device object specified as the *DeviceObject* parameter in the call to the **IoAllocateController** function.

Irp - A pointer to the current I/O Request Packet that the device is worked on.

MapRegisterBase - A reserved pointer that should be set to null.

Context - A pointer to be used as context for the routine. This value of this parameter is the same as the *Context* parameter in the call to the **IoAllocateController** function.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **IoAllocateController** function allocates the controller specified by the *ControllerObject* parameter. The *DeviceObject* is potentially queued to the controller object wait queue if the controller is busy.

Once the controller has been allocated, the driver's *ExecutionRoutine* is invoked.

The execution routine may return a value that indicates whether or not the controller is to remain allocated to the device. If so, then the driver must subsequently invoke the appropriate routine to explicitly deallocate it.

This function is used by device drivers that service devices that are referenced through a physical device controller.

10.4. IoAllocateErrorLogEntry

An error log entry buffer may be allocated using the **IoAllocateErrorLogEntry** function:

PVOID
IoAllocateErrorLogEntry(
    IN PDEVICE_OBJECT *DeviceObject,*
    IN UCHAR *EntrySize*
    );

Parameters:

DeviceObject - A pointer to the device object to be associated with the error log entry.

EntrySize - The size of the buffer to be allocated. This value must be less than the maximum number of bytes in an entry buffer as specified by ERROR_LOG_MAXIMUM_SIZE.

The **IoAllocateErrorLogEntry** function allocates an error log entry buffer and returns a pointer to it. The size of the buffer allocated may be specified by the *EntrySize* parameter. This parameter specifies the size in bytes and must be less than the maximum size of an error log buffer.

The device driver may then fill in the data buffer and use the **IoWriteErrorLogEntry** function to post the entry to the error log thread. The device driver may treat the data as anything it likes provided that it does not overflow the size of the buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.5. IoAllocateIrp

An I/O Request Packet may be allocated and initialized using the **IoAllocateIrp** function:

PIRP
IoAllocateIrp(
    IN CCHAR *StackSize,*
    IN BOOLEAN *ChargeQuota*
    );

Parameters:

StackSize - Specifies the number of stack locations needed in the IRP. This value should equal the number of layers in the chain of layered drivers servicing this request.

ChargeQuota - Specifies whether the current thread should be charged quota for the pool memory used to allocate the IRP. This flag should only be specified by system services. File system processes should not charge quota for associated IRPs used to implement a function.

The **IoAllocateIrp** function allocates and initializes an I/O Request Packet (IRP). It allocates the packet so that it contains *StackSize* stack locations at the end of the packet for use in layering drivers.

10.6. IoAllocateMdl

An MDL (Memory Descriptor List) may be allocated and initialized using the **IoAllocateMdl** function:

PMDL
IoAllocateMdl(
    IN PVOID *VirtualAddress,*
    IN ULONG *Length,*
    IN BOOLEAN *SecondaryBuffer,*
    IN BOOLEAN *ChargeQuota,*
    IN OUT PIRP *Irp* **OPTIONAL**
    );

Parameters:

VirtualAddress - Specifies the base virtual address of the buffer that the MDL is to describe.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Length - Specifies the length of the buffer starting at *VirtualAddress* that the MDL is to describe, in bytes.

SecondaryBuffer - Indicates whether this buffer is a primary or secondary buffer. This determines how the MDL will be linked into the IRP, if specified. All buffers except for the first buffer described by an MDL in an IRP are considered secondary buffers.

ChargeQuota - Indicates whether quota should be charged to the current thread for the non-paged pool that is allocated to contain the MDL.

Irp - Optionally specifies a pointer to an IRP that the MDL is to be associated with. If this parameter is specified, then the MDL is linked into the IRP's MDL list according to the value of *SecondaryBuffer*.

The **IoAllocateMdl** function allocates and initializes a Memory Descriptor List. The MDL is allocated in such a way that it can later be used to map the buffer; that is, there is enough storage to contain the Page Frame Numbers (PFNs) that map the buffer into physical memory. The PFNs themselves are not initialized. The MDL header is initialized to describe the specified buffer.

This function is used by the I/O system to map the caller's buffers. It is also used by any device driver that needs to break a buffer into parts, each mapped by an MDL, or to map a separate buffer. Mapping a complete buffer may be used to when a driver is given a pointer to the caller's buffer and needs to lock it, or when it needs to lock a buffer that the driver has allocated.

10.7. IoAsynchronousPageWrite

The modified page writer can asynchronously write pages of data to the paging file or to a mapped file using the **IoAsynchronousPageWrite** function:

```
NTSTATUS
IoAsynchronousPageWrite(
    IN PFILE_OBJECT FileObject,
    IN PMDL MemoryDescriptorList,
    IN PLARGE_INTEGER StartingOffset,
    IN PIO_APC_ROUTINE ApcRoutine,
    IN PVOID ApcContext,
    OUT PIO_STATUS_BLOCK IoStatusBlock
    );
```

Parameters:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileObject - A pointer to a referenced file object representing the file to write.

MemoryDescriptorList - A Memory Descriptor List (MDL) that describes the locked-down buffer containing the data to write to the file.

StartingOffset - The starting byte offset where the write operation is to begin.

ApcRoutine - The address of an APC routine that is to be executed once the I/O operation is complete. This is the only valid synchronization technique for this type of request.

ApcContext - A value that will be given to the caller's *ApcRoutine* when it is invoked.

IoStatusBlock - A variable to receive the final completion status and information about the write operation. The number of bytes actually written is returned in the *Information* field.

The **IoAsynchronousPageWrite** function gives the **Windows NT** Modified Page Writer a quick way of building and starting an I/O request to write data to a file asynchronously. This allows I/O completion to be short circuited for paging I/O.

The function writes the number of bytes specified by the MDL from the buffer described by the MDL, beginning at the *StartingOffset* within the file. The *ApcRoutine* is invoked once the operation has completed.

This function is only invoked by the **Windows NT** Modified Page Writer.

10.8. IoAttachDeviceByName

A device object may be attached to another device object to allow association between the two using the **IoAttachDevice** function:

```
NTSTATUS
IoAttachDevice(
    IN PDEVICE_OBJECT SourceDevice,
    IN PSTRING TargetDevice
    );
```

Parameters:

SourceDevice - A pointer to the device object that should be attached. This device object must belong to the calling driver.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

TargetDevice - The name of the device that the *SourceDevice* should be attached to for servicing.

The **IoAttachDevice** function allows a device driver to attach a device object to another device object. This association allows operations given to the lower level device driver to be given to the device driver for the *SourceDevice* device object. It also establishes the layering between drivers so that the same IRP may be used by all layers. If the device has already been attached to, then this function returns an appropriate error status.

This service is used by intermediate device drivers. It allows a driver to attach a device object to another device in such a way that any requests being made to the original device will now be given to the intermediate device driver's device object. For example, a file system normally interfaces directly to a disk device driver. This function allows an intermediate driver, such as a striper driver, to attach itself to the disk driver's device object such that when the file system attempts to communicate with the disk driver via its device object, the request will be routed to the intermediate device driver first.

10.9. IoBuildAsynchronousFsdRequest

A file system driver may build an asynchronous I/O Request Packet for use in performing a read or a write to another device driver using the **IoBuildAsynchronousFsdRequest** function:

```
PIRP
IoBuildAsynchronousFsdRequest(
    IN ULONG MajorFunction,
    IN PDEVICE_OBJECT DeviceObject,
    IN OUT PVOID Buffer,
    IN ULONG Length,
    IN PLARGE_INTEGER StartingOffset,
    OUT PIO_STATUS_BLOCK IoStatusBlock
    );
```

Parameters:

MajorFunction - The function that the FSD is requesting the lower level driver to perform. This value must be one of *IRP_MJ_READ* or *IRP_MJ_WRITE*.

DeviceObject - A pointer to the device object that represents the target of the read or the write operation. The device driver for this device will be invoked with the IRP that this function builds.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Buffer - A pointer to a buffer that, on a write contains the data to write, or, on a read is to receive the data read.

Length - Specifies the length, in bytes, of the data to be read or written. (For many devices, such as disks, this value must be an integral of 512.)

StartingBlock - Specifies the byte offset that the transfer is to begin. (For many devices, such as disks, this value must specify a sector boundary.)

IoStatusBlock - A variable to receive the final status and information from the operation. The final status will be written to the *Status* field and the number of bytes read or written will be contained in the *Information* field of this variable once the operation has completed. This variable must be in a nonpageable page.

The **IoBuildAsynchronousFsdRequest** function builds an I/O Request Packet (IRP) that can be given to a device driver to perform an asynchronous read or a write operation. The IRP contains only enough information to get the operation started and to complete to the FSD. No other context information is kept track of since the request is context-independent.

It is up to the FSD to determine when the I/O has completed, if it is interested. This can be done by setting a completion routine in the returned IRP using the **IoSetCompletionRoutine** function.

This function is used by file system drivers operating in a thread-context-independent manner to issue I/O requests.

10.10. IoBuildFspRequest

A file system process may build an I/O Request Packet for use in performing a read or a write to another device driver using the **IoBuildFspRequest** function:

PIRP
IoBuildFspRequest(
    IN ULONG *MajorFunction,*
    IN PDEVICE_OBJECT *DeviceObject,*
    IN OUT PVOID *Buffer,*
    IN ULONG *Length,*
    IN PLARGE_INTEGER *StartingOffset,*
    IN PKEVENT *Event* **OPTIONAL,**
    IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN PVOID *ApcContext* **OPTIONAL,**
OUT PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

MajorFunction - The function that the FSP is requesting the target device to perform. This value must be one of *IRP_MJ_READ* or *IRP_MJ_WRITE*.

DeviceObject - A pointer to the device object that represents the target of the read or the write operation. The device driver for this device will be invoked with the IRP that this function builds.

Buffer - A pointer to a buffer that contains the data to write, or is the location that is to receive the data read.

Length - Specifies the length, in bytes, of the data to be read or written. (For many devices, such as disks, this value must be an integral of 512.)

StartingOffset - Specifies the byte offset at which the transfer is to begin. (For many devices, such as disks, this value must be the start of a sector boundary.)

Event - An optional pointer to a kernel event that should be set to the Signaled state once the operation completes.

ApcRoutine - Optionally specifies the address of an APC routine that should be executed upon completion of the request. This routine is invoked with the address of the I/O status block and the *ApcContext* value as its parameters.

ApcContext - Optionally specifies a context parameter that should be passed to the *ApcRoutine* when it is invoked upon completion of the I/O operation.

IoStatusBlock - A variable to receive the final status and information from the operation. The final status will be written to the *Status* field and the number of bytes read or written will be contained in the *Information* field of this variable once the operation has completed.

The **IoBuildFspRequest** function builds an I/O Request Packet (IRP) that can be given to a device driver to perform a read or write operation. The IRP closely resembles an IRP that would be built by the general **NtReadFile** or **NtWriteFile**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

services. This packet can be modified by the FSP before the FSP invokes the device driver with the request, if necessary.

Completion of the I/O request occurs as for normal I/O requests, except that this packet refers to a kernel event, whereas a normal packet refers to an event object.

This function is used by file system processes (FSPs) operating in their own context and issuing I/O operations on behalf of I/O system users.

10.11. IoBuildPartialMdl

A Memory Descriptor List (MDL) may be built to describe part of a buffer described by another MDL (i.e., "master") by using the **IoBuildPartialMdl** function:

VOID
IoBuildPartialMdl(
    IN PMDL *SourceMdl,*
    IN OUT PMDL *TargetMdl,*
    IN PVOID *VirtualAddress,*
    IN ULONG *Length,*
    );

Parameters:

SourceMdl - A pointer to the MDL that describes the original buffer, a subset of which is to be mapped by this function.

TargetMdl - A pointer to an MDL to be filled in that describes the desired subset of the buffer specified by the *SourceMdl* parameter. The MDL must be large enough to contain the PFNs required to map the subset buffer.

VirtualAddress - Specifies the base virtual address of the *TargetMdl* buffer to be mapped; this value must be contained within the buffer mapped by the *SourceMdl.*

Length - Specifies the length in bytes to be mapped by the *TargetMdl*; this value in combination with that of *VirtualAddress* must specify a buffer that is a proper subset of the *SourceMdl* buffer. If specified as zero, then the function maps the remainder of the *SourceMdl* buffer, starting at *VirtualAddress*, as the *TargetMdl.*

The **IoBuildPartialMdl** function maps a subset of a buffer that is currently mapped by the *SourceMdl.* The *VirtualAddress* and *Length* parameters describe the mapped

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

subset. The descriptor to map the specified subset is written to the *TargetMdl*. If a length of zero is specified, then the remainder of the specified buffer is mapped starting at *VirtualAddress*. See section 4.4, *Splitting Transfers*, for an overview of how this function is used.

10.12. IoBuildSynchronousFsdRequest

A file system driver may build a synchronous I/O Request Packet for use in performing a read or a write to another device driver using the **IoBuildSynchronousFsdRequest** function:

PIRP
IoBuildSynchronousFsdRequest(
    IN ULONG *MajorFunction,*
    IN PDEVICE_OBJECT *DeviceObject,*
    IN OUT PVOID *Buffer,*
    IN ULONG *Length,*
    IN PLARGE_INTEGER *StartingOffset,*
    IN PKEVENT *Event,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock*
    );

Parameters:

MajorFunction - The function that the FSD is requesting the lower level driver to perform. This value must be one of *IRP_MJ_READ* or *IRP_MJ_WRITE*.

DeviceObject - A pointer to the device object that represents the target of the read or the write operation. The driver for this device will be invoked with the IRP that this function builds.

Buffer - A pointer to a buffer that contains the data to write, or is the location that is to receive the data read.

Length - Specifies the length, in bytes, of the data to be read or written. (For many devices, such as disks, this value must be an integral of 512.)

StartingOffset - Specifies the byte offset that the transfer is to begin. (For many devices, such as disks, this value must specify the start of a sector boundary.)

Event - A pointer to a kernel event that is to be set to the Signaled state once the operation completes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IoStatusBlock - A variable to receive the final status and information from the operation. The final status will be written to the *Status* field and the number of bytes read or written will be contained in the *Information* field of this variable once the operation has completed. This variable must be in a nonpageable page.

The **IoBuildSynchronousFsdRequest** function builds an I/O Request Packet (IRP) that can be given to a device driver to perform a synchronous read or write operation. The IRP contains only enough information to get the operation started and to complete to the FSD.

It is up to the FSD to determine when the I/O has completed by waiting on the *Event*. It should be noted that performing the wait operation causes the current thread to wait. Therefore this operation should be used during initialization of the driver or when the I/O being performed is synchronous.

10.13. IoCallDriver

The I/O system or a layered driver may invoke a driver at its major function entry using the **IoCallDriver** function:

NTSTATUS
IoCallDriver(
    IN PDEVICE_OBJECT *DeviceObject,*
    IN OUT PIRP *Irp*
    );

Parameters:

DeviceObject - A pointer to the device object upon which the I/O request is to be performed. The *Irp* is given to the driver that is servicing the device.

Irp - A pointer to the I/O Request Packet that represents the request to be performed.

The **IoCallDriver** function invokes a driver's major function routine according to the IRP function code. The driver that is called is the one that is servicing the device specified by *DeviceObject*. The IRP specified by *Irp* is passed to the driver at its appropriate entry point.

This function is used by I/O system services and layered drivers.

Once this function has been invoked the IRP is no longer accessible to the driver.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.14. IoCancelThreadIo

All pending I/O operations for a given thread may be canceled by using the **IoCancelThreadIo** function:

```
VOID
IoCancelThreadIo(
    IN PETHREAD Tcb
    );
```

Parameters:

> *Tcb* - A pointer to the Thread Control Block for the thread whose pending I/O operations should be canceled.

The **IoCancelThreadIo** function chases all of the pending I/O requests for the specified thread and cancels each one. Some requests may already be in a state of being completed and these are allowed to complete. Pending operations that are not in a state of being completed are canceled if they can be. Most of the operations that would normally occur during I/O completion, such as unlocking buffers and dereferencing events, are performed so that thread rundown occurs properly.

This function is used by the executive during thread rundown.

10.15. IoCheckDesiredAccess

A driver can check whether a desired access is permitted to a file using the **IoCheckDesiredAccess** function:

```
NTSTATUS
IoCheckDesiredAccess(
    IN OUT PACCESS_MASK DesiredAccess,
    IN ACCESS_MASK GrantedAccess
    );
```

Parameters:

> *DesiredAccess* - Supplies a pointer to the access mask that represents the type of access to the file that is desired.

> *GrantedAccess* - Supplies the access mask that represents the access already granted to the file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **IoCheckDesiredAccess** function checks whether or not the caller has the desired access rights to a file based on the current granted access mask. If not, then an access denied error status is returned. Otherwise, a successful status is returned and the *DesiredAccess* variable is overwritten with an expanded representation of the actual access desired. That is, all generic accesses to the file are expanded to individual access bits.

This function is used as a security filter by kernel mode components that are accessing files for user mode clients. Since no access checks are made for kernel mode requests, a server system needs a way of determining whether or not its client has the appropriate access to perform a given function on the file. The **IoCheckDesiredAccess** routine is used to provide this functionality.

10.16. IoCheckFunctionAccess

A driver can check whether an operation is permitted to a file using the **IoCheckFunctionAccess** function:

```
NTSTATUS
IoCheckFunctionAccess(
        IN ACCESS_MASK GrantedAccess,
        IN UCHAR MajorFunction,
        IN UCHAR MinorFunction,
        IN PFILE_INFORMATION_CLASS FileInformationClass OPTIONAL,
        IN PFS_INFORMATION_CLASS FsInformationClass OPTIONAL
        );
```

Parameters:

GrantedAccess - Supplies the access mask that represents the access granted to the file.

MajorFunction - Supplies the IRP major function code that represents the operation to be performed on the file.

MinorFunction - Supplies the IRP minor function code that represents the operation to be performed on the file, if any.

FileInformationClass - Supplies the file information class for the set or query operation to be performed on the file. This parameter is optional if the function code does not specify a set or query operation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FsInformationClass - Supplies the file system information class for the set or query volume operation to be performed. This parameter is optional if the function code does not specify a set or query volume operation.

The **IoCheckFunctionAccess** function checks whether or not the caller has the access rights to a file to perform a specific function given his granted access. If not, then an access denied error status is returned. Otherwise, a successful status is returned.

This function is used as a security filter by kernel mode components that are accessing files for user mode clients. Since no access checks are made for kernel mode requests, a server system needs a way of determining whether or not its client has the appropriate access to perform a given function on the file. The **IoCheckFunctionAccess** routine is used to provide this functionality.

10.17. IoCheckShareAccess

A file system may check whether shared access is permitted to a file using the **IoCheckShareAccess** function:

NTSTATUS
IoCheckShareAccess(
 IN ACCESS_MASK *DesiredAccess,*
 IN ULONG *DesiredShareAccess,*
 IN OUT PFILE_OBJECT *FileObject,*
 IN OUT PSHARE_ACCESS *ShareAccess*
 IN BOOLEAN *Update*
 );

Parameters:

DesiredAccess - Supplies the types of access that the current open request would like to the file. This parameter is generally the same *DesiredAccess* parameter given to the file system by the I/O system when the open request is made via the create file IRP.

DesiredShareAccess - Supplies the types of shared access that the current open request would like to the file. This parameter is generally the same *ShareAccess* parameter given to the file system by the I/O system when the open request is made via the create file IRP.

FileObject - A pointer to the file object for the current open request.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ShareAccess - A pointer to the common share access data structure associated with the file being opened. This structure is treated as an opaque type by drivers.

Update - Supplies a BOOLEAN value indicating whether the share access information for the file is to be updated if the open request is permitted.

The **IoCheckShareAccess** function checks a file open request to determine if the types of desired and shared accesses specified are compatible with the way in which the file is currently being accessed by other opens of the file.

File systems maintain state about files through structures called File Control Blocks **(FCBs)**. The **SHARE_ACCESS** is a structure that describes how the file is currently accessed by all opens. It is contained in the FCB as part of the open file state.

If the requestor's access to the file is compatible with the way in which the file is currently open, a status of *STATUS_SUCCESS* is returned and the **SHARE_ACCESS** information for the file is updated according to the *Update* parameter. If the file request is denied because of a file sharing violation, then a status of *STATUS_SHARING_VIOLATION* is returned.

This function is used by file systems, after ensuring that the requestor has access to the file, to determine whether or not the open request can be satisfied according to the **Windows NT** file sharing semantics.

10.18. IoCompleteRequest

The processing for an I/O request may be declared complete using the **IoCompleteRequest** function:

```
VOID
IoCompleteRequest(
    IN PIRP Irp,
    IN CCHAR PriorityBoost
    );
```

Parameters:

Irp - A pointer to the I/O Request Packet that represents the I/O operation to be completed.

PriorityBoost - Specifies the amount of priority boost the requesting thread should be given when the special kernel APC is queued to it for I/O completion.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **IoCompleteRequest** function "completes" an I/O operation for the request packet that represents it. Completing an operation involves notifying all drivers in the IRP stack that the I/O operation has completed, provided that they would like to know (that is, they invoked the **IoSetCompletionRoutine** function to set the address of a completion routine). It also involves unlocking the caller's buffers, posting the event if one was specified, queueing the APC to the requesting thread if one was specified, as well as other operations.

This function must be invoked at DISPATCH_LEVEL.

This function is used by drivers to indicate that the I/O operation has completed and the driver is finished with its processing.

10.19. IoCreateController

A driver can create a controller object using the **IoCreateController** function:

PDEVICE_OBJECT
IoCreateController(
        );

<u>Parameters:</u>

> *None.*

The **IoCreateController** function allocates and initializes a controller object for use by a driver. The controller object is used to synchronize access to various hardware devices connected to the controller hardware. For more information on controller objects and how they are used see the *Driver Model Description* section of this document.

10.20. IoCreateDevice

A driver can create a device object using the **IoCreateDevice** function:

PDEVICE_OBJECT
IoCreateDevice(
        IN PDRIVER_OBJECT *DriverObject,*
        IN ULONG *DeviceExtension,*
        IN PSTRING *DeviceName* **OPTIONAL,**
        IN DEVICETYPE *DeviceType,*
        IN BOOLEAN *Exclusive*
        );

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

DriverObject - Supplies a pointer to the driver object created by the I/O system when the driver was loaded. This object indicates which driver is to be associated with the device object being created.

DeviceExtension - The size, in bytes, of any extension that should be allocated beyond the end of the I/O system's notion of the device object. This part of the device object can be used by the device driver to contain context information or a communication region for use in communicating between an FSD and an FSP.

DeviceName - An optional pointer to a string that describes the name of the device that the device object represents. This name is associated with the device object and inserted in the object directory hierarchy.

DeviceType - The type of the device that the device object represents. The values for this parameter must have the same representation and meaning that they have in the **NtQueryInformationFile** system service.

Exclusive - Indicates that this device is created as an exclusive device; that is, once the device object is "opened" by one process, no other processes may open the device.

The **IoCreateDevice** function allocates and initializes a device object for use by a driver. The object is a permanent object, is exclusive if the *Exclusive* parameter is TRUE, and is "owned" by the device driver associated with the *DriverObject* parameter. The device object is also linked into the I/O database in such a way that if the driver is unloaded, all of its device objects can be found.

Device objects for disks, tapes, CD ROMs, and RAM disks are given a Volume Parameter Block (VPB) that is initialized to indicate that the volume has never been mounted on the device.

A device driver may use the *DeviceExtension* parameter to cause storage to be allocated at the end of the device object. This storage, located by the *DeviceExtension* field in the device object, may be used by the device driver to keep device-specific context information.

This function must be invoked by each driver to create one or more device objects; otherwise the driver cannot be located by the I/O system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.21. IoCreateFile

The I/O system or a driver can create or open a file using the **IoCreateFile** function:

NTSTATUS
IoCreateFile(
OUT PHANDLE *FileHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes,*
OUT PIO_STATUS_BLOCK *IoStatusBlock,*
IN PLARGE_INTEGER *AllocationSize* **OPTIONAL,**
IN ULONG *FileAttributes,*
IN ULONG *ShareAccess,*
IN HANDLE *ErrorPort* **OPTIONAL,**
IN ULONG *Disposition,*
IN ULONG *Options,*
IN PVOID *EaBuffer* **OPTIONAL,**
IN ULONG *EaLength,*
IN BOOLEAN *ForceAccessCheck,*
IN BOOLEAN *PagingFileOpen*
);

Parameters:

FileHandle - A variable to receive the handle to the file.

DesiredAccess - Specifies the type of access that the caller requires to the file. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, a security descriptor, a quality of service descriptor, and a set of file object attributes flags. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The actual action taken by the system is written to the *Information* field of this variable.

AllocationSize - Optionally specifies the initial allocation size of the file in bytes. The size has no effect unless the file is created, overwritten, or superseded.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileAttributes - Specifies the file attributes for the file. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

ShareAccess - Specifies the type of share access that the caller would like to the file. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

ErrorPort - Optionally specifies a handle to an open port to be RPC'd to if an error such as "the wrong volume is in the drive" occurs. If a handle is specified, the caller must have **PORT_READ** and **PORT_WRITE** access to the port.

CreateDisposition - Specifies the actions to be taken if the file does or does not already exist. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

CreateOptions - Specifies the options that should be used when creating or opening the file. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

EaBuffer - Optionally specifies a list of EAs that should be set on the file if it is created. This is done as an atomic operation. That is, if an error occurs setting the EAs on the file, then the file will not be created.

EaLength - Supplies the length of the *EaBuffer*. If no buffer is supplied then this value should be zero.

ForceAccessCheck - Indicates whether access checking should be forced even though the caller's previous mode is kernel. If TRUE, then access checking will be performed.

PagingFileOpen - Indicates whether a paging file is being opened by the Modified Page Writer. Special handling is performed in some file systems when a paging file is being opened.

The **IoCreateFile** function is used by the I/O system to implement the **NtCreateFile** and **NtOpenFile** system services. It is also used by those kernel components that require special processing such as forcing access checks, or opening a paging file.

Each of the parameters to this function are syntactically and semantically the same as those specified in the **NtCreateFile** system service. The only differences between this function and the system service are the final two parameters.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.22. IoCreateStreamFile

A file system can create a stream file object using the **IoCreateStreamFile** function:

PFILE_OBJECT
IoCreateStreamFile(
    IN PFILE_OBJECT *FileObject* **OPTIONAL,**
    IN PDEVICE_OBJECT *DeviceObject* **OPTIONAL**
    );

Parameters:

FileObject - A pointer to a file object that the stream file object is to be modeled after. This parameter is optional if the *DeviceObject* parameter is specified.

DeviceObject - A pointer to device object representing the physical device on which the stream file is being opened. This parameter is optional if the *FileObject* parameter is specified.

The **IoCreateStreamFile** function creates a stream file object that can be used to cache a stream of a file other than the data of the file. This function is used by file systems to represent those parts of the on-disk structure that are not included in proper files.

10.23. IoDeallocateAdapterChannel

A driver can explicitly deallocate an adapter channel using the **IoDeallocateAdapterChannel** function:

VOID
IoDeallocateAdapterChannel(
    IN PADAPTER_CHANNEL *AdapterObject*
    );

Parameters:

AdapterObject - A pointer to the adapter object representing the adapter channel to be deallocated.

The **IoDeallocateAdapterChannel** function frees an adapter channel that was previously allocated using the **IoAllocateAdapterChannel** function. If any map registers were allocated, then they are deallocated as well.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function is used by a device driver to allocate an adapter channel and/or map registers because the device it services is referenced through a bus adapter or a DMA controller.

10.24. IoDeallocateController

A driver may explicitly deallocate a controller object using the **IoDeallocateController** function:

VOID
IoDeallocateController(
    IN PCONTROLLER_OBJECT *ControllerObject*
    );

Parameters:

ControllerObject - A pointer to the controller object that is to be deallocated.

The **IoDeallocateController** function frees a controller object that was previously allocated using the **IoAllocateController** function.

This function is used by a device driver to allocate a controller because the device it services is referenced through a device controller.

10.25. IoDeallocateIrp

An I/O Request Packet may be deallocated using the **IoDeallocateIrp** function:

VOID
IoDeallocateIrp(
    IN PIRP *Irp*
    );

Parameters:

Irp - A pointer to the I/O Request Packet to be deallocated.

The **IoDeallocateIrp** function deallocates the specified IRP.

This function is used by the I/O system to deallocate IRPs once all of the processing for the request has been completed. It is also possible, in some cases, for a file system process to use this service to dispose of associated IRPs that the FSP created to implement a request.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.26. IoDeleteController

A controller object can be deleted using the **IoDeleteController** function:

VOID
IoDeleteController(
    IN PCONTROLLER_OBJECT *Controller*
    );

Parameters:

Controller - A pointer to the controller object to delete.

The **IoDeleteController** function deletes a controller object. This function is invoked when a device driver is unloading. It is an error to attempt to delete a controller object if it is owned by a device object or if a device object is currently waiting to allocate the controller.

10.27. IoDeallocateMdl

A Memory Descriptor List (MDL) may be deallocated using the **IoDeallocateMdl** function:

VOID
IoDeallocateMdl(
    IN PMDL *Mdl*
    );

Parameters:

Mdl - A pointer to the MDL to be deallocated.

The **IoDeallocateMdl** function deallocates an MDL that was previously allocated through the **IoAllocateMdl** function. The function frees the storage for the MDL back to the MDL pool from which it was allocated.

This function is used by the I/O system completion code as well as by any drivers that perform their own local buffer management.

10.28. IoDeleteDevice

A driver can delete a device object using the **IoDeleteDevice** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
IoDeleteDevice(
    IN PDEVICE_OBJECT *DeviceObject*
    );

Parameters:

DeviceObject - A pointer to the device object that is to be deleted.

The **IoDeleteDevice** function marks a device object for deletion as soon as its reference count is decremented to zero. No other references may be established to the object once it is marked for deletion; it is treated as if the object does not exist.

This function is invoked by a device driver to delete its device objects when the driver is being unloaded.

10.29. IoDeregisterFileSystem

A file system may deregister itself as an active file system using the **IoDeregisterFileSystem** function:

VOID
IoDeregisterFileSystem(
    IN OUT PDEVICE_OBJECT *DeviceObject*
    );

Parameters:

DeviceObject - A pointer to the device object for the file system.

The **IoDeregisterFileSystem** function causes the file system specified by the *DeviceObject* to be deregistered as an active file system. This is done by simply removing the specified device object from the list of active file systems.

This function is invoked by a registered file system (see **IoRegisterFileSystem**) when the driver for the file system is being unloaded.

10.30. IoDetachDevice

A driver may use the **IoDetachDevice** function to detach one device object from another device object:

VOID
IoDetachDevice(

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        IN OUT PDEVICE_OBJECT TargetDevice
        );
```

Parameters:

TargetDevice - A pointer to the target device object that is to be detached from by the device that is servicing it.

The **IoDetachDevice** function detaches the device that is currently attached to the specified *TargetDevice*. This function disassociates two devices previous attached to each other with the **IoAttachDevice** function.

This function is invoked by intermediate drivers when they are unloading or when they have been told to stop servicing a device.

10.31. IoFlushAdapterBuffers

A driver can flush the buffers of an I/O adapter using the **IoFlushAdapterBuffers** function:

```
VOID
IoFlushAdapterBuffers(
        IN PADAPTER_OBJECT AdapterObject
        );
```

Parameters:

AdapterObject - A pointer to the adapter object representing the adapter whose buffers are to be flushed.

The **IoFlushAdapterBuffers** function is used to flush any remaining data in the I/O adapter's buffers. This function must be invoked at the end of each data transfer by all device drivers that deal with devices attached to the adapter.

10.32. IoGetAttachedDevice

The I/O system or a driver may obtain a pointer to the highest level device attached to a specific device object using the **IoGetAttachedDevice** function:

```
PDEVICE_OBJECT
IoGetAttachedDevice(
        IN PDEVICE_OBJECT DeviceObject
        );
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

> *DeviceObject* - A pointer to the device for which the highest level attached device object is to be returned.

The **IoGetAttachedDevice** function returns the highest level device attached to a specified device object. That is, it follows all of the links of the devices that are attached to the specified device. This allows the I/O system or a driver to pass a request to the highest level driver associated with a device. If no devices are attached to the specified device object, then a pointer to the specified device object is returned.

This function is invoked by the I/O system and drivers to determine what driver an IRP should be passed to.

10.33. IoGetCurrentIrpStackLocation

A driver can obtain a pointer to the current stack location in an I/O Request Packet (IRP) using the **IoGetCurrentIrpStackLocation** function:

```
PIO_STACK_LOCATION
IoGetCurrentIrpStackLocation(
        IN PIRP Irp
        );
```

Parameters:

> *Irp* - A pointer to the I/O Request Packet that contains the stack location whose address is to be returned.

The **IoGetCurrentIrpStackLocation** function returns a pointer to the current stack location in the specified IRP. This location contains the function codes, parameters, and I/O system information that describe the operation being requested to the driver.

This function is invoked by all device drivers to determine the operation to perform, as well as to determine the parameters, if any, that have been specified for the operation.

10.34. IoGetNextIrpStackLocation

A driver may obtain a pointer to the next stack location in an I/O Request Packet (IRP) using the **IoGetNextIrpStackLocation** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PIO_STACK_LOCATION
IoGetNextIrpStackLocation(
    IN PIRP *Irp*
    );

Parameters:

Irp - A pointer to the I/O Request Packet that contains the stack location whose address is to be returned.

The **IoGetNextIrpStackLocation** function returns a pointer to the next stack location in the specified IRP. This allows the current device driver to pass parameter and function code information to the next level driver using the same packet with which the current driver was invoked.

This function is invoked by all device drivers that pass IRPs to lower level drivers to pass function and parameter information. Note that even if the parameters are exactly the same, they must still be placed into the next driver's stack location since it will automatically look for them there.

10.35. IoGetRelatedDeviceObject

The device object referred to by a file object can be obtained using the **IoGetRelatedDeviceObject** function:

PDEVICE_OBJECT
IoGetRelatedDeviceObject(
    IN PFILE_OBJECT *FileObject*
    );

Parameters:

FileObject - A pointer to the file object whose related device object is returned.

The **IoGetRelatedDeviceObject** function returns a pointer to the device object that a file object refers to after all device object links have been chased.

This function is used by the I/O system and by any device driver that needs to determine the highest level device object to which a file object refers.

10.36. IoGetRequestorProcess

A driver may obtain a pointer to the process that originally made an I/O request using the **IoGetRequestorProcess** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PEPROCESS
IoGetRequestorProcess(
    IN PIRP *Irp*
    );

Parameters:

Irp - A pointer to the I/O Request Packet representing the request whose originator process is returned.

The **IoGetRequestorProcess** function allows a driver to obtain a pointer to the process data structure for the originator of a specified I/O request. This function is useful to file systems in keeping track of which processes own locks.

This function is used by file systems to keep track of lock owners and processes that have associated events with pipes.

10.37. IoInitializeDpcRequest

A device driver may initialize its device object's DPC using the **IoInitializeDpcRequest** function:

VOID
IoInitializeDpcRequest(
    IN PDEVICE_OBJECT *DeviceObject,*
    IN PIO_DPC_ROUTINE *DpcRoutine*
    );

Parameters:

DeviceObject - A pointer to the device object that contains the DPC entry that is to be initialized.

DpcRoutine - The address of a routine that is to be invoked at DISPATCH_LEVEL when the Deferred Procedure Call entry is removed from the DPC queue by the kernel.

The routine specified by the *DpcRoutine* parameter has the following type definition:

typedef
VOID
(*PIO_DPC_ROUTINE) (

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
    IN PKDPC Dpc,
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );
```

Parameters:

Dpc - A pointer to the kernel DPC used to represent the call to this procedure. This parameter is ignored by device drivers.

DeviceObject - A pointer to the device object whose request needs servicing. This is the same device object as specified in the **IoInitializeDpcRequest** and **IoRequestDpc** I/O system calls.

Irp - A pointer to the I/O Request Packet that needs to be serviced. This is generally the reason that the DPC was requested in the first place.

Context - A pointer to whatever context is required by the device driver.

The **IoInitializeDpcRequest** function is used by the device driver's initialization routine to initialize the DPC in the device driver's device object so that the DPC can be used later to submit DPC requests. This allows the driver's interrupt service routine to request a DPC through the **IoRequestDpc** interface without having to initialize the DPC at device IRQL.

10.38. IoInitializeTimer

A device driver timer may be initialized using the **IoInitializeTimer** function:

```
VOID
IoInitializeTimer(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIO_TIMER_ROUTINE TimerRoutine
    );
```

Parameters:

DeviceObject - A pointer to the device object that contains the timer to be used.

TimerRoutine - Specifies the timer routine that is to be invoked once every second with a pointer to the counter associated with the device object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The routine specified by the *TimerRoutine* parameter has the following type definition:

```
typedef
VOID
(*PIO_TIMER_ROUTINE) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PLONG TimerCounter
    );
```

Parameters:

DeviceObject - A pointer to the device object with which the timer counter is associated.

TimerCounter - A pointer to the timer counter associated with the device object.

The **IoInitializeTimer** function sets up a timer that expires once every second. Each time the timer expires, the system invokes the routine specified by the *TimerRoutine* parameter. The timer is actually started using the **IoStartTimer** function.

10.39. IoIsOperationSynchronous

A driver can determine whether an I/O operation is synchronous using the **IoIsOperationSynchronous** function:

```
BOOLEAN
IoIsOperationSynchronous(
    IN PIRP Irp
    );
```

Parameters:

Irp - Pointer to the I/O Request Packet for the operation to be checked.

The **IoIsOperationSynchronous** function checks whether the I/O request represented by the specified IRP is synchronous and returns a BOOLEAN value of TRUE if it is synchronous.

This function is used by drivers to determine whether an operation is synchronous and therefore whether or not the requestor's thread may be used by the driver to perform the I/O operation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.40. IoMakeAssociatedIrp

An associated I/O Request Packet can be allocated and initialized using the **IoMakeAssociatedIrp** function:

PIRP
IoMakeAssociatedIrp(
    IN PIRP *Irp,*
    IN CCHAR *StackSize*
    );

Parameters:

Irp - Pointer to the master I/O Request Packet with which the new packet should be associated.

StackSize - Specifies the number of stack locations needed in the IRP. This value should equal the number of layers in the chain of layered drivers servicing this request.

The **IoMakeAssociatedIrp** allocates and initializes an I/O Request Packet and associates it with a master packet. The count in the master packet should already have been set by the caller to the number of packets to be associated with it. The number of stack locations to be allocated for the associated IRP is specified by the *StackSize* parameter.

10.41. IoMapTransfer

A DMA I/O transfer may be mapped through an adapter or DMA controller using the **IoMapTransfer** function:

VOID
IoMapTransfer(
    IN PADAPTER_OBJECT *AdapterObject,*
    IN PMDL *Mdl,*
    IN PVOID *MapRegisterBase,*
    IN PVOID *CurrentVa,*
    IN ULONG *Length,*
    IN BOOLEAN *WriteToDevice*
    );

Parameters:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

AdapterObject - A pointer to the adapter object representing the adapter or DMA controller where the map registers reside.

Mdl - A pointer to a Memory Descriptor List (MDL) that maps the locked-down buffer to/from which the I/O is to take place.

MapRegisterBase - A pointer to the base of the map registers in the adapter or DMA controller. This value is passed to the driver's *ExecutionRoutine* when the adapter object and map registers have been allocated by the **IoAllocateAdapterChannel** function.

CurrentVa - A pointer to the current virtual address in the buffer described by the *Mdl* where the I/O operation is to take place.

Length - Supplies the length of the transfer to map.

WriteToDevice - Supplies a BOOLEAN value that indicates that the direction of the data transfer is to the device.

The **IoMapTransfer** function loads the appropriate map registers in the adapter or DMA controller to cause the I/O operation to map to the appropriate memory locations described by the *Mdl, CurrentVa,* and *Length* parameters.

This function is used by device drivers to map DMA I/O to the appropriate memory buffer.

10.42. IoPageRead

The pager can read pages of data from the paging file or from a mapped file using the **IoPageRead** function:

```
NTSTATUS
IoPageRead(
    IN PFILE_OBJECT FileObject,
    IN PMDL MemoryDescriptorList,
    IN PLARGE_INTEGER StartingOffset,
    IN PKEVENT Event,
    OUT PIO_STATUS_BLOCK IoStatusBlock
    );
```

Parameters:

FileObject - A pointer to a referenced file object representing the file to be read.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

MemoryDescriptorList - A Memory Descriptor List (MDL) that describes the locked-down buffer into which data from the file is to be read.

StartingOffset - The starting byte offset within the specified file where the read operation is to begin.

Event - A pointer to a kernel event that should be set to the Signaled state once the I/O operation is complete. This is the only valid synchronization technique for this type of request.

IoStatusBlock - A variable to receive the final completion status and information for the read operation. The number of bytes actually read is returned in the *Information* field.

This variable must be locked into memory so that it cannot move until the I/O is complete.

The **IoPageRead** function gives the **Windows NT** Pager a quick way of building and starting an I/O request to read data from a file. This allows I/O completion to be short circuited so that no APCs or pagefaults occur while attempting to complete a page read operation.

The function reads the number of bytes specified by the MDL into the buffer described by the MDL, beginning at the *StartingBlock* within the file. The *Event* is set to the Signaled state once the operation has completed.

This function is only invoked by the **Windows NT** Pager.

10.43. IoQueryInformation

Information about a file object may be obtained using the **IoQueryInformation** function:

NTSTATUS
IoQueryInformation(
    IN PFILE_OBJECT *FileObject,*
    IN FILE_INFORMATION_CLASS *FileInformationClass,*
    IN ULONG *Length,*
    OUT PVOID *FileInformation,*
    OUT PULONG *ReturnedLength*
    );

Parameters:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileObject - A pointer to the file object for which information is to be returned.

FileInformationClass - The file information class of the type of information that is to be returned.

Length - The length of the *FileInformation* buffer, in bytes.

FileInformation - A buffer to receive the returned information about the file.

ReturnedLength - A variable to receive the length of the information returned in the *FileInformation* buffer.

The **IoQueryInformation** function returns information about a file according to the type of information requested. The types of information that can be requested are defined by the **FILE_INFORMATION_CLASS** data type.

This function performs the same basic function as the **NtQueryInformationFile** system service, but uses a pointer to a file object rather than a handle interface. It must be invoked from kernel mode. It is also used by the **NtQueryObject** object system service to obtain the size of an ACL for a file.

10.44. IoRegisterFileSystem

A file system driver may register itself as an active file system using the **IoRegisterFileSystem** function:

VOID
IoRegisterFileSystem(
    IN OUT PDEVICE_OBJECT *DeviceObject*
    );

Parameters:

DeviceObject - A pointer to the device object that represents the file system.

The **IoRegisterFileSystem** function registers a driver as an active file system. This is accomplished by placing the file system's *DeviceObject* into a LIFO-ordered list of file systems to be searched when a file system is needed to service a device.

This function is invoked by each file system driver. The file systems are placed in various queues depending on the type of file system. For example, disk file systems are placed in a queue that is searched whenever a disk media is to be automatically

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

mounted. Each disk file system driver is queried in turn to determine whether or not the driver recognizes the on-disk structure.

10.45. IoRemoveShareAccess

A file's share access information may be removed for a given open instance using the **IoRemoveShareAccess** function:

VOID
IoRemoveShareAccess(
    IN PFILE_OBJECT *FileObject,*
    IN OUT PSHARE_ACCESS *ShareAccess*
    );

Parameters:

FileObject - A pointer to the file object for the current open request.

ShareAccess - A pointer to the common share access data structure associated with the file being closed.

The **IoRemoveShareAccess** function removes the share access information for the file being closed as described by the *FileObject* parameter. This updates how the file is currently being accessed.

When a file is being closed, the file system uses the **IoRemoveShareAccess** function to update how the file is currently opened. This function updates the **SHARE_ACCESS** structure according to how the file was opened by the specified file object. It is unnecessary to call this function for the last close request on the file. That is, if the file is only accessed through the single, specified file object, the file system need not invoke this function to update the share access information for that file.

10.46. IoRequestDpc

A DPC routine can be queued by using the **IoRequestDpc** function:

VOID
IoRequestDpc(
    IN PDEVICE_OBJECT *DeviceObject,*
    IN PIRP *Irp,*
    IN PVOID *Context*
    );

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

DeviceObject - A pointer to the device object that represents the device for which I/O was requested.

Irp - A pointer to an I/O Request Packet that the DPC routine is to service. This pointer is passed to the driver's DPC routine as one of its arguments.

Context - Supplies a pointer to whatever context the interrupt service routine would like to pass to the DPC routine. This pointer is passed to the driver's DPC routine as one of its arguments.

The **IoRequestDpc** function requests that the DPC routine associated with the *DeviceObject* be invoked at DISPATCH_LEVEL and passed a pointer to the *DeviceObject*, a pointer to the *Irp*, and the *Context* parameter. The device object's DPC entry must have been initialized using the **IoInitializeDpcRequest** I/O function.

10.47. IoSendMessage

A terminal driver may send an "unsolicited input" message to a message port using the **IoSendMessage** function:

VOID
IoSendMessage(
    IN PSTRING *DestinationPort,*
    IN PSTRING *TerminalName*
    );

Parameters:

DestinationPort - The name of the port to which the message is sent.

TerminalName - The name of the terminal on which the unsolicited input occurred.

The **IoSendMessage** function allocates a datagram and sends it to the *DestinationPort.* The datagram contains a message that indicates that unsolicited input occurred on the terminal specified by the *TerminalName* parameter.

This function is invoked by terminal drivers when unsolicited input is encountered to give the system notification that perhaps a user is attempting to log on.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.48. IoSetCompletionRoutine

A driver may set a completion routine address and a context parameter in an I/O Request Packet (IRP) stack location using the **IoSetCompletionRoutine** function:

```
VOID
IoSetCompletionRoutine(
    IN PIRP Irp,
    IN PIO_COMPLETION_ROUTINE CompletionRoutine,
    IN PVOID Context,
    IN BOOLEAN InvokeOnSuccess,
    IN BOOLEAN InvokeOnError,
    IN BOOLEAN InvokeOnCancel
    );
```

Parameters:

Irp - A pointer to the IRP that contains the stack location in which the completion routine is set.

CompletionRoutine - The address of a completion routine to be executed upon completion of the I/O request.

Context - A context parameter that is passed to the completion routine. The driver can use this parameter for any context that it needs.

InvokeOnSuccess - Indicates that the completion routine is to be invoked if the I/O operation completes successfully.

InvokeOnError - Indicates that the completion routine is to be invoked if the I/O operation completed with an error.

InvokeOnCancel - Indicates that the completion routine is to be invoked if the I/O operation is being canceled.

The routine specified by the *CompletionRoutine* has the following type definition:

```
typedef
NTSTATUS
(*PIO_COMPLETION_ROUTINE) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

DeviceObject - Supplies a pointer to the device driver's device object.

Irp - Supplies a pointer to the I/O Request Packet.

Context - Supplies the value of the *Context* parameter specified in the call to the **IoSetCompletionRoutine** function.

The **IoSetCompletionRoutine** stores the address of the completion routine into the current IRP so that when the operation completes the driver can be invoked. If the driver does not wish to be invoked for any of the above reasons, then it need not perform any function calls.

This function is invoked by any layered driver that wishes to be notified when an I/O operation that it has passed to a lower level driver completes.

10.49. IoSetShareAccess

A file's share access information may be set using the **IoSetShareAccess** function:

```
VOID
IoSetShareAccess(
    IN ACCESS_MASK DesiredAccess,
    IN ULONG DesiredShareAccess,
    IN OUT PFILE_OBJECT FileObject,
    OUT PSHARE_ACCESS ShareAccess
    );
```

Parameters:

DesiredAccess - Supplies the types of access that the current open request would like to the file. This is the same desired access parameter given to the file system by the I/O system when the open request is made.

DesiredShareAccess - Supplies the types of shared access that the current open request would like to the file. This is the same shared access parameter given to the file system by the I/O system when the open request is made.

FileObject - A pointer to the file object for the current open request.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ShareAccess - A pointer to the common share access data structure associated with the file being opened.

The **IoSetShareAccess** function sets the initial share access state for a file when it is opened or created for the first time. After the initial file open/create, other requests may be checked against the share access for the file using the **IoCheckSharedAccess** function.

File systems maintain state about files through structures called File Control Blocks **(FCBs)**. The **SHARE_ACCESS** is a structure that describes how the file is currently accessed by all opens. It is contained in the FCB as part of the open file state for the file. The **SHARE_ACCESS** data structure itself should be treated as an opaque data type by file systems and drivers. That is, its contents should only be accessed through the I/O system functions. This allows the structure to change from release to release without having to modify driver source code.

It should be noted that this function provides no synchronization with other updates to the **SHARE_ACCESS** structure. The file system should lock access to the structure by locking its FCB.

This function is used by file systems to set the initial share access for a file.

10.50. IoStartNextPacket

The next packet queued to a driver can be started using the **IoStartNextPacket** function:

VOID
IoStartNextPacket(
    IN PDEVICE_OBJECT *DeviceObject*
    );

Parameters:

DeviceObject - A pointer to the device object that contains the device queue for the device on which the I/O request is performed.

The **IoStartNextPacket** function checks the device queue in the specified *DeviceObject* for an IRP and, if one is found, it is dequeued and passed to the driver's start I/O routine with a pointer to the IRP and a pointer to the *DeviceObject*.

This function is generally invoked by a driver after the processing of the current IRP has been completed by the driver. This allows the driver to start the next operation that is pending for the device.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.51. IoStartPacket

An I/O request can be started on a device using the **IoStartPacket** function:

VOID
IoStartPacket(
    IN PDEVICE_OBJECT *DeviceObject,*
    IN PIRP *Irp,*
    IN PULONG *Key* **OPTIONAL**
    );

Parameters:

DeviceObject - A pointer to the device object for the device on which the request is to be performed.

Irp - A pointer to the I/O Request Packet to be started on the specified device, provided that the device is not already busy.

Key - An optional key value that specifies where in the pending IRP list the *Irp* should be queued if the device is already busy.

The **IoStartPacket** function checks the device queue in the specified *DeviceObject* and either starts the request by passing it to the driver's start I/O routine or queues it to the device's work queue for later processing. If the *Irp* is queued to the device's work queue, then a *Key* may optionally be specified that indicates where in the pending list the request is queued.

For more information on how a packet is actually queued to a device queue, see the **Windows NT Kernel Specification**. In particular, refer to the section on Device Queue Objects.

This function is used by a driver's major function routine to start an operation on a device or to have it queued if the device is already busy.

10.52. IoStartTimer

An initialized one second timer can be started by using the **IoStartTimer** function:

VOID
IoStartTimer(
    IN PDEVICE_OBJECT *DeviceObject*
    );

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

DeviceObject - Specifies the device object whose timer is to be started. The timer must have been initialized using the **IoInitializeTimer** function.

The **IoStartTimer** function starts the timer that was previously initialized by **IoInitializeTimer**. Once the timer has been started, the timer routine is invoked once every second.

This function is used by drivers to time out operations. The timer counter should be initialized either to a negative count if no timed operation is being performed, or to the number of seconds that the operation has to complete if a timed operation is being performed. If the timer routine decrements the counter and it becomes zero, then the operation did not complete in time. If the counter is a negative number, then the routine should not modify it.

10.53. IoStopTimer

A one-second timer can be stopped using the **IoStopTimer** function:

```
VOID
IoStopTimer(
    IN PDEVICE_OBJECT DeviceObject
    );
```

Parameters:

DeviceObject - Specifies the device object whose timer is to be stopped.

The **IoStopTimer** function stops the timer that was previously started by the **IoStartTimer** function. While the timer is then canceled and will not expire again, it is possible for the timer routine to be invoked one more time after the call to this routine has been completed. This is because the timer could have expired and been placed into the queue during a window that cannot be canceled. Furthermore, the timer could have expired on another processor at the same time that the call to this routine was being made.

This function is used by drivers to cancel one second timers. This function is generally only invoked when the driver is being unloaded from the system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.54. IoSynchronousPageWrite

The pager can synchronously write pages of data to the paging file or to a mapped file using the **IoSynchronousPageWrite** function:

```
NTSTATUS
IoSynchronousPageWrite(
        IN PFILE_OBJECT FileObject,
        IN PMDL MemoryDescriptorList,
        IN PLARGE_INTEGER StartingOffset,
        IN PKEVENT Event,
        OUT PIO_STATUS_BLOCK IoStatusBlock
        );
```

Parameters:

FileObject - A pointer to a referenced file object representing the file to write.

MemoryDescriptorList - A Memory Descriptor List (MDL) that describes the locked-down buffer containing the data to write to the file.

StartingOffset - The starting byte offset within the specified file where the write operation is to begin.

Event - Supplies a pointer to a kernel event that to set to the signaled state once the write is complete.

IoStatusBlock - A variable to receive the final completion status and information about the write operation. The number of bytes actually written is returned in the *Information* field.

The **IoSynchronousPageWrite** function gives the memory manager a quick way of building and starting an I/O request to write data to a file. This allows I/O completion to be short circuited for paging I/O.

The function writes the number of bytes specified by the MDL from the buffer described by the MDL, beginning at the *StartingOffset* within the file. The *Event* is set to the signaled state once the operation has completed.

This function is only invoked by the memory manager.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.55. IoUpdateShareAccess

A file system can update the share access for a file using the **IoUpdateShareAccess** function:

VOID
IoUpdateShareAccess(
    IN OUT PFILE_OBJECT *FileObject,*
    IN OUT PSHARE_ACCCESS *ShareAccess*
    );

Parameters:

FileObject - A pointer to the file object for the current open request.

ShareAccess - A pointer to the common share access data structure associated with the file being opened. This structure is treated as an opaque type by drivers.

The **IoUpdateShareAccess** function updates the *ShareAccess* according to the types of access being requested for the current open request. This function may only be invoked if a previous call to **IoCheckShareAccess** succeeded.

This function simply updates the **SHARE_ACCESS** structure maintained for files. It performs the same update functionality as the **IoCheckShareAccess** function but does not perform the check access functionality.

10.56. IoWriteErrorLogEntry

An error log entry buffer may be written to the error log queue using the **IoWriteErrorLogEntry** function:

VOID
IoWriteErrorLogEntry(
    IN OUT PVOID *ErrorLogEntry*
    );

Parameters:

ErrorLogEntry - A pointer to the error log entry buffer that contains the entry data. This entry must have been allocated using the **IoAllocateErrorLogEntry** function.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **IoWriteErrorLogEntry** queues the specified error log entry buffer to the error log thread's database so that the thread can send it to the error log process. The error log process is actually responsible for writing the entry out to the error log file.

This function is used by drivers to post an error log entry.

11. I/O System Folklore

The following sections describe those features of the I/O system that are not fully described in any other documentation. These sections further describe how the I/O system works and how driver writers can use this knowledge and I/O system features to develop robust, high-performance drivers.

11.1. Rules for Completing an I/O Request

A driver needs to complete an I/O request in one of two different situations:

o - The request packet was in error and will not be processed at all. For example, a parameter was invalid for the specified function.

o - The request packet parameters are correct, so the I/O request will be processed.

In the first case, the packet is given to the driver at the appropriate dispatch entry point for the function code in the IRP stack location. For a file system driver, this means that the packet is given to the FSD. Since the packet is in error, there is no reason to return a pending status and then asynchronously complete the request at a later time. The driver dispatch routine can immediately determine that this is the case, so the packet should be aborted as follows:

o - Set the error status in the IRP by writing to the *IoStatus.Status* field of the packet.

o - Raise IRQL to DISPATCH_LEVEL, saving the old IRQL.

o - Invoke the **IoAbortInvalidRequest** function.

o - Lower IRQL to the previous IRQL returned from the raise operation.

o - Return to the caller of the dispatch routine with the same status written to the status field of the IRP.

This sequence causes the I/O system to return the error to the original caller of the system service. The I/O system will not set the caller's file handle or optional event

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

to the Signaled state, and it will not write the caller's I/O status block. This is the definition of an I/O request that was in error. Notice that the driver must still write the status block in the IRP even though it will not be written to the caller's I/O status block. This must be done in case there is a layered driver above the current driver.

In the second case, all of the parameters for the specified function are correct. This means that the I/O request will be processed.

Once the driver has determined that the I/O request is to be completed, it has one of two options:

o - Process the request and complete it immediately.

o - Queue the request to be performed at a later time and return a pending status.

If the request can be immediately processed without causing the current thread to wait, then the driver should do so. An example of such a situation is when the caller has requested information about a file and the information is in memory. The driver can simply place the information into the buffer and complete the request. Notice that an optimization for the user can be made here if the driver returns a status of *STATUS_SUCCESS*. This means that the request was not only successful, but it is actually complete at this point. That is, the file object or event has been set to the Signaled state, the I/O status block has been written, etc.

If the packet is to be queued and completed at a later time, then the driver should queue the packet and return *STATUS_PENDING*.

Once the request completes, then the driver should invoke the normal completion function. In either of the preceding cases, the sequence that the driver uses to complete the request is as follows:

o - Set the appropriate status in the IRP *IoStatus.Status* field.

o - Raise IRQL to DISPATCH_LEVEL, saving the old IRQL.

o - Invoke the **IoCompleteRequest** function.

o - Lower IRQL to the previous IRQL returned from the raise operation.

o - Return to the caller of the dispatch routine with the same status written to the status field of the IRP.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Notice that if the request is queued and processed later it may still incur an error. The **IoCompleteRequest** sequence should still be used by the driver to complete the I/O request.

11.2. Accessing Another Driver

The **Windows NT** I/O system allows drivers to be layered so that one driver may communicate directly with another driver. The upper-level driver may do this either by reusing the same IRP or by passing a new, separate IRP.

In either case, a driver may invoke another driver by using the **IoCallDriver** function. This function takes two parameters:

o - A pointer to the device object for the device upon which the request is to be performed

o - A pointer to the I/O Request Packet itself

To obtain a pointer to the device object the upper level driver must first open the device. This can be done by simply invoking the **NtOpenFile** system service. This service returns a handle to a file object that represents a connection to the device. The file object can then be referenced by invoking the **ObReferenceObjectByHandle** function. This function writes the address of the file object as one of its output parameters. The file object itself contains a pointer to the device object for the device that was opened. This pointer can now be used to reference the device in **IoCallDriver** function calls.

If the driver is either unloading or it is told to close the device through a configuration control function packet, it should perform the following steps:

o - Dereference the file object pointer by invoking the **ObDereferenceObject** function.

o - Close the handle to the file object by invoking the **NtClose** system service.

These steps will cause the appropriate reference counts to be decremented so that the device can be removed from the system if necessary.

11.3. Generating Packets

Most of the time, a driver that would like to communicate with another driver can simply reuse the I/O Request Packet (IRP) that it is given by simply using the next stack location in the IRP. It can do this by invoking the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IoGetNextIrpStackLocation function to get a pointer to the next stack location in the IRP. The driver can then fill in the function and parameter fields and pass the packet to the driver by using the **IoCallDriver** function.

However, there are times when this is insufficient and the driver must allocate a different IRP to pass to the next driver. This happens when the driver implements a request by splitting it into several different parallel requests. There are several different routines that can be used to aid in this situation:

o - **IoAllocateIrp** - This routine simply allocates and initializes an IRP. It is then up to the calling driver to fill in the appropriate header locations as well as the stack location to tell the target driver the function that is to be performed. Obtaining a pointer to the appropriate stack location can be done using the **IoGetNextIrpStackLocation**.

Allocation of the packet is done using the appropriate IRP lookaside list if there are packets available in the system lists. The *StackSize* parameter required by the **IoAllocateIrp** function can be obtained from the *StackSize* field of the target device object.

o - **IoBuildSynchronousFsdRequest** - This function can be used to build a packet that is adequate to request that a target driver perform either a read or a write operation. However, the packet that this function builds is synchronized by an event specified as one of its parameters, so the current thread will have to wait for the event to be set to the Signaled state in order to synchronize the completion of the request. Therefore, it is recommended that these types of packets only be used in cases where the requesting thread is performing a synchronous I/O function.

o - **IoBuildAsynchronousFsdRequest** - This function can be used to build a packet that is adequate to request that a driver perform either a read or a write operation. It is up to the driver that builds the packet to synchronize the completion of the packet by specifying a completion routine for itself. This is done using the **IoSetCompletionRoutine** function. That is, once the packet is built, the driver sets the address of its completion routine before giving it to the next driver. In this way, the driver is notified when the request packet has completed.

This function is generally used by the FSD part of drivers because the specified completion routine is executed in a thread-independent context.

o - **IoBuildFspRequest** - This function can be used to build a packet that is adequate to request that a driver perform either a read or a write operation. It

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

is generally used by the FSP part of a driver because synchronization of the packet is performed through either a kernel event or an APC routine. The FSP must supply one of these two parameters as its synchronization mechanism or it will not be able to determine when the packet is complete.

\\ *There will certainly be other routines that can be used to generate packets that will be added before the system ships. These routines will be added as needed. It is not recommended that drivers actually use the* **IoAllocateIrp** *interface and then generate everything by hand, however there is no better way to do this today unless one of the other routines provides the exact functionality needed by the driver.*\\

11.4. Direct vs. Buffered vs. Neither I/O

The **Windows NT** I/O system provides drivers with a choice of three different methods for implementing I/O operations. These are as follows:

1. Direct I/O - Direct I/O refers to the capability to perform I/O directly into the caller's buffer. That is, the I/O system will set up the necessary data structures to allow the I/O operation to be performed directly into the caller's buffer. The driver writer specifies that this type of I/O is desired by setting the *DO_DIRECT_IO* flag in the *Flags* field of the device object.

   If this flag is set, the I/O system performs the following operations before passing the IRP to the driver:

   o - The caller's buffer is probed for the appropriate access according to whether the request being performed is a read or a write operation.

   o - The caller's buffer will be locked into memory so that the physical memory backing the buffer cannot be reused for some other operation.

   o - An MDL will be built that describes the user's buffer.

   o - The *MdlAddress* field of the IRP will be set to point to the MDL that was built.

   A driver might do direct I/O for two different reasons. It will use this type of I/O if a device that it is servicing performs DMA I/O. The MDL can be used in a call to the **IoMapTransfer** function to map the caller's buffer so that when the DMA controller reads from or writes to memory the appropriate locations will be read or written.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

A driver might also use direct I/O if it needs to gain direct access to the caller's buffer but will not be executing in the context of the caller. For example, an FSP thread might need to copy data directly into the caller's buffer but, by definition, does not have direct access to it. The **MmMapLockedPages** function must be used to map the caller's buffer into the FSP thread's virtual address space by passing it a pointer to the MDL. This temporarily allows direct access to those physical pages backing the caller's buffer. Once the copy operation has completed, the **MmUnmapLockedPages** function can be used to unmap the caller's buffer.

2. Buffered I/O - Buffered I/O refers to the capability to perform I/O operations to an intermediate buffer that contains a copy of the data from the caller (write operation) or a copy of the data that is to be copied back to the caller's buffer (read operation) when the request is complete.

This type of I/O is generally used when a device that is being serviced cannot perform DMA I/O directly into a buffer. It is also used when keeping the caller's buffer locked for an extended period could cause system resources to be depleted.

A driver may specify that it performs buffered I/O by setting the *DO_BUFFERED_IO* flag in the *Flags* field of its device object. When this flag is set, the I/O system performs the following operations to set up the caller's buffer before passing the IRP to the driver:

o - The caller's buffer is probed for the appropriate access according to whether the request being performed is a read or a write operation.

o - A sufficiently large buffer is allocated from system non-paged pool to handle all of the data being read or written.

o - If the operation is a write, the data in the caller's buffer is copied into the allocated system buffer. If the operation is a read, the *IRP_INPUT_OPERATION* flag is set in the IRP flags field so that the contents of the system buffer will be copied into the caller's buffer after the operation has completed. The *IRP_DEALLOCATE_BUFFER* flag is also set in the IRP flags field so that the buffer will be deallocated after the copy operation is complete.

o - The *AssociatedIrp.SystemBuffer* field of the IRP is set to point to the allocated system buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Once the operation completes, the system buffer is automatically deallocated by the I/O system.

3. Neither I/O - Under certain circumstances, a driver might postpone specifying either direct I/O or buffered I/O until it has a chance to determine which type of operation is appropriate, based on whether data is immediately available or whether the data must be obtained from elsewhere asynchronously. For this case, the driver sets neither of the flags in the device object *Flags* field. It is then up to the driver to perform the necessary steps to allow either direct or buffered I/O to be performed. Notice that for some cases, the driver may not have to perform either type of I/O operation and can simply copy the data directly into the user's buffer.

If this type of I/O operation is specified by the device object, the I/O system performs the following steps before passing the IRP to the driver:

o - The caller's buffer is probed for the appropriate access according to whether the request being performed is a read or a write operation.

o - A pointer to the caller's buffer is passed in the *UserBuffer* field of the IRP.

If the driver determines that the data is immediately available and wants to copy it directly into the caller's buffer, then it does so by using an exception handler around the code that performs the copy. This is done to catch access violations that occur when another thread executing in the same process changes the virtual address space described by the caller's buffer. This may also occur because of a kernel APC being executed in the context of the current thread.

If the data is not immediately available, then the driver may wish to perform either direct or buffered I/O. It must perform the same steps that the I/O system does to setup the appropriate structures so that the I/O can be completed normally. Any deviation from the exact setup can cause the system to crash.

It is also possible for a device driver to specify a preallocated driver buffer that contains data to be copied into the caller's buffer. This can be accomplished by performing the following steps:

o - Specify Neither I/O in the device object *Flags* field, setting neither of the other two device object flags.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - Set the *AssociatedIrp.SystemBuffer* field to point to the preallocated driver buffer.

o - Set the *IRP_INPUT_OPERATION* flag in the IRP flags field, but not set the *IRP_DEALLOCATE_BUFFER* flag in the IRP flags field.

These three types of I/O are used for all **NtReadFile** and **NtWriteFile** operations. Most other NT API services use buffered I/O almost exclusively, except for the **NtDeviceIoControlFile** and **NtFsControlFile** system services. These services pass their buffers according to the *method bits* in the I/O control code. More detailed information is contained in the *Windows NT IRP Language Definition* specification.

11.5. Building Virtually Discontiguous Buffers

There are currently no **Windows NT** APIs in the I/O system that allow callers to provide more than one input or output buffer. This keeps the design of the I/O system as simple as possible and causes I/O completion to execute more quickly. Since no complex I/O user buffer state is required, there is at most only one copy operation that takes place during I/O completion.

However, layered drivers, such as network drivers, may need to provide each other with more than one virtually discontiguous buffer. A transport driver may wish to add a transport header to the front of a user data buffer. A datalink driver may wish to put another header in front of the transport's header, and so on. Rather than having each driver allocate a buffer, place its data into the buffer and then copy all of the previous data after its own data, it is much more efficient to simply insert a virtual buffer descriptor in front of the current data descriptor.

The data structures that represent these virtual buffer descriptors in **Windows NT** are *Memory Descriptor Lists* **(MDLs)**. Each MDL describes the physical pages that make up a single virtually contiguous buffer. By chaining MDLs through the structure's *Next* pointer, virtually discontiguous buffers may be specified in different driver layers.

During I/O completion, if the drivers do not run down the MDLs, then the I/O system will provide this functionality automatically. That is, all MDLs chained together from the IRP's *MdlAddress* field will automatically be deallocated and the pages described by those MDLs will be unlocked. If the driver that specified the MDL needs to perform its own buffer management, then it should deal with this by unlinking its MDL from the chain in its I/O completion routine.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

11.6. I/O Services Synchronization

The **Windows NT** I/O system provides many services to users. Among these services are those that may be completed synchronously as well as those that may be completed asynchronously. This section explains how the I/O system services actually work to provide these capabilities to users, even though the I/O system's design is based on an asynchronous model. This section provides background information to enable driver writers to better understand the environment in which their driver is executing.

The I/O system services that complete asynchronously may either be invoked as asynchronous system services, or they may be invoked to execute synchronously. (The latter is the case when the file that the services are operating on was opened with one of the *FILE_SYNCHRONOUS_IO* options.) For those services that are asynchronous and are invoked to complete as such, no special processing is required. Once the packet is given to the driver, the I/O system simply returns to the caller.

However, a user may open a file using one of the synchronous I/O options. For this case, all services, synchronous and asynchronous, perform the steps outlined below to synchronize access to the file. Note that these options also cause all I/O operations on the file to be serialized.

o - The parameters for the service are probed, captured, and validated.

o - The file object is referenced by calling the object manager with the caller's file handle.

o - The semaphore associated with the file object is then waited on in either an alertable or non-alertable manner, depending on which synchronous I/O option was used in the open or create call.

o - The service calls the driver normally and then waits for the file object itself to be set to the Signaled state. This is a special case where the I/O completion routines will set the file object to the Signaled state along with an event, if one was specified. The other special operation performed is to copy the I/O status into the file object itself so that the service can return it to the caller without having to touch the caller's I/O status block. This makes special error recovery code unnecessary when the address space for the I/O status block has been deleted while the driver was servicing the request.

o - The semaphore is released and the service returns to the caller.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Some services that are synchronous must also deal with the asynchronous I/O system, even when the user did not open the file specifying one of the synchronous I/O options. These services perform the following steps to make it appear as if the I/O system is synchronous:

o - The parameters for the service are probed, captured, and validated.

o - The file object is referenced by calling the object manager with the caller's handle.

o - A local kernel event variable is initialized to the Not-Signaled state and used as the user-specified event in the IRP. A local I/O status block variable is used as the user-specified I/O status block and its address is placed in the IRP.

o - The *IRP_SYNCHRONOUS_API* flag is set in the flags field of the IRP. This flag informs the I/O completion code that the event is a kernel event, rather than a normal user object system event, and should not, therefore, be dereferenced during completion.

o - The service calls the driver.

o - If the return status from the driver is *STATUS_PENDING*, then the service waits for the local kernel event to be set to the Signaled state.

o - The local I/O status block contents are copied to the caller's I/O status block, and the status field is returned as the final status from the service.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

12. Revision History

Original Draft 1.0, March 21, 1989

Changes from I/O specification revision 1.2, from which this draft was spawned:

- Redo IRP stacks; single routine; parameters.
- Add new I/O APIs.
- Fill in sections on APIs and data structures.

Revision 1.1, February 12, 1990

- Brought spec up to current design level.
- Fixed lots of typos and grammatical errors.
- Removed **IoAllocateAdapterAndChannel**.
- Removed **IoIsRequestComplete**.
- Added **IoAbortInvalidRequest**.
- Added **IoAllocateAdapterChannel**.
- Added **IoAllocateController**.
- Added **IoDeallocateAdapterChannel**.
- Added **IoDeallocateController**.
- Added **IoDeallocateMdl**.
- Added **IoGetRelatedDeviceObject**.
- Added **IoGetRequestorProcess**.
- Added **IoMapTransfer**.
- Added **IoGetAttachedDevice**.
- Fixed IN, OUT, and IN OUT in IoXxx calls.
- Added I/O system folklore section.

Revision 1.2, July 20, 1990

- Add device object pointer to timer routine.
- Updated share access manipulation routines to reflect latest design.
- Added **IoDeallocateIrpAtDispatchLevel**.
- Added **IoFlushAdapterBuffers**.
- Added **IoIsOperationSynchronous**.
- Added **IoCreateStreamFile**.
- Replaced **IoPageWrite** with new routines.
- Added description of stream file objects.
- Updated cancel description to reflect latest design.
- Updated timer descriptions to match latest design.
- Updated description of I/O completion w/o using PFN mutex.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision 1.3, xxxx ??, 1990

- Update description of unlocking pages during completion
- Document **IoIsOperationSynchronous**.
- Remove **IoDeallocateIrpAtDispatchLevel**.
- Modify **IoAsynchronousPageWrite** to use 64-bit offset.
- Modify **IoBuildAsynchronousFsdRequest** to use 64-bit offset.
- Modify **IoBuildFspRequest** to use 64-bit offset.
- Modify **IoBuildSynchronousFsdRequest** to use 64-bit offset.
- Modify **IoPageRead** to use 64-bit offset.
- Modify **IoSynchronousPageWrite** to use 64-bit offset.
- Removed **IoQueryAcl** function.
- Removed **IoSetAcl** function.
- Remove access parameters from **IoUpdateShareAccess**.
- Changed references from SUCCESS to NT_SUCCESS.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Design Workbook Introduction

Author: *Lou Perazzoli*

Original Draft 1.0, March 31, 1989
Revision 2.0, May 5, 1989
Revision 3.0, August 17, 1989
Revision 4.0, October 15, 1989
Revision 5.0, January 15, 1990
Revision 6.0, July 25, 1990

Hardcopy released to The Smithsonian Institute.

Digital copy released to Universities for non-commercial academic use under the Windows Research Kernel License.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction.. 1

2. Project Goals.. 1

3. NT OS/2 Components.. 2

4. Functional Specifications.. 3

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

The **NT OS/2** system is a portable implementation of OS/2 developed in a high-level language. The initial release of **NT OS/2** is targeted for Intel 860-based hardware, which includes both personal computers (*Frazzle*) and servers (*Dazzle*).

The first systems based on a RISC microprocessor will be available for testing in the fall of 1990.

2. Project Goals

The ultimate goal of the **NT OS/2** project is to develop a portable implementation of OS/2 executing on the Intel 860 and to establish this combination of hardware and software as the standard for high-performance personal computers and server systems.

NT OS/2 has the following overall project goals (though not all these goals will be attained by the first implementation of **NT OS/2**):

- o Portability to a variety of hardware architectures. Though the first implementation is targeted to the Intel 860, the overall system design isolates the machine-dependent portions for portability to other architectures.

- o Support for multiple processors with shared memory via symmetric multiprocessing. This provides performance improvements for multiprocessor workstations and servers.

- o Compatibility with the OS/2 V2.0 32-bit application programming interface (API). Because the initial target system is not an Intel x86 architecture, all applications will have to be recompiled and relinked. In addition, any assembly language code will have to be rewritten or converted to a higher level language such as C.

- o Security at the C2 level with future versions achieving higher levels of security. This includes login/logout options on the personal computer and the server system, and declaration and enforcement of protection attributes for shareable resources (files, IPC, memory objects, etc.).

- o Support for a POSIX-compliant API interface that passes the POSIX validation suites.

- o Support for internationalization.

- o Support for LANMAN networking and management of personal computers and servers.

- o Support for the current Presentation Manager API running in both the OS/2 environment and the POSIX environment.

- o Support for distributed applications. The network is integrated into the system to allow transparent distribution of applications and services within a network.

- o Support for object-oriented file systems and object-oriented presentation manager.

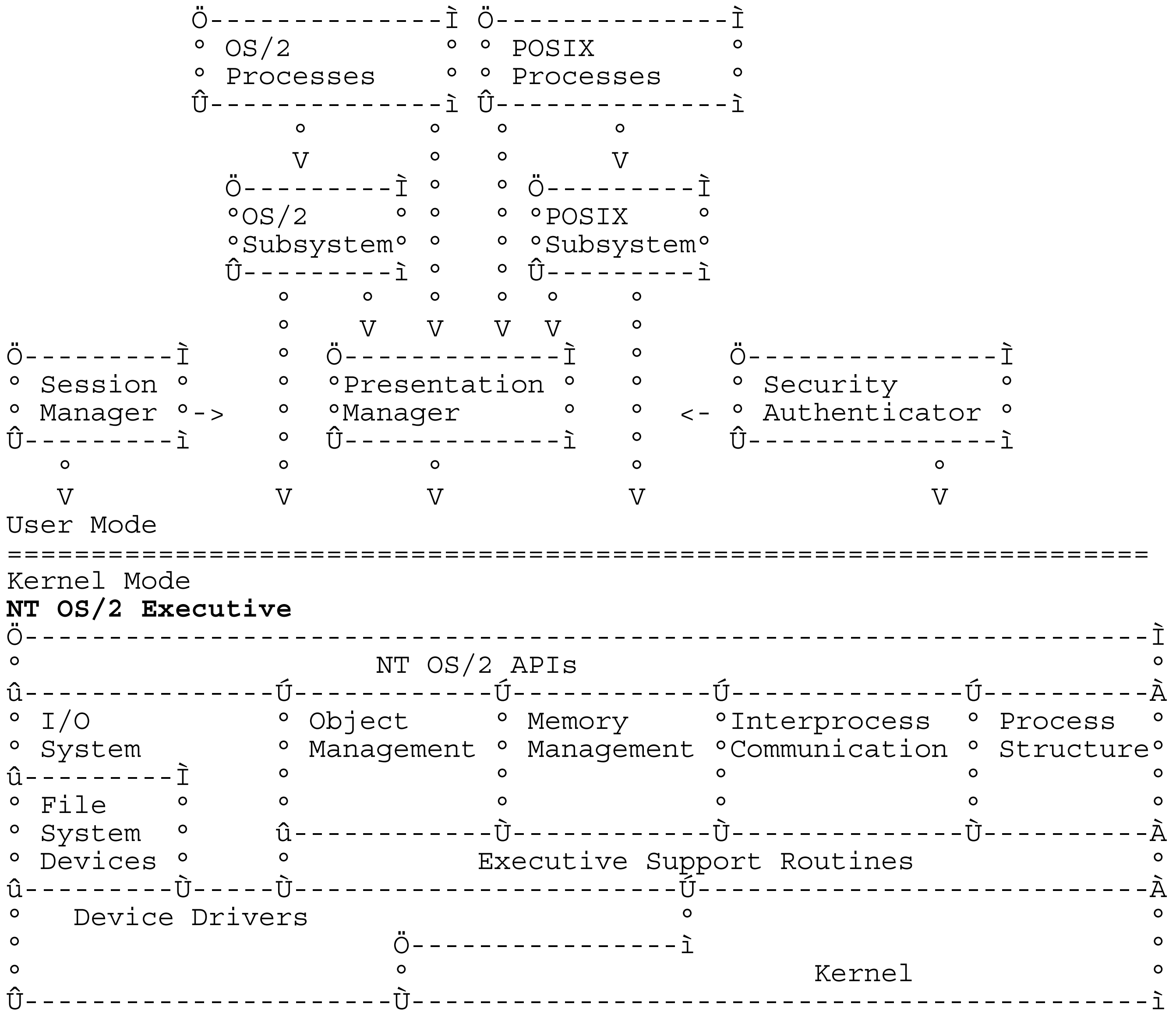
Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Easy extensibility by layering new features on the existing system without modifying the underlying system.
- o Simultaneous execution by multiple users, each with a unique security profile.
- o Interoperability and data interchange between OS/2 and POSIX applications.
- o High reliability that prevents errant user programs from causing a system crash or exhausting system-wide resources. Resource quotas, a protected kernel, and protected objects are used to improve reliability.

3. NT OS/2 Components

NT OS/2 consists of a highly integrated kernel / executive that executes in kernel mode. It provides the necessary services to allow the emulation of OS/2 and POSIX APIs via protected subsystems executing in user mode. Both the OS/2 and POSIX subsystems provide these services through remote procedure calls from a client to the server subsystem. The server subsystem, in turn, emulates the desired operation locally or by calling the executive, and returns the results to the caller. The following diagram illustrates the structure of **NT OS/2**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License



Block Diagram of NT OS/2:

4. Functional Specifications

The following specifications are contained within this design workbook. Each specification contains an abstract of the component it describes, how that component fits into the system, the various APIs that are used to access the functionality, and enough detail to ensure the defined capability can be implemented.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The goal of the specifications is to allow someone to understand the functionality provided by a particular piece of the system. It is NOT a goal to describe the actual implementation.

Each specification addresses Cruiser and POSIX compatibility, if appropriate. The following is a list of design specifications included in this version of the workbook:

1. Kernel - Describes the function of the kernel, the objects implemented, and the various interfaces provided to manipulate these objects. This specification contains implementation details, where necessary, to reveal how multiprocessing and processor dispatching take place. This specification also describes synchronization, scheduling/dispatching, and Asynchronous Procedure Calls (APCs).

2. Object Management - Describes how the executive deals with objects, what they are for, how they are protected, how they are named, how they are allocated, how they are accounted for, and how they are deleted. This specification also addresses object directories and how to access them using the file system directory operations.

3. Process Structure - Describes the process and thread objects and the operations that can be performed on them. This specification also explains signals and how OS/2 compatibility and POSIX compliance are addressed.

4. Virtual Memory - Describes the virtual memory objects and the operations that can be performed on these objects.

5. I/O Management - Describes the APIs and objects available for I/O operations.

6. Security - Describes how security is provided in the system, the ACL format, ACL access checking rules, login/logout, the authorization file, and the partial closure of covert channels. This specification also describes audit and alarm logging.

7. Local Process Communication - Describes the client/server protected subsystem model, client impersonation, port objects, and connection/disconnection operations.

8. Remote Procedure Call - Describes a transport-independent interface to remote procedure calls.

9. Session Manager - Describes how the subsystems for OS/2 and POSIX are created, and how they interact with each other.

10. File System - Describes the file systems, how they are put together, the functions they perform, and how they accomplish the tasks that they are given.

11. Semaphores and Events - Describes the APIs and objects available for synchronization.

12. Argument Validation - Describes the argument probing and capture requirements for system services.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

13. Timers - Describes the timer object, which is used to mark time, and the functions available to manipulate it.
14. Coding Guidelines - Describes the naming and structure of **NT OS/2** code.
15. LAN Manager Software - Describes the network capabilities of the system, how network drivers fit together, and how the protocol stacks are managed.
16. Exceptions - Describes the dispatching of hardware exceptions to the condition dispatcher and the arguments that accompany each exception. It also explains guard page handling, automatic stack expansion, and access violations on the user stack, as well as how signals are handled at the user level.
17. OS/2 Emulation Subsystem - Describes the requirements and methods used to design and build the OS/2 emulation subsystem.
18. Status values - Describes the format for status values return by **NT OS/2** APIs.
19. Subsystem Design Rational - Describes the rationale for designing OS/2 and POSIX emulation as subsystems as opposed to supporting the APIs directly in the executive.
20. Shared Resource Specification - Describes the routines that implement multiple-readers, single-writer access to a share resource.
21. Executive Support Routines - Describes executive support routines which are available in kernel mode and not documented in other chapters.
22. Driver Model - Describes the device driver model, how I/O is managed throughout the system, how the file system and network capabilities fit into the system, and the objects and operations that are available to help manage the I/O system. It also presents I/O validation, queueing, page lockdown, double mapping, I/O completion, and error logging.
23. POSIX Emulation Subsystem - Describes the requirements and methods used to design and build the POSIX emulation subsystem.
24. Time Conversion Specification - Describes the APIs available for viewing time and converting to and from different formats.
25. Mutant Specification - Describes the mutant object and services which operate upon the object.
26. Transport Driver Interface - Describes the interface for the network transport layer.
27. Network Driver Interface Specification - Describes the interface for the network physical layer.
28. Lan Manager Server - Describes the design of the Lan Manager server and the operations supported.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

29. C Structured Exception Handling - Describes the extensions to C in the MS 860 compiler to support structured exception handling.

30. NT C User's Guide - Describes the command syntax and language issues for the MS 860 C compiler.

31. Prefix Table - Describes the prefix table package.

32. System Startup Design Note - Describes system startup after phase one initialization.

33. Debug Architecture - Describes the debug architecture for NT OS/2.

34. Linker/Librarian - Describes the NT OS/2 linker, librarian, and image format.

35. Caching Design Note - Describes the system-wide file caching implementation.

36. Utility Design Specification - Describes the basic support routines for NT OS/2 utilities.

37. OS/2 Environment Subsection Security - Describes the security features of the OS/2 environment subsystem.

38. Security Account Manager Protected Server - Describes the security account manager which maintains user and group account information.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, March 31, 1989

Revision 2.0, May 5, 1989

1. The following specifications were added to the design workbook:

    o Local Process Communication

    o File Systems

    o Session Manager

    o Semaphores and Events

    o Argument Validation

    o Timers

    o Coding Guidelines

2. The in-progress specification list was changed to add the OS Emulation Environment specification.

3. The block diagram was modified.

Revision 3.0, August 17, 1989

1. The following specifications were added to the design workbook:

    o Subsystem Design Rationale

    o Status Codes

    o Shared Resources

    o Executive Support Routines

    o User-mode Interlocked and Fast Lock Routines

    o OS/2 Subsystem Emulation

Revision 4.0, October 15, 1989

1. The following specifications were added to the design workbook:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o POSIX Subsystem Emulation
- o Time Conversion Specification

2. The User-mode Interlocked and Fast Lock Routines Specification was dropped because the APIs were non-portable.
3. The specification list was revised to match the actual specifications in the workbook.
4. The File System Specification was replaced by the File System Design Note.
5. The I/O System Specification was broken into two separate specifications:
    - o I/O System Specification - Documents I/O system API
    - o Driver Model Specification - Documents drivers and I/O system internals

Revision 5.0, January 15, 1990.

1. The following specifications were added to the design workbook:
    - o Mutant Specification
    - o Transport Driver Interface
    - o Physical Driver Interface
    - o Lan Manager Server
    - o C Structured Exception Handling
    - o NT C User's Guide

Revision 6.0, July 25, 1990.

1. The following specifications were added to the design workbook:
    - o System Startup Design Note
    - o Debug Architecture
    - o OS/2 Linker/Librarian/Image Format Specification
    - o Caching Design Note
    - o Utility Design Specification

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- OS/2 Environment Subsystem Security
- Security Account Manager Protected Server

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Exception Handling Specification

Author: *David N. Cutler*

Original Draft 1.0, May 22, 1989
Revision 1.1, June 2, 1989
Revision 1.2, June 6, 1989
Revision 1.3, August, 4, 1989
Revision 1.4, August, 15, 1989
Revision 1.5, November 7, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction.. 1

2. Goals 1

3. Exception Architecture.. 2
   3.1 Frame-Based Exception Handlers.. 2
   3.2 Exception Dispatching 3
   3.3 Exception Handling and Unwind 4
   3.4 Exception Record 4
   3.5 Exception Context... 6

4. Hardware-Defined Exceptions 7
   4.1 Access Violation 8
   4.2 Breakpoint 8
   4.3 Data-Type Misalignment 8
   4.4 Floating Divide By Zero 8
   4.5 Floating Overflow 9
   4.6 Floating Underflow 9
   4.7 Floating Reserved Operand.. 9
   4.8 Illegal Instruction... 9
   4.9 Privileged Instruction 9
   4.10 Invalid Lock Sequence... 10
   4.11 Integer Divide By Zero 10
   4.12 Integer Overflow 10
   4.13 Single Step.. 10

5. Windows NT Software-Defined Exceptions.................................. 11
   5.1 Guard Page Violation.. 11
   5.2 Page Read Error 11
   5.3 Paging File Quota Exceeded.. 11

6. Standard Exception Handling 11
   6.1 Alignment Faults... 12
   6.2 IEEE Floating Faults 12

7. Exception Handling Interfaces.. 12
   7.1 Exception Dispatcher 12
   7.2 Exception Handler.. 13
   7.3 Raise Exception.. 15
   7.4 Continuing From An Exception 15
   7.5 Unwinding From An Exception... 16
   7.6 Last Chance Exception Handling... 18

8. OS/2 2.0 Compatibility... 18

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

8.1 Windows NT Intel i860 Implementation 19
8.2 OS/2 2.0 Intel x86 Implementation 19
8.3 Windows NT Implementation of OS/2 Capabilities.................. 20

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the exception handling capabilities of **Windows NT**. An *exception* is an event that occurs during the execution of a program which requires the execution of software outside the normal flow of control.

Exceptions can result from the execution of certain instruction sequences, in which case they are initiated by hardware. Other conditions may arise as the result of the execution of a software routine (e.g., an invalid parameter value), and are therefore initiated explicitly by software.

When an exception is initiated, a systematic search is performed in an attempt to find an *exception handler* that will dispose of (handle) the exception.

An exception handler is a function written to explicitly deal with the possibility that an exception may occur in a certain sequence of code.

Exception handlers are declared in a language-specific syntax and are associated with a specific scope of code. The scope may be a block, a set of nested blocks, or an entire procedure or function.

Microsoft compilers for **Windows NT** adhere to a common calling standard which enables exception handlers to be *established* and *disestablished* in a very efficient manner.

> \ *The initial hardware target for* **Windows NT** *is the* **Intel i860**, *and therefore, the* **Microsoft** *C compiler for the* **i860** *will be the first compiler that conforms to the required calling standard. As* **Windows NT** *is ported to other architectures, compilers will be required to implement a calling standard that is functional enough to support the* **Windows NT** *exception handling capabilities.* \

Exception handling capabilities are an integral and pervasive part of the **Windows NT** system. They enable a very robust implementation of the system software. It is envisioned that ISVs, application writers, and third-party compiler writers will see the benefits of exception handling capabilities and also use them in a pervasive manner.

2. Goals

The goals of the **Windows NT** exception handling capabilities are the following:

- o Provide a single mechanism for exception handling that is usable across all languages.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Provide a single mechanism for the handling of hardware-, as well as software-generated exceptions.
- o Provide a single exception handling mechanism that can be used by privileged, as well as nonprivileged software.
- o Provide a single mechanism for the handling of exceptions and for the capabilities necessary to support sophisticated debuggers.
- o Provide an exception handling architecture with the capabilities necessary to emulate the exception handling capabilities of other operating systems (e.g. OS/2 and POSIX).
- o Provide an exception handling mechanism that is portable and which separates machine-dependent from machine-independent information.
- o Provide an exception handling mechanism that supports the structured exception handling extensions being proposed by **Microsoft** for the C language (see Structured Exception Handling in C by Don MacLaren, May 10, 1989).

3. Exception Architecture

The overall exception architecture of **Windows NT** encompasses the process creation primitives, system service emulation subsystems, the **Microsoft** calling standard(s), and system routines that raise, dispatch, and unwind exceptions.

Two optional exception ports may be specified when a process is created. These ports are called the debugger port and the system service emulation subsystem port.

When an exception is initiated, an attempt is made to send a message to the recipient process's debugger port. If there is no debugger port, or the associated debugger does not handle the exception, then a search of the current thread's call frames is conducted in an attempt to locate an exception handler. If no frame-based handler can be found, or none of the frame-based handlers handle the exception, then another attempt is made to send a message to the recipient process's debugger port. If there is no debugger port, or the associated debugger does not handle the exception, then an attempt is made to send a message to the recipient process's system service emulation subsystem port. If there is no subsystem port, or the subsystem does not handle the exception, then the system provides default handling based on the exception type.

Thus the search hierarchy is:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Debugger first chance
2. Frame-based handlers
3. Debugger second chance
4. Emulation subsystem

The purpose of this architecture is to provide a very robust exception architecture, while at the same time allow for the emulation of the exception handling capabilites of various operating system environments (e.g., OS/2 2.0 exception handling, POSIX signals, etc.).

Throughout this document, explanations concerning the implementation of the **Windows NT** exception architecture are given referring to the **Intel i860**. It should not be inferred that the described implementation is the only possible implementation, and in fact, the actual implementation on other hardware architectures may be different.

3.1 Frame-Based Exception Handlers

An exception handler can be associated with each call frame in the procedure call hierarchy of a program. This requires that each procedure or function that either saves nonvolatile registers or establishes an associated exception handler, have a call frame.

Microsoft compilers for **Windows NT** adhere to a standard calling convention for the construction of a call frame. A call frame for the **Intel i860** contains the following:

1. A register save mask that describes the nonvolatile registers saved in the frame. These registers are saved in a standard place relative to the frame pointer.
2. Two flags that specify whether an extended register save mask and/or exception handler address is present in the frame.
3. An optional extended register save mask that describes the volatile registers saved in the frame. These registers are saved in a standard place relative to the frame pointer.
4. An optional address of an exception handler that is associated with the frame.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Call frames for other architectures contain similar information. The exact details of the call frame layout are described in the **Microsoft Windows NT** calling standard(s).

3.2 Exception Dispatching

When a hardware exception occurs, the **Windows NT** trap handling software gets control and saves the hardware state of the current thread in a *context record*. The reason for the trap is determined, and an *exception record* is constructed which describes the exception and any pertinent parameters. Executive software is then called to dispatch the exception.

If the previous processor mode was kernel, then the exception dispatcher is called to search the kernel stack call frames in an attempt to locate an exception handler. If a frame-based handler cannot be located, or no frame-based handler handles the exception, then **KeBugCheck** is called to shut down system operation. Unhandled exceptions emanating from within privileged software are considered fatal bugs.

If the previous processor mode was user, then an attempt is made to send a message to the associated debugger port. This message includes the exception record and the identification of the client thread. The debugger may handle the exception (e.g., breakpoint or single step) and modify the thread state as appropriate, or not handle the exception and defer to any frame-based exception handlers found on the user stack.

If the debugger replies that it has handled the exception, then the machine state is restored and thread execution is continued. Otherwise, if the debugger replies that it has not handled the exception, or there is no debugger port, then executive software must prepare to execute the exception dispatcher in user mode.

If the debugger does not dispose of the exception, then stack space is allocated on the user stack, and both the exception record and the context record are moved to the user stack. The machine state of the thread is modified such that thread execution will resume in code that is part of the executive, but it executes in user mode.

The machine state is restored and execution of the thread is resumed in user mode within executive code that calls the exception dispatcher to search the user stack for an exception handler. If a frame-based handler handles the exception, then thread execution is continued. Otherwise, if no frame-based handler is found, or no frame-based handler handles the exception, then the **NtLastChance** system service is executed.

The purpose of the **NtLastChance** system service is to provide the debugger a second chance to handle the exception and to provide the system service emulation

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

subsystem associated with the thread's process, if any, a chance to perform any subsystem-specific exception processing. A second attempt is made to send a message to the associated debugger port. This message includes the exception record and the identification of the client thread. The debugger may handle the exception (e.g., query the user and receive a disposition) and modify the thread state as appropriate, or not handle the exception and defer to the system service emulation subsystem associated with the thread's process.

If the debugger replies that it has handled the exception, then the machine state is restored and thread execution is continued. Otherwise, if the debugger replies that it has not handled the exception, or there is no debugger port, then an attempt is made to send a message to the associated subsystem. This message includes the exception record and the identification of the client thread. The subsystem may handle the exception and modify the thread state as appropriate, or not handle the exception and defer to any default handling supplied by the executive.

If the subsystem replies that it has handled the exception, then the machine state is restored and thread execution is continued. Otherwise, the executive provides default handling of the exception, which is most cases causes the thread to be terminated.

3.3 Exception Handling and Unwind

During the dispatching of an exception, each frame-based handler is called specifying the associated exception and context records as parameters. The exception handler can handle the exception and continue execution, not handle the exception and continue the search for an exception handler, or handle the exception and initiate an unwind operation.

Handling an exception may be as simple as noting an error and setting a flag that will be examined later, printing a warning or error message, or taking some other overt action. If execution can be continued, then it may be necessary to change the machine state by modifying the context record (e.g., advance the continuation instruction address).

If execution can be continued, then the exception handler returns to the exception dispatcher with a status code that specifies that execution should be continued. Continuing execution causes the exception dispatcher to stop its search for an exception handler. The machine state from the context record is restored and execution is continued accordingly.

If execution of the thread cannot be continued, then the exception handler usually initiates an unwind operation by calling a system-supplied function specifying a target call frame and a continuation address. The unwind function walks the stack

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

backwards searching for the target call frame. As it walks the stack, the unwind function calls each exception handler that is encountered to allow it to perform any cleanup actions that may be necessary (e.g., release a semaphore, etc.). When the target call frame is reached, the machine state is restored and execution is continued at the specified address.

3.4 Exception Record

An *exception record* describes an exception and its associated parameters. The same structure is used for both hardware-, and software-generated exceptions.

An exception record has the following structure:

Exception Record Structure

NTSTATUS *ExceptionCode* - The status code that specifies the reason for the exception.

ULONG *ExceptionFlags* - A set of flags that describes attributes of the exception.

Exception Flags

EXCEPTION_NONCONTINUABLE - The exception is not continuable, and any attempt to continue will cause the exception **STATUS_NONCONTINUABLE_EXCEPTION** to be raised.

EXCEPTION_UNWINDING - The exception record describes an exception for which an unwind is in progress.

EXCEPTION_EXIT_UNWIND - The exception record describes an exception for which an exit unwind is in progress.

EXCEPTION_STACK_INVALID - The user stack was not within the limits specified by the Thread Environment Block (**TEB**) when the exception was raised in user mode. Alternately, during the trace backwards through the call frames on the user (or kernel) stack, a call frame was encountered that was not within the stack limits specified by the **TEB** (or within the kernel stack limits), or a call frame was encountered that was unaligned.

EXCEPTION_NESTED_CALL - The exception record describes an exception that was raised while the current exception handler was active, i.e., a nested exception is in progress and the current handler was also called to handle the previous exception.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PEXCEPTION_RECORD *ExceptionRecord* - An optional pointer to an associated exception record. Exception records can be chained together to provide additional information when nested exceptions are raised.

PVOID *ExceptionAddress* - The instruction address at which the hardware exception occurred or the address from which the software exception was raised.

ULONG *NumberParameters* - The number of additional parameters that further describe the exception and immediately follow this parameter in the exception record.

ULONG *ExceptionInformation[NumberParameters]* - Additional information that describes the exception.

The *EXCEPTION_NONCONTINUABLE* bit in the exception flags field is the only flag that can be set by the user. The remaining flags are set by system supplied software as the result of dispatching an exception or the unwinding of call frames.

3.5 Exception Context

A *context record* describes the machine state at the time an exception occurred. This record is hardware architecture dependent and is not portable. Therefore, in general, software should not use the information contained is this record. Hardware-architecture-dependent code such as math libraries, however, can make use of this information to optimize certain operations.

For a hardware-initiated exception, the context record contains the complete machine state at the time of the exception. For a software-initiated exception, the context record contains the machine state at the time the exception was raised by software.

The context record is constructed so that it has an identical format to the call frames generated by the **Microsoft** compilers for the **Intel i860**. The context record for the **Intel i860** has the following structure:

Context Record Structure

ULONG *ContextFlags* - A set of flags that describes which sections of the context record contain valid information.

Context Flags

CONTEXT_CONTROL - The *Psr*, *Epsr*, *Fir*, *IntR1*, *IntFp*, and *IntSp* fields of the context record are valid.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

CONTROL_FLOATING_POINT - The *FltF2...FltF31* and *Fsr* fields of the context record are valid.

CONTEXT_INTEGER - The *IntR4...IntR31* fields of the context record are valid.

CONTEXT_PIPELINE - The *AddStageX*, *MulStageX*, *FldStageX*, *IntResult*, *Kr*, *Ki*, *Merge*, *T*, *Fsr1, Fsr2, and Fsr3* fields of the context record are valid.

ULONG *Fsr* - The contents of the floating point status register (FSR) at the time of the exception.

UQUAD *AddStage1*, *AddStage2*, *AddStage3* - Stages 1 - 3 of the floating point addition pipeline.

UQUAD *MulStage1*, *MulStage2*, *MulStage3* - Stages 1 - 3 of the floating point multiplication pipeline.

UQUAD *FldStage1*, *FldStage2*, *FldStage3* - Stages 1 - 3 of the floating point load pipeline.

UQUAD *IntResult* - The integer result of the graphics pipeline.

UQUAD *Kr* - The contents of the **KR** register.

UQUAD *Ki* - The contents of the **KI** register.

UQUAD *Merge* - The contents of the **MERGE** register.

UQUAD *T* - The contents of the **T** register.

ULONG *Fir* - The continuation instruction pointer.

ULONG *Fsr1*, *Fsr2*, *Fsr3* - The contents of the floating status register (FSR) for stages 1 - 3 of the pipeline.

ULONG *IntR4...IntR31* - The contents of the integer registers **r4** - **r31**.

UQUAD *FltF2...FltF31* - The contents of the floating point registers **f2** - **f31**.

ULONG *IntSp* - The contents of the stack pointer at the time of the exception.

ULONG *ExtendedSaveMask* - The extended register save mask that specifies that register *Int16...IntR31* are saved in the record.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG *Handler* - The address of the associated exception handler.

ULONG *RegisterSaveMask* - The standard register save mask that specifies that registers *IntR4...IntR15* and *FltF2...FltF31* are saved in the record.

ULONG *IntFp* - The contents of the frame pointer at the time of the exception.

ULONG *IntR1* - The contents of the register **R1** (return address) at the time of the exception.

ULONG - *Psr* - The processor status (PSR) at the time of the exception.

ULONG - *Epsr* - The extended processor status (EPSR) at the time of the exception.

4. Hardware-Defined Exceptions

Hardware-defined exceptions are initiated by the executive when a particular kind of fault condition is encountered as the result of instruction execution, e.g., an integer overflow. System software collects the information necessary to initiate the exception and then calls a routine that routes the exception to the appropriate exception handler.

The following sections describe the various hardware-defined exceptions in a machine-independent format. For each exception, the exception status code and any additional parameters are specified. These values are placed in the exception record when the particular type of exception is generated. In addition, any pertinent **Intel i860**-dependent information is also provided.

Not all hardware architectures generate all the various exceptions that are defined. Each port of **Windows NT** to a new hardware architecture requires a mapping of the hardware-defined exceptions onto the machine-independent format given below.

\ *The following sections must be carefully examined to ensure that they represent a machine-independent description for x86, as well as* ***i860****, exceptions.* \

4.1 Access Violation

An access violation exception is generated when an attempt is made to load or store data from/to a location that is not accessible to the current process, or when an attempt is made to execute an instruction that is not accessibile to the current process.

Exception Code: **STATUS_ACCESS_VIOLATION**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Additional Parameters: 2

Read/Write - A value of zero signifies a read; a value of one signifies a write.

Virtual Address - The virtual address of the data that is not accessible.

4.2 Breakpoint

A breakpoint exception occurs when a breakpoint instruction is executed, or a hardware-defined breakpoint is encountered (e.g. an address in a breakpoint register). This exception is intended for use by debuggers.

Exception Code: **STATUS_BREAKPOINT**

Additional Parameters: 1

Read/Write - A value of zero signifies a read; a value of one signifies a write.

i860 Implementation: The execution of a **TRAP r30,r29,r0** instruction, or a match with the address in the breakpoint register causes a breakpoint exception on the **Intel i860**.

4.3 Data-Type Misalignment

A data-type misalignment exception is generated when an attempt is made to load or store data from/to an address that is not naturally aligned, on a hardware architecture that does not provide alignment hardware. For example, 16-bit entities must be aligned on two-byte boundaries, 32-bit entities must be aligned on four-byte boundaries, etc.

Exception Code: **STATUS_DATATYPE_MISALIGNMENT**

Additional Parameters: 3

Read/Write - A value of zero signifies a read; a value of one signifies a write.

Data-type Mask - A data-type mask that specifies how many low-address bits must be zero. For example, the data-type mask for a 16-bit entity is one, a 32-bit entity three, etc.

Virtual Address - The virtual address of the misaligned data.

4.4 Floating Divide By Zero

A floating divide by zero exception is generated when an attempt is made to divide a floating point dividend by a floating point divisor of zero.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Exception Code: **STATUS_FLOATING_DIVIDE_BY_ZERO**

Additional Parameters: None

4.5 Floating Overflow

A floating overflow exception is generated when the resulting exponent of a floating point operation is greater than the magnitude allowed for the respective floating point data type.

Exception code: **STATUS_FLOATING_OVERFLOW**

Additional Parameters: None

4.6 Floating Underflow

A floating underflow exception is generated when the resulting exponent of a floating point operation is less than the magnitude provided for the respective floating point data type.

Exception Code: **STATUS_FLOATING_UNDERFLOW**

Additional Parameters: None

4.7 Floating Reserved Operand

A floating reserved operand exception is generated when one or more of the source operands in a floating point operation have a format that is reserved.

Exception Code: **STATUS_FLOATING_RESERVED_OPERAND**

Additional Parameters: None

4.8 Illegal Instruction

An illegal instruction exception is generated when an attempt is made to execute an instruction whose operation is not defined for the host machine architecture.

Exception Code: **STATUS_ILLEGAL_INSTRUCTION**

Additional Parameters: None

i860 Implementation: The execution of a **TRAP** instruction other than **TRAP r30,r29,r0** or **TRAP r30,r28,r0** or **TRAP r30,r27,r0** causes an illegal instruction exception.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.9 Privileged Instruction

A privileged instruction exception is generated when an attempt is made to execute an instruction whose operation is not allowed in current machine mode (e.g., an attempt to execute an instruction from user mode that is only allowed in kernel mode).

Exception Code: **STATUS_PRIVILEGED_INSTRUCTION**

Additional Parameters: None

4.10 Invalid Lock Sequence

An invalid lock sequence exception is generated when an attempt is made to execute an operation, within an interlocked section of code, such that the sequence is invalid for the host machine architecture.

Exception Code: **STATUS_INVALID_LOCK_SEQUENCE**

Additional Parameters: None

i860 Implementation: Exceeding the 32-instruction limit within a lock sequence, an attempt to execute a **TRAP** instruction within a lock sequence, or an attempt to execute an **INTOVR** instruction within a lock sequence causes an invalid lock sequence exception.

4.11 Integer Divide By Zero

An integer divide-by-zero exception is generated when an attempt is made to divide an integer dividend by an integer divisor of zero.

Exception Code: **STATUS_INTEGER_DIVIDE_BY_ZERO**

Additional Parameters: None

4.12 Integer Overflow

An integer overflow exception is generated when the result of an integer operation causes a carry out of the the most significant bit of the result, which is not the same as the carry into of the most significant bit of the result. For example, the addition of two positive integers that produces a negative result.

Exception Code: **STATUS_INTEGER_OVERFLOW**

Additional Parameters: None

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

i860 Implementation: The execution of an **INTOVR** instruction when **OF** set in **EPSR** causes an integer overflow exception. The **OF** bit in EPSR is cleared prior to initiating this exception.

4.13 Single Step

A single step exception is generated when a trace trap or other single instruction execution mechanism signals that one instruction has been executed. This exception is intended for use by debuggers.

Exception Code: **STATUS_SINGLE_STEP**

Additional Parameters: None

i860 Implementation: The execution of a **TRAP r30,r28,r0** instruction causes a single step exception.

5. Windows NT Software-Defined Exceptions

Windows NT software-defined exceptions are explicitly raised by system software when certain conditions are encountered, e.g., a page file read error. System software collects the information necessary to initiate the exception and then calls a routine that routes the exception to the appropriate exception handler.

5.1 Guard Page Violation

A guard page violation exception is generated when an attempt is made to load or store data from/to a location that is contained within a guard page. Memory management software immediately turns the guard page into a demand zero page and initiates a guard page violation exception.

Exception Code: **STATUS_GUARD_PAGE_VIOLATION**

Additional Parameters: 2

Read/Write - A value of zero signifies a read; a value of one signifies a write.

Virtual Address - The virtual address of the data within a guard page.

5.2 Page Read Error

A page read error exception is generated when an attempt is made to read a page into memory and an I/O error is encountered.

Exception Code: **STATUS_IN_PAGE_ERROR**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Additional Parameters: 1

Virtual Address - A virtual address within the page that was being read.

5.3 Paging File Quota Exceeded

A page file quota exceeded exception is generated when an attempt is made to commit backing store space for a page that is being removed from a process's working set.

Exception Code: **STATUS_PAGEFILE_QUOTA**

Additional Parameters: 1

Virtual Address - A virtual address within the page that was being read.

6. Standard Exception Handling

Standard exception handling is provided for some exceptions in which it is most likely that the user will select the default handling as the first resort, rather than wait until all other handlers have been given an opportunity to handle the exception. This enables the fault to be handled in the most efficient manner.

This capability is provided in **Windows NT** for alignment faults and IEEE floating point faults.

6.1 Alignment Faults

Standard handling of alignment faults ... TBS

6.2 IEEE Floating Faults

Standard handling of IEEE faults ... TBS

7. Exception Handling Interfaces

Several interfaces are supplied by the **Windows NT** system to implement the exception handling architecture described above. Some of these interfaces are intended for use only by the exception handling components themselves, while others are available to user-level software. The following subsections describe the exception handling APIs that are provided by **Windows NT**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.1 Exception Dispatcher

The exception dispatcher is responsible for searching the stack for frame-based exception handlers. There is a single exception dispatcher and it is responsible for dispatching both hardware-, and software-generated exceptions.

The exception dispatcher can be invoked with the **RtlpDispatchException** function:

BOOLEAN
RtlDispatchException (
    IN PEXCEPTION_RECORD *ExceptionRecord,*
    IN PCONTEXT *ContextRecord*
    );

Parameters:

ExceptionRecord - A pointer to an exception record that describes the exception, and the parameters of the exception, that has been raised.

ContextRecord - A pointer to a context record that describes the machine state at the time the exception occurred.

The exception dispatcher walks backward through the call frame hierarchy attempting to find an exception handler that will handle the exception. As each handler is encountered, it is called specifying the exception record, the context record, the address of the call frame of the establisher of the handler, and whether the handler is being called recursively or not, as parameters.

The exception handler may handle the exception or request that the scan of call frames be continued. As each step backwards is made in the call hierarchy, a check is made to ensure that the previous call frame address is within the current thread's stack limits and is aligned. If the stack is not within limits or is unaligned, then the *EXCEPTION_STACK_INVALID* flag is set in the exception flags field and the **NtLastChance** system service is called to finish processing of the exception. Otherwise, the previous frame is examined to determine if it specifies an exception handler.

The exception dispatcher is called by **RtlRaiseException** and by the executive code that processes hardware-generated exceptions.

7.2 Exception Handler

An exception handler is usually called by the exception dispatcher, specifying parameters that describe the exception and the environment in which the exception handler was established. Exception handlers, however, are also called during an

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

unwind operation and are given a chance to clean up data structures, deallocate resources, or do any other operations that are necessary to unwind the establisher's call frame.

The *EXCEPTION_UNWINDING*, *EXCEPTION_EXIT_UNWIND*, and *EXCEPTION_NESTED_CALL* flags in the exception record determine how an exception handler is being called. These flags are set by the exception dispatcher and the unwind function. If both the *EXCEPTION_UNWINDING* and *EXCEPTION_EXIT_UNWIND* flags are clear, then the exception handler is being called to handle an exception. Otherwise, an unwind operation is in progress, and the exception handler is being called to perform any necessary cleanup operations. If the exception handler is being called to handle an exception, then the *EXCEPTION_NESTED_CALL* flag determines whether a nested exception is in progress (i.e., another exception was raised in the containing scope before the previous exception was disposed of).

An exception handler has the following type definition:

typedef
EXCEPTION_DISPOSITION
(***PEXCEPTION_ROUTINE**) (
    IN PEXCEPTION_RECORD *ExceptionRecord*,
    IN PVOID *EstablisherFrame*,
    IN OUT PCONTEXT *ContextRecord*,
    IN OUT PVOID *DispatcherContext*
    );

Parameters:

ExceptionRecord - A pointer to an exception record that describes the exception and the parameters of the exception.

EstablisherFrame - A pointer to the call frame of the establisher of the exception handler.

ContextRecord - A pointer to a context record that describes the machine state at the time the exception occurred.

DispatcherContext - A pointer to a record that receives state information on nested exceptions and collided unwinds.

When an exception handler is called to handle an exception, it has several options for how it processes an exception:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. It can handle the exception, provide any fixup that is necessary by modifying the context record, and then continue execution of the program at the point of the exception by returning a disposition value of *ExceptionContinueExecution*.

2. It can handle the exception, determine that execution cannot be continued, and initiate an unwind operation.

3. It can decide that it cannot handle the exception and return a disposition value of *ExceptionContinueSearch*, which causes the exception dispatcher to continue the search for an exception handler.

When an exception handler is called during an unwind operation, it also has several options for how it processes the call:

1. It can perform any necessary cleanup operations and return a disposition value of *ExceptionContinueSearch*.

2. It can perform any necessary cleanup operations and initiate another unwind operation to a different target.

3. It can restore the machine state from the context record and continue execution directly.

If the exception handler belongs to the exception dispatcher itself, then it can also return a disposition value of *ExceptionNestedException* when it is called to handle an exception. Likewise, if the exception handler belongs to the unwind function, then it can also return a disposition value of *ExceptionCollidedUnwind* when it is called to perform any necessary cleanup operations (i.e., an unwind is in progress). For both of these cases, the *DispatcherContext* parameter is used to return information to either the exception dispatcher or the unwind function. No other exception handler can place information in this output parameter.

If an invalid disposition value is returned by an exception handler, then the exception **STATUS_INVALID_DISPOSITION** is raised by the exception dispatcher.

The *ContextRecord* parameter is intended for use by machine-specific code that either restores the machine state during an unwind operation, or manipulates the machine state in such a way as to fix up an exception. An example of such an exception handler is the default IEEE floating point exception handler, which uses the machine state information to determine how a floating point exception should actually be handled. Another example is the fixup necessary for unaligned data references. This type of exception handler is machine specific and will generally be supplied by **Microsoft**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

When an exception handler is called to handle an exception, the context record contains the machine state at the time of the exception. However, when an exception handler is called during an unwind operation, the context record contains the machine state of the exception handler's establisher.

When a disposition value of *ExceptionContinueExecution* is returned, the exception dispatcher checks to determine if the exception is continuable. If it is not continuable (i.e., the *EXCEPTION_NONCONTINUABLE* flag is set in the exception flags field of the exception record), then the exception dispatcher raises the exception **STATUS_NONCONTINUABLE_EXCEPTION**. Otherwise, the machine state is restored and execution resumes at the point of the exception.

A disposition value of *ExceptionContinueSearch* causes the exception dispatcher or unwind function to continue its scan of call frames.

If the exception handler of the exception dispatcher is encountered during the scan for an exception handler, then it returns a disposition value of *ExceptionNestedException* and the address of the call frame that established the exception handler most recently called by the exception dispatcher. The *EXCEPTION_NESTED_CALL* flag is set in the exception flags field of the exception record for each exception handler that is called between the exception dispatcher handler and the establisher of the most recently called exception handler. It is the responsibility of the individual exception handlers themselves to determine if they can be recursively called.

The exception handler of the unwind function returns a disposition value of *ExceptionCollidedUnwind* and the target frame of the current unwind. This information is used to determine the new scope of the unwind.

7.3 Raise Exception

A software exception can be raised with the **RtlRaiseException** function:

VOID
RtlRaiseException (
  IN PEXCEPTION_RECORD *ExceptionRecord*
  );

Parameters:

ExceptionRecord - A pointer to an exception record that describes the exception, and the parameters of the exception, that is raised.

Raising a software exception captures the machine state of the current thread in a context record. The *ExceptionAddress* field of the exception record is set to the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

caller's return address, and the exception dispatcher is then called in an attempt to locate a frame-based exception handler to handle the exception. Note that the associated debugger, if any, is not given a first chance to handle software exceptions.

If an exception handler returns a disposition value of *ExceptionContinueExecution*, then execution will return to the caller of **RtlRaiseException**. If no frame-based exception handler disposes of the exception, then **NtLastChance** is called to enable the appropriate system service emulation subsystem to perform any subsystem-specific processing.

7.4 Continuing From An Exception

Execution of a thread can be continued from the point of an exception with the **NtContinue** function:

VOID
NtContinue (
    IN PCONTEXT *ContextRecord*,
    IN BOOLEAN *TestAlert*
    );

Parameters:

ContextRecord - A pointer to a context record that describes the machine state that is to be restored.

TestAlert - A boolean value that specifies whether an alert should be tested for the previous processor mode. This parameter is used for APC processing.

This function restores the machine state from the specified context record and resumes execution of the thread.

> \ *Note that such a service would not normally be required. The* **Intel i860** *architecture, however, does not allow the entire machine state to be completely restored in user mode, and therefore, a system service must be called in kernel mode to perform this operation.* \

This function is called by the exception dispatcher to continue the execution of a thread when an exception handler returns a dispostion value of *ExecptionContinueExecution.*

7.5 Unwinding From An Exception

An exception can be unwound with the **RtlUnwind** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
VOID
RtlUnwind (
    IN PVOID TargetFrame OPTIONAL,
    IN PVOID TargetIp OPTIONAL,
    IN PEXCEPTION_RECORD ExceptionRecord OPTIONAL
    );
```

Parameters:

TargetFrame - An optional pointer to the call frame that is the target of the unwind. If this parameter is not specified, then the *EXCEPTION_EXIT_UNWIND* flag is set in the exception flags field of the exception record.

TargetIp - An optional instruction address that specifies the continuation address. This parameter is ignored if the *TargetFrame* parameter is not specified.

ExceptionRecord - An optional pointer to an exception record that is used when each exception handler is called during the unwind operation.

This function initiates an unwind of procedure call frames. The machine state at the time of the call to **RtlUnwind** is captured in a context record, the *EXCEPTION_UNWINDING* flag is set in the exception flags field of the exception record, and the *EXCEPTION_EXIT_UNWIND* flag is also set if the *TargetFrame* parameter is not specified. A backward walk through the procedure call frames is then performed to find the target of the unwind operation.

As each call frame is unwound, the machine state of the previous frame is computed by restoring any registers stored by the procedure. The previous frame is then examined to determine if it has an associated exception handler. If the call frame has an exception handler, then it is called specifying the exception record, the establisher's frame pointer, and the context record that contains the machine state of the handler's establisher. The exception handler should perform any cleanup operations that are necessary, and continue the unwind operation by returning a disposition value of *ExceptionContinueSearch*, initiating another unwind operation, or directly restoring the machine state from the context record.

Note that languages that support a termination model for exception handling (e.g., Ada, Modula-3, and the proposed extensions to **Microsoft** C) can implement this capability by unwinding to the frame of the establisher when a language-specific exception handler is invoked during either an unwind operation or during the dispatching of an exception.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

There is no return from a call to **RtlUnwind**. Control is either transferred to the specified instruction pointer address, or **NtLastChance** is called to perform secondary debugger processing and/or subsystem-specific default processing at the completion of the unwind operation. If **RtlUnwind** encounters an error during its processing, it raises another exception rather than return control to the caller.

If the target call frame is reached and an exit unwind is not being performed (i.e. the *TargetFrame* parameter is specified), then the computed machine state is restored from the context record and control is transfered to the address specified by the *TargetIp* parameter. Note that the stack pointer is not restored making it possible to transfer information on the stack. It is the responsibility of the code at the target address to reset the stack pointer as necessary.

If an exit unwind is being performed (i.e. the *TargetFrame* parameter is not specified), then all call frames are unwound until the base of the stack is reached. **NtLastChance** is then called to perform secondary debugger processing and/or subsystem-specific processing.

If the *ExceptionRecord* parameter is specified, then each exception handler encountered during the unwind operation is called using the specified record. If this parameter is not specified, then **RtlUnwind** constructs an exception record that specifies the exception **STATUS_UNWIND**.

During an unwind operation, it is possible for one unwind to *collide* with a previous unwind. This occurs when the scope of the second unwind overlaps the scope of the first unwind.

There are two cases to consider:

1. The target frame of the second unwind is a frame that has already been unwound by the first unwind.

2. The target frame of the second unwind occurs earlier in the call hierarchy than the target of the first unwind.

The first case is processed by unwinding call frames for the second unwind until the first call frame unwound by the first unwind is encountered. The second unwind is then terminated and processing of the first unwind is continued at the point where the first unwind was interrupted by the second unwind.

The second case is processed by changing the target of the first unwind to that of the second unwind, and then applying the handling that is provided for case one.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.6 Last Chance Exception Handling

Last chance exception handling can be invoked with the **NtLastChance** function:

NTSTATUS
NtLastChance (
    IN PEXCEPTION_RECORD *ExceptionRecord*,
    IN PCONTEXT *ContextRecord*
    );

Parameters:

ExceptionRecord - A pointer to an exception record that describes the exception, and the parameters of the exception, that has been raised.

ContextRecord - A pointer to a context record that describes the machine state at the time the exception occurred.

Last chance handling copies the exception and context records onto the kernel stack and checks to determine if a system service emulation subsystem port is associated with the thread's process. If a subsystem port is associated with the thread's process, then a message is sent to the port specifying the exception record and the identification of the client thread. Otherwise, default handling is provided for the exception.

This function is called by the exception dispatcher to perform subsystem and/or default handling for an exception that is not handled by any of the frame-based exception handlers. It is not called by any other component of the system.

Normally there is no return from the call to **NtLastChance**. However, if the context or exception record is not accessible to the calling process, then an access violation status is returned.

8. OS/2 2.0 Compatibility

The OS/2 Cruiser project is currently designing a new exception handling capability for OS/2 that replaces the current **DosSetVec** interface, and which can provide the basis for frame-based exception handling.

It is desirable to be able to directly emulate the proposed exception capabilities of OS/2 with the native frame-based exception handling provided by **Windows NT**. Furthermore, it is desirable to be able to use both the OS/2 style of exception handlers in the same program as the **Windows NT** frame-based handlers without a conflict arising. Currently this is not possible without further refinement of the OS/2 proposal and the introduction of certain constraints concerning the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

establishment and disestablishment of OS/2 exception handlers. Other problems with the current OS/2 design include the visibility of x86 architectural features, which makes the user interface nonportable.

The following changes and restrictions need to be specified:

1. The machine-dependent state must be separated from the exception state in the OS/2 proposal so that portable software can be written that makes use of the exception handling capabilities of OS/2 on architectures other than the x86.

2. The exception information included in an exception record for OS/2 must be specified in such a way as to be portable to architectures other than the x86 (i.e., a higher level of abstraction is needed for the parameter values).

3. A restriction must be placed on the establishment and disestablishment of OS/2 exception handlers such that they are strictly frame based (i.e., an exception handler that is established in a frame must be disestablished before leaving the frame).

4. A restriction must be placed on the use of OS/2 style exception handlers in the same frame as **Windows NT** frame-based exception handlers.

5. The semantics of **DosRaiseException** must be corrected to return to the call site if a continuation status is returned by an exception handler.

If these changes and restrictions are implemented, then the **Windows NT** exception handling capabilities, with slight modification, can directly emulate the OS/2 exception handling capabilities.

8.1 Windows NT Intel i860 Implementation

The **Intel i860 Windows NT** implementation of exception handling is frame based. Each call frame has a pointer that is dedicated to holding the address of an exception handler for the frame. Usually this is a language-supplied handler that provides whatever semantics are required to provide exception handling for the language. If there is no handler associated with the frame, then a flag is clear in the frame to signify that there is no exception handler and the dedicated pointer does not contain meaningful information. If the handler address is specified as VOID, then there is also no exception handler associated with the frame.

When an exception occurs, or an unwind operation is initiated, a backward walk through the call frames is conducted. If a call frame contains an exception handler, then it is called with the appropriate arguments.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Establishing an exception handler does not require the allocation of any heap storage, or the initialization of any data structure on the part of the user. Exception handlers are automatically disestablished upon leaving a procedure and deleting its call frame from the stack.

Unwind does not return to its caller. Rather it unwinds call frames, calling exception handlers as appropriate until the target of the unwind is reached, and then restores the machine state and transfers control to a specified destination instruction address.

8.2 OS/2 2.0 Intel x86 Implementation

The OS/2 implementation of exception handling on the **Intel x86** is list based. The head of the list is anchored in the Thread Information Block (TIB) of a thread. When an exception handler is established, a structure called an exception-handler-structure is supplied by the user, and linked into the last in, first out (LIFO) list of exception handlers. Disestablishing an exception handler removes the appropriate exception-handler-structure from the list.

The exception-handler-structure contains a link pointer and a pointer to the exception handler associated with the structure. The fields of the structure are exported to the user who is free to change the address of the exception handler while the structure is in the exception list.

When an exception occurs, or an unwind operation is initiated, a forward walk through the exception handler list is performed. Each handler is called with the appropriate arguments.

After completing an unwind operation (no unwind is actually done), the OS/2 function returns control to the caller, which must perform a longjmp() if necessary.

OS/2 defines the following APIs for exception handling:

1. **DosSetExceptionHandler** (**exception-handler-structure*) - This function establishes an exception handler by placing the specified exception-handler-structure at the front of the exception handler list.

2. **DosUnsetExceptionHandler** (**exception-handler-structure*) - This function disestablishes an exception handler by removing the specified exception-handler-structure from the exception handler list.

3. **DosRaiseException** (**exception-structure*) - This function raises a software exception.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. **DosUnwindException** (**exception-handler-structure*) - This function causes exception handlers to be called up, including the exception handler specified by the exception-handler-structure.

8.3 Windows NT Implementation of OS/2 Capabilities

In order to directly emulate OS/2 exception handling in **Windows NT**, the restrictions and changes described above for OS/2 must be made. Assuming these changes are made, the following paragraphs describe how **Windows NT** can directly emulate the OS/2 capabilities.

The meaning of the handler address in a call frame is expanded to be either a handler address (low-order bit is clear), or a pointer to a LIFO list of exception-handler-structures (low-order bit is set). A call frame can contain either a list head for OS/2 style exception handlers or a pointer to a single exception handler for **Windows NT** exceptions.

The function **DosSetExceptionHandler** inserts an exception-handler-structure in the LIFO list of exception handlers defined for the current call frame. If there is a Windows **NT** exception handler already established for the frame, then an attempt to insert an OS/2 style handler causes the exception **STATUS_INCOMPATIBLE_EXCEPTION_HANDLER** to be raised. Otherwise, the specified exception-handler-structure is inserted at the front of the exception handler list and the low-order bit of the exception handler address is set.

The function **DosUnsetExceptionHandler** removes an exception-handler-structure from the exception list associated with the current frame. If the current frame contains a Windows **NT** exception handler, or the specified exception-handler-structure is not in the current frame's exception handler list, then the exception **STATUS_HANDLER_NOT_FOUND** is raised. Otherwise, the specified exception-handler-structure is removed from the exception handler list of the currrent frame.

The function **DosRaiseException** reformats the exception record that it is passed into the exception record expected by **RtlRaiseException**. No other processing is required. If an exception handler returns a continuation status, then control returns to the caller of **DosRaiseException**.

The function **DosUnwindException** performs a prescan of call frames in an attempt to locate the specified exception-handler-structure. The prescan is performed by walking backwards through the call frame and examining the exception handler list for each frame that contains such a list. If the specified exception-handler-structure is not found, then the exception **STATUS_HANDLER_NOT_FOUND** is raised. Otherwise, **RtlUnwind** is called specifying the address of the target frame to unwind

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

to and the address of the exception-handler-structure as the continuation instruction address.

The **Windows NT** exception dispatcher performs a walk backwards through the call frames when an exception is raised. If it encounters a frame with a handler that has the low-order bit set, it knows that this is not really the address of a handler, but rather the address of an exception handler list for the frame. It calls each handler in the list, one after the other, exactly in the same manner as OS/2, thus implementating exactly the exception dispatching semantics of OS/2.

The function **RtlUnwind** also performs a walk backwards through the call frames when an unwind operation is initiated. This function also recognizes that frames containing a handler with the low-order bit set really point to a list of OS/2 style exception handlers. If the target of the unwind is a frame that contains an exception handler list, then it is known that the continuation address is really the address of an exception-handler-structure that is the target of the unwind and that control is to be returned to the caller of unwind. This implements exactly the same unwind semantics as OS/2.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, May 22, 1989

Revision 1.1, June 2, 1989

1. Major update to include first draft comments.
2. Added section on the implementation of OS/2 exception handling on top of the **Windows NT** capabilities.

Revision 1.2, June 6, 1989

1. Minor corrections of typos.
2. Deleted parameter to illegal instruction, privileged instruction, and invalid lock sequence exceptions to make them more portable.
3. Changed the type name of the context record to match the definition of thread context in the process structure.

Revision 1.3, August 4, 1989

1. Changed the exception dispatch sequence to include a second call to the debugger just before calling the system service emulation subsystem.
2. Changed the name of the *RECURSIVE_CALL* flag to *NESTED_CALL*.
3. Changed the definition of **NtLastChance** so that the function returns an access violation status if the exception or context record are not accessible to the calling process.

Revision 1.4, August 15, 1989

1. Changed names of exception flags to include a leading "EXCEPTION_" tag.
2. Changed field names in the context record to reflect the actual implementation which uses the context record as a call frame.
3. Changed the name of the exception dispatcher to a private internal name and added stack limit parameters.
4. Changed the exception disposition values from manifest constants to an enumerated type.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. The exception STATUS_INVALID_DISPOSITION is raised if an invalid disposition value.

6. Change name of NtContinueExecution to NtContinue and add a boolean parameter to specify whether a test alert should be executed.

7. The registers f0, f1, and r0 are no longer saved in the context record.

Revision 1.5, November 6, 1989

1. Delete stack limit arguments from exception dispatcher routine.

2. Change name of collided unwind status code from ExceptionNestedUnwind to ExceptionCollidedUnwind.

3. Change name of exception dispatcher from RtlpDispatchException to RtlDispatchException.

4. Change name of unwind routine from NtUnwind to RtlUnwind.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Executive Support Routines Specification

Author: *David Treadwell, Windows NT team*

Revision 1.0, August 2, 1989
Revision 1.1, October 11, 1989
Revision 1.2, January 31, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction.. 1

2. Get Information About Pages.. 3
   2.1 ExCreateBitMap 3
   2.2 DeleteBitMap 3
   2.3 ExInitializeBitMap... 4
   2.4 ExClearAllBits.. 4
   2.5 ExSetAllBits 5
   2.6 ExFindClearBits.. 5
   2.7 ExFindSetBits 5
   2.8 ExFindClearBitsAndSet.. 6
   2.9 ExFindSetBitsAndClear.. 6
   2.10 ExClearBits.. 7
   2.11 ExSetBits 7
   2.12 ExFindLongestRunClear.. 8
   2.13 ExFindLongestRunSet... 8
   2.14 ExCheckBit 9

3. Determine Pool Type 10
   3.1 MmDeterminePoolType.. 10

4. Allocate and Deallocate Pool.. 11
   4.1 ExLockPool 11
   4.2 ExUnlockPool... 11
   4.3 InitializePool.. 12
   4.4 ExAllocatePool.. 12
   4.5 ExAllocatePoolWithQuota 13
   4.6 ExDeallocatePool... 14

5. Initialize and Extend Zone Buffer 15
   5.1 ExInitializeZone 15
   5.2 ExExtendZone.. 16

6. Perform Interlocked Allocate and Free from Zone.......................... 17
   6.1 ExAllocateFromZone 17
   6.2 ExFreeToZone 17
   6.3 ExIsFullZone.. 17
   6.4 ExInterlockedAllocateFromZone 18
   6.5 ExInterlockedFreeToZone 18

7. Zero and Move Memory... 20
   7.1 ExZeroMemory.. 20
   7.2 ExMoveMemory... 20

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

8. Manage Memory for I/O 22
8.1 MmProbeAndLockPages 22
8.2 MmUnlockPages 22
8.3 MmMapLockedPages 23
8.4 MmUnmapLockedPages 23
8.5 MmMapIoSpace 24
8.6 MmUnmapIoSpace 25
8.7 MmGetPhysicalAddress 25
8.8 MmSizeOfMdl 26
8.9 MmCreateMdl 26

9. Is Address Valid 28
9.1 MmIsAddressValid 28

10. Perform Bit Map Operations 29
10.1 PAGE_ALIGN 29
10.2 BYTES_TO_PAGES 29
10.3 ROUND_TO_PAGES 29
10.4 BYTE_OFFSET 30
10.5 ADDRESS_AND_SIZE_TO_SPAN_PAGES 30

11. Manage Object Handles and Handle Tables 32
11.1 ExCreateHandleTable 32
11.2 ExLockHandleTable 33
11.3 ExUnlockHandleTable 33
11.4 ExDupHandleTable 34
11.5 ExDestroyHandleTable 34
11.6 ExDumpHandleTable 35
11.7 ExEnumHandleTable 35
11.8 ExCreateHandle 36
11.9 ExDestroyHandle 37
11.10 ExMapHandleToPointer 37

12. Probe and Validate Arguments 39
12.1 ProbeForRead 39
12.2 ProbeForWrite 39
12.3 ProbeAndReadChar 40
12.4 ProbeAndReadUchar 40
12.5 ProbeAndReadShort 40
12.6 ProbeAndReadLong 40
12.7 ProbeAndReadUlong 40
12.8 ProbeAndReadQuad 40
12.9 ProbeAndReadUquad 40
12.10 ProbeAndReadHandle 41

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

12.11 ProbeAndReadBoolean 41
12.12 ProbeForWriteChar 41
12.13 ProbeForWriteUchar 41
12.14 ProbeForWriteShort 41
12.15 ProbeForWriteUshort 41
12.16 ProbeForWriteLong 41
12.17 ProbeForWriteUlong 41
12.18 ProbeForWriteQuad 41
12.19 ProbeForWriteUquad 42
12.20 ProbeForWriteHandle 42
12.21 ProbeForWriteBoolean 42
12.22 ProbeAndWriteChar 42
12.23 ProbeAndWriteUchar 42
12.24 ProbeAndWriteShort 42
12.25 ProbeAndWriteUshort 42
12.26 ProbeAndWriteLong 42
12.27 ProbeAndWriteUlong 43
12.28 ProbeAndWriteQuad 43
12.29 ProbeAndWriteUquad 43
12.30 ProbeAndWriteHandle 43
12.31 ProbeAndWriteBoolean 43

13. Perform Restricted Interlock Operations 44
13.1 ExInterlockedAddLong 44
13.2 ExInterlockedAddShort 44
13.3 ExInterlockedInsertHeadList 45
13.4 ExInterlockedInsertTailList 45
13.5 ExInterlockedRemoveHeadList 46
13.6 ExInterlockedPopEntryList 46
13.7 ExInterlockedPushEntryList 47

14. Allocate and Free Spin Locks 48
14.1 ExAllocateSpinLock 48
14.2 ExFreeSpinLock 48

15. Perform General Interlocked Operations 49
15.1 RtlInterlockedAddLong 49
15.2 RtlInterlockedAddShort 49
15.3 RtlInterlockedInsertHeadList 50
15.4 RtlInterlockedInsertTailList 50
15.5 RtlInterlockedRemoveHeadList 51
15.6 RtlInterlockedRemoveHeadList 51
15.7 RtlInterlockedPopEntryList 52
15.8 RtlInterlockedPushEntryList 52

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

16. Perform Operations on Counted Strings 54
16.1 RtlInitString 54
16.2 RtlCopyString... 54
16.3 RtlCompareString.. 55
16.4 RtlEqualString 55

17. Debugging Support Functions.. 57
17.1 DbgBreakPoint... 57
17.2 DbgCommand 57
17.3 DbgQueryInstructionCounter 57
17.4 DbgPrint 58
17.5 DbgPrompt... 58
17.6 DbgLoadImageFileSymbols 59
17.7 DbgSetDirBaseForImage.. 59
17.8 DbgKillDirBase... 60
17.9 DbgCheckpointSimulator 60

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This chapter describes executive support routines that are not documented elsewhere in the ***Windows NT*** *Design Workbook*. The routines are callable from kernel mode within the **Windows NT** executive. The following routines are presented in subsequent sections:

Get Information About Pages —Routines to calculate values related to the memory pagesize

Determine Pool Type —A memory management routine that determines whether a virtual address resides in paged or nonpaged memory pool

Allocate and Deallocate Pool —Routines used to allocate and deallocate memory pool using a binary buddy algorithm

Initialize and Extend Zone Buffer —Routines that initialize or extend a zone buffer (used primarily by local process communication)

Perform Interlocked Allocate and Free from Zone —Routines to allocate and free memory from a zone in a multiprocessor-safe manner

Zero and Move Memory —Routines to zero and move memory

Manage Memory for I/O —Routines that provide memory management support for the I/O system

Is Address Valid —A routine that determines if a given virtual address will cause a page fault if read

Perform Bit Map Operations —Routines to create, initialize, and manipulate bit maps

Manage Object Handles and Handle Tables —Routines that support object handles and handle tables

Probe and Validate Arguments —Routines that provide argument validation for system service calls

Perform Restricted Interlocked Operations —Restricted routines (no page faults allowed) implementing operations that must be synchronized across processors in a multiprocessing system

Allocate and Free Spin Locks —Routines to allocate and free spin locks (specialized mutual exclusion semaphores)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Perform General Interlocked Operations —Unrestricted routines (page faults allowed) implementing operations that must be synchronized across processors in a multiprocessing system

Perform Operations on Counted Strings —Routines that manipulate counted strings (strings that maintain a length field)

Debugging Support Functions —Routines for interfacing kernel-mode commands to the kernel-mode debugger.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2. Get Information About Pages

Implementation of the bit map routines for the Windows NT executive.

Bit numbers within the bit map are zero based. The first is numbered zero.

A bit map is allocated and initialized using the ExCreateBitMap routine. Once a bit map has been created, it must be set to a known state using either the ExSetAllBits or the ExClearAllBits routine.

The ExInitializeBitMap routine is provided to initialize preallocated bit maps.

The bit map routines keep track of the number of bits clear or set by subtracting or adding the number of bits operated on as bit ranges are cleared or set; individual bit states are not tested. This means that if a range of bits is set, it is assumed that the total range is currently clear.

2.1 ExCreateBitMap

PEX_BITMAP
ExCreateBitMap(
    IN ULONG *SizeOfBitMap*,
    IN POOL_TYPE *PoolType*
)

Routine Description:

This procedure allocates a bit map from the specified pool and returns a pointer to the bit map.

Parameters:

SizeOfBitMap - Supplies the number of bits required in the bitmap.

PoolType - Supplies the type of pool from which to allocate the bit map.

Return Value:

PEX_BITMAP - Returns a pointer to the allocated bit map. The bit map is not initialized.

2.2 DeleteBitMap

VOID
DeleteBitMap(

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        IN PEX_BITMAP BitMap
        )
```

Routine Description:

This procedure deallocates a bit map from the specified pool.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

Return Value:

None.

2.3 ExInitializeBitMap

```
VOID
ExInitializeBitMap(
        IN PEX_BITMAP BitMap,
        IN ULONG SizeOfBitMap
        )
```

Routine Description:

This procedure initializes a bit map which has already been allocated.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

SizeOfBitMap - Supplies the number of bits required in the bit map.

Return Value:

None.

2.4 ExClearAllBits

```
VOID
ExClearAllBits(
        IN PEX_BITMAP BitMap
        )
```

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This procedure clears all bits in the specified bit map.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

Return Value:

None.

2.5 ExSetAllBits

```
VOID
ExSetAllBits(
    IN PEX_BITMAP BitMap
    )
```

Routine Description:

This procedure sets all bits in the specified bit map.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

Return Value:

None.

2.6 ExFindClearBits

```
ULONG
ExFindClearBits(
    IN PEX_BITMAP BitMap,
    IN ULONG NumberToFind
    )
```

Routine Description:

This procedure searches the specified bit map for the specified contiguous region of clear bits.

Uses methods from Pinball scan for bit block algorithm.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

NumberToFind - Supplies the size of the contiguous region to find.

Return Value:

ULONG - Starting value (zero based) of the contiguous region found.

2.7 ExFindSetBits

ULONG
ExFindSetBits(
IN PEX_BITMAP *BitMap*,
IN ULONG *NumberToFind*
)

Routine Description:

This procedure searches the specified bit map for the specified contiguous region of set bits.

Uses methods from Pinball scan for bit block algorithm.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

NumberToFind - Supplies the size of the contiguous region to find.

Return Value:

ULONG - Starting value (zero based) of the contiguous region found.

2.8 ExFindClearBitsAndSet

ULONG
ExFindClearBitsAndSet(
IN PEX_BITMAP *BitMap*,
IN ULONG *NumberToFind*
)

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This procedure searches the specified bit map for the specified contiguous region of clear bits, sets the bits and returns the starting bit number which was clear then set.

Uses methods from Pinball scan for bit block algorithm.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

NumberToFind - Supplies the size of the contiguous region to find.

Return Value:

ULONG - Starting value (zero based) of the contiguous region found.

2.9 ExFindSetBitsAndClear

ULONG
ExFindSetBitsAndClear(
    IN PEX_BITMAP *BitMap*,
    IN ULONG *NumberToFind*
)

Routine Description:

This procedure searches the specified bit map for the specified contiguous region of set bits, clears the bits and returns the starting bit number which was set then clear.

Uses methods from Pinball scan for bit block algorithm.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

NumberToFind - Supplies the size of the contiguous region to find.

Return Value:

ULONG - Starting value (zero based) of the contiguous region found.

2.10 ExClearBits

VOID

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
ExClearBits(
    IN PEX_BITMAP BitMap,
    IN ULONG StartingLocation,
    IN ULONG NumberToClear
    )
```

Routine Description:

This procedure clears the specified range of bits within the specified bit map.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

StartingLocation - Supplies the number of the first bit to clear.

NumberToClear - Supplies the number of bits to clear.

Return Value:

None.

2.11 ExSetBits

```
VOID
ExSetBits(
    IN PEX_BITMAP BitMap,
    IN ULONG StartingLocation,
    IN ULONG NumberToSet
    )
```

Routine Description:

This procedure sets the specified range of bits within the specified bit map.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

StartingLocation - Supplies the number of the first bit to set.

NumberToClear - Supplies the number of bits to set.

Return Value:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

None.

2.12 ExFindLongestRunClear

ULONG
ExFindLongestRunClear(
    IN PEX_BITMAP *BitMap*
)

Routine Description:

This procedure finds the largest contiguous range of clear bits within the specified bit map.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

Return Value:

ULONG - Largest contiguous range of clear bits.

2.13 ExFindLongestRunSet

ULONG
ExFindLongestRunSet(
    IN PEX_BITMAP *BitMap*
)

Routine Description:

This procedure finds the largest contiguous range of set bits within the specified bit map.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

Return Value:

ULONG - Largest contiguous range of set bits.

2.14 ExCheckBit

ULONG

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ExCheckBit(
IN PEX_BITMAP *BitMap***,**
IN ULONG *BitPosition*
)

Routine Description:

This procedure returns the state of the specified bit within the specified bit map.

Parameters:

BitMap - Supplies a pointer to the previously allocated bit map.

BitPosition - Supplies the bit number of which to return the state.

Return Value:

ULONG - The state of the specified bit.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. Determine Pool Type

This module contains the routines which allocate and deallocate one or more pages from paged or nonpaged pool.

3.1 MmDeterminePoolType

POOL_TYPE
MmDeterminePoolType(
  IN PVOID *VirtualAddress*
)

Routine Description:

This function determines which pool a virtual address resides within.

Parameters:

VirtualAddress - Supplies the virtual address to determine which pool it resides within.

Return Value:

Returns the POOL_TYPE (PagedPool or NonPagedPool).

Environment:

Kernel Mode Only.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Allocate and Deallocate Pool

Implementation of the binary buddy pool allocator for the Windows NT executive.

4.1 ExLockPool

HANDLE
ExLockPool(
    IN POOL_TYPE *PoolType*
)

Routine Description:

This function locks the pool specified by pool type.

Parameters:

PoolType - Specifies the pool that should be locked.

Return Value:

Opaque - Returns a lock handle that must be returned in a subsequent call to ExUnlockPool.

4.2 ExUnlockPool

VOID
ExUnlockPool(
    IN POOL_TYPE *PoolType***,**
    IN HANDLE *LockHandle***,**
    IN BOOLEAN *Wait*
)

Routine Description:

This function unlocks the pool specified by pool type. If the value of the Wait parameter is true, then the pool's lock is released using "wait == true".

Parameters:

PoolType - Specifies the pool that should be unlocked.

LockHandle - Specifies the lock handle from a previous call to ExLockPool.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Wait - Supplies a boolean value that signifies whether the call to ExUnlockPool will be immediately followed by a call to one of the kernel Wait functions.

Return Value:

None.

4.3 InitializePool

VOID
InitializePool(
IN POOL_TYPE *PoolType*,
IN ULONG *Threshold*
)

Routine Description:

This procedure initializes a pool descriptor for a binary buddy pool type. Once initialized, the pool may be used for allocation and deallocation.

This function should be called once for each pool type during system initialization.

Each pool descriptor contains an array of list heads for free blocks. Each list head holds blocks of a particular size. One list head contains page-sized blocks. The other list heads contain 1/2- page-sized blocks, 1/4-page-sized blocks.... A threshold is associated with the page-sized list head. The number of free blocks on this list will not grow past the specified threshold. When a deallocation occurs that would cause the threshold to be exceeded, the page is returned to the page-aliged pool allocator.

Parameters:

PoolType - Supplies the type of pool being initialized (e.g. nonpaged pool, paged pool...).

Threshold - Supplies the threshold value for the specified pool.

Return Value:

None.

4.4 ExAllocatePool

PVOID

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ExAllocatePool(
    IN POOL_TYPE *PoolType*,
    IN ULONG *NumberOfBytes*
)

Routine Description:

This function allocates a block of pool of the specified type and returns a pointer to the allocated block. This function is used to access both the page-aligned pools, and the binary buddy (less than a page) pools.

If the number of bytes specifies a size that is too large to be satisfied by the appropriate binary buddy pool, then the page-aligned pool allocator is used. The allocated block will be page-aligned and a page-sized multiple.

Otherwise, the appropriate binary buddy pool is used. The allocated block will be 64-bit aligned, but will not be page aligned. The binary buddy allocator calculates the smallest block size that is a power of two and that can be used to satisfy the request. If there are no blocks available of this size, then a block of the next larger block size is allocated and split in half. One piece is placed back into the pool, and the other piece is used to satisfy the request. If the allocator reaches the paged-sized block list, and nothing is there, the page-aligned pool allocator is called. The page is added to the binary buddy pool...

Parameters:

PoolType - Supplies the type of pool to allocate.

NumberOfBytes - Supplies the number of bytes to allocate.

Return Value:

Non-NULL - Returns a pointer to the allocated pool.

4.5 ExAllocatePoolWithQuota

PVOID
ExAllocatePoolWithQuota(
    IN POOL_TYPE *PoolType*,
    IN ULONG *NumberOfBytes*
)

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function allocates a block of pool of the specified type, returns a pointer to the allocated block, and if the binary buddy allocator was used to satisfy the request, charges pool quota to the current process. This function is used to access both the page-aligned pools, and the binary buddy.

If the number of bytes specifies a size that is too large to be satisfied by the appropriate binary buddy pool, then the page-aligned pool allocator is used. The allocated block will be page-aligned and a page-sized multiple. No quota is charged to the current process if this is the case.

Otherwise, the appropriate binary buddy pool is used. The allocated block will be 64-bit aligned, but will not be page aligned. After the allocation completes, an attempt will be made to charge pool quota (of the appropriate type) to the current process object. If the quota charge succeeds, then the pool block's header is adjusted to point to the current process. The process object is not dereferenced until the pool is deallocated and the appropriate amount of quota is returned to the process. Otherwise, the pool is deallocated, a "quota exceeded" condition is raised.

Parameters:

PoolType - Supplies the type of pool to allocate.

NumberOfBytes - Supplies the number of bytes to allocate.

Return Value:

Non-NULL - Returns a pointer to the allocated pool.

Unspecified - If insuffient quota exists to complete the pool allocation, the return value is unspecified.

4.6 ExDeallocatePool

VOID
ExDeallocatePool(
IN PVOID *P*
)

Routine Description:

This function deallocates a block of pool. This function is used to deallocate to both the page aligned pools, and the binary buddy (less than a page) pools.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the address of the block being deallocated is page-aligned, then the page-aliged pool deallocator is used.

Otherwise, the binary buddy pool deallocator is used. Deallocation looks at the allocated block's pool header to determine the pool type and block size being deallocated. If the pool was allocated using ExAllocatePoolWithQuota, then after the deallocation is complete, the appropriate process's pool quota is adjusted to reflect the deallocation, and the process object is dereferenced.

Parameters:

P - Supplies the address of the block of pool being deallocated.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Initialize and Extend Zone Buffer

This module implements a simple zone buffer manager. The primary consumer of this module is local LPC.

The zone package provides a fast and efficient memory allocator for fixed-size 64-bit aligned blocks of storage. The zone package does not provide any serialization over access to the zone header and associated free list and segment list. It is the responsibility of the caller to provide any necessary serialization.

The zone package views a zone as a set of fixed-size blocks of storage. The block size of a zone is specified during zone initialization. Storage is assigned to a zone during zone initialization and when a zone is extended. In both of these cases, a segment and length are specified.

The zone package uses the first ZONE_SEGMENT_HEADER portion of the segment for zone overhead. The remainder of the segment is carved up into fixed-size blocks and each block is added to the free list maintained in the zone header.

As long as a block is on the free list, the first SINGLE_LIST_ENTRY (32 bit) sized piece of the block is used as zone overhead. The rest of the block is not used by the zone package and may be used by applications to cache information. When a block is not on the free list, its entire contents are available to the application.

5.1 ExInitializeZone

NTSTATUS
ExInitializeZone(
    IN PZONE_HEADER *Zone***,**
    IN ULONG *BlockSize***,**
    IN PVOID *InitialSegment***,**
    IN ULONG *InitialSegmentSize*
)

Routine Description:

This function initializes a zone header. Once successfully initialized, blocks can be allocated and freed from the zone, and the zone can be extended.

Parameters:

Zone - Supplies the address of a zone header to be initialized.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BlockSize - Supplies the block size of the allocatable unit within the zone. The size must be larger that the size of the initial segment, and must be 64-bit aligned.

InitialSegment - Supplies the address of a segment of storage. The first ZONE_SEGMENT_HEADER-sized portion of the segment is used by the zone allocator. The remainder of the segment is carved up into fixed size (BlockSize) blocks and is made available for allocation and deallocation from the zone. The address of the segment must be aligned on a 64-bit boundary.

InitialSegmentSize - Supplies the size in bytes of the InitialSegment.

Return Value:

STATUS_UNSUCCESSFUL - BlockSize or InitialSegment was not aligned on 64-bit boundaries, or BlockSize was larger than the initial segment size.

STATUS_SUCCESS - The zone was successfully initialized.

5.2 ExExtendZone

NTSTATUS
ExExtendZone(
    IN PZONE_HEADER *Zone***,**
    IN PVOID *Segment***,**
    IN ULONG *SegmentSize*
)

Routine Description:

This function extends a zone by adding another segment's worth of blocks to the zone.

Parameters:

Zone - Supplies the address of a zone header to be extended.

Segment - Supplies the address of a segment of storage. The first ZONE_SEGMENT_HEADER-sized portion of the segment is used by the zone allocator. The remainder of the segment is carved up into fixed-size (BlockSize) blocks and is added to the zone. The address of the segment must be aligned on a 64- bit boundary.

SegmentSize - Supplies the size in bytes of Segment.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>Return Value:</u>

STATUS_UNSUCCESSFUL - BlockSize or Segment was not aligned on 64-bit boundaries, or BlockSize was larger than the segment size.

STATUS_SUCCESS - The zone was successfully extended.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6. Perform Interlocked Allocate and Free from Zone

Public executive data structures and procedure prototypes.

6.1 ExAllocateFromZone

PVOID
ExAllocateFromZone(
IN PZONE_HEADER *Zone*
)

Routine Description:

This routine removes an entry from the zone and returns a pointer to it.

Parameters:

Zone - Pointer to the zone header controlling the storage from which the entry is to be allocated.

Return Value:

The function value is a pointer to the storage allocated from the zone.

6.2 ExFreeToZone

VOID
ExFreeToZone(
IN PZONE_HEADER *Zone*,
IN PVOID *Block*
)

Routine Description:

This routine places the specified block of storage back onto the free list in the specified zone.

Parameters:

Zone - Pointer to the zone header controlling the storage to which the entry is to be inserted.

Block - Pointer to the block of storage to be freed back to the zone.

Return Value:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

None.

6.3 ExIsFullZone

BOOLEAN
ExIsFullZone(
    IN PZONE_HEADER *Zone*
)

Routine Description:

This routine determines if the specified zone is full or not. A zone is considered full if the free list is empty.

Parameters:

Zone - Pointer to the zone header to be tested.

Return Value:

TRUE if the zone is full and FALSE otherwise.

6.4 ExInterlockedAllocateFromZone

PVOID
ExInterlockedAllocateFromZone(
    IN PZONE_HEADER *Zone*,
    IN PKSPIN_LOCK *Lock*
)

Routine Description:

This routine removes an entry from the zone and returns a pointer to it. The removal is performed with the specified lock owned for the sequence to make it MP-safe.

Parameters:

Zone - Pointer to the zone header controlling the storage from which the entry is to be allocated.

Lock - Pointer to the spin lock which should be obtained before removing the entry from the allocation list. The lock is released before returning to the caller.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

The function value is a pointer to the storage allocated from the zone.

6.5 ExInterlockedFreeToZone

VOID
ExInterlockedFreeToZone(
    IN PZONE_HEADER *Zone***,**
    IN PVOID *Block***,**
    IN PKSPIN_LOCK *Lock*
)

Routine Description:

This routine places the specified block of storage back onto the free list in the specified zone. The insertion is performed with the lock owned for the sequence to make it MP-safe.

Parameters:

Zone - Pointer to the zone header controlling the storage to which the entry is to be inserted.

Block - Pointer to the block of storage to be freed back to the zone.

Lock - Pointer to the spin lock which should be obtained before inserting the entry onto the free list. The lock is released before returning to the caller.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7. Zero and Move Memory

This module implements functions to zero and move memory blocks of memory. If the memory is aligned on 8 byte boundaries then these functions are very efficient, otherwise they do their work a byte at a time.

7.1 ExZeroMemory

```
VOID
ExZeroMemory(
    IN PVOID Destination,
    IN ULONG Length
    )
```

Routine Description:

These functions zero memory. The ExZeroMemory function determines the most efficient method to use based on the alignment of the Destination pointer and the Length. If the Destination pointer is aligned but the Length is not, then it will zero alignment sized units and then zero the odd number of bytes to finish up. If the Destination pointer is not aligned, then it will zero the entire length by bytes.

Parameters:

Destination (r16) - Supplies a pointer to the memory to zero.

Length (r17) - Supplies the Length, in bytes, of the memory to be zeroed.

Return Value:

None.

Performance:

10 Instructions to setup

2 Instructions per MEMORY_ALIGNMENT bytes zeroed

4 Instructions for each trailing odd byte

4 Instructions to finish

Zero ObjectTableEntry (16 bytes, quad aligned) is 18 instructions

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.2 ExMoveMemory

VOID
ExMoveMemory(
IN PVOID *Destination*,
IN PVOID *Source* **OPTIONAL,**
IN ULONG *Length*
)

Routine Description:

This function moves memory. The ExMoveMemory function determines the most efficient method to use based on the alignment of the Source and Destination pointers and the Length.

Parameters:

Destination (r16) - Supplies a pointer to the destination of the move.

Source (r17) - Supplies a pointer to the memory to move. If NULL then zeros the memory at Destination.

Length (r18) - Supplies the Length, in bytes, of the memory to be moved.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

8. Manage Memory for I/O

This module contains routines which provide support for the I/O system.

8.1 MmProbeAndLockPages

VOID
MmProbeAndLockPages(
  IN OUT PMDL *MemoryDescriptorList***,**
  IN KPROCESSOR_MODE *AccessMode***,**
  IN LOCK_OPERATION *Operation*
)

Routine Description:

This routine probes the specified pages, makes the pages resident and locks the physical pages mapped by the virtual pages in memory. The Memory descriptor list is updated to describe the physical pages.

Parameters:

MemoryDescriptorList - Supplies a pointer to a Memory Descriptor List (MDL). The supplied MDL must supply a virtual address, byte offset and length field. The physical page portion of the MDL is updated when the pages are locked in memory.

AccessMode - Supplies the access mode in which to probe the arguments. One of KernelMode or UserMode.

Operation - Supplies the operation type. One of IoReadAccess, IoWriteAccess or IoModifyAccess.

Return Value:

None - exceptions are raised.

Environment:

Kernel mode.

8.2 MmUnlockPages

VOID
MmUnlockPages(
  IN OUT PMDL *MemoryDescriptorList*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

)

Routine Description:

This routine unlocks physical pages which are described by a Memory Descriptor List.

Parameters:

MemoryDescriptorList - Supplies a pointer to a memory description list (MDL). The supplied MDL must have been supplied to MmLockPages to lock the pages down. As the pages are unlocked, the MDL is updated.

Return Value:

None.

Environment:

Kernel mode.

8.3 MmMapLockedPages

PVOID
MmMapLockedPages(
IN PMDL *MemoryDescriptorList***,**
IN KPROCESSOR_MODE *AccessMode*
)

Routine Description:

This function maps physical pages described by a memory description list into the system virtual address space.

Parameters:

MemoryDescriptorList - Supplies a valid Memory Descriptor List which has been updated by MmProbeAndLockPages.

AccessMode - Supplies an indicator of where to map the pages; KernelMode indicates that the pages should be mapped in the system part of the address space, UserMode indicates the pages should be mapped in the user part of the address space.

Return Value:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Returns the base address where the pages are mapped. The base address has the same offset as the virtual address in the MDL.

This routine will raise an exception if the processor mode is USER_MODE and quota limits or VM limits are exceeded.

Environment:

Kernel mode.

8.4 MmUnmapLockedPages

VOID
MmUnmapLockedPages(
    IN PVOID *BaseAddress*,
    IN PMDL *MemoryDescriptorList*
)

Routine Description:

This routine unmaps locked pages which were previously mapped via a MmMapLockedPages function.

Parameters:

BaseAddress - Supplies the base address where the pages were previously mapped.

MemoryDescriptorList - Supplies a valid Memory Descriptor List which has been updated by MmProbeAndLockPages.

Return Value:

None.

Environment:

Kernel mode.

8.5 MmMapIoSpace

PVOID
MmMapIoSpace(
    IN PHYSICAL_ADDRESS *PhysicalAddress*,

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
IN ULONG NumberOfBytes
)
```

Routine Description:

This function maps the specified physical address into the non-pageable portion of the system address space.

Parameters:

PhysicalAddress - Supplies the starting physical address to map.

NumberOfBytes - Supplies the number of bytes to map.

Return Value:

Returns the virtual address which maps the specified physical addresses.

Environment:

Kernel mode. APCs disabled.

8.6 MmUnmapIoSpace

```
VOID
MmUnmapIoSpace(
    IN PVOID BaseAddress,
    IN ULONG NumberOfBytes
    )
```

Routine Description:

This function unmaps a range of physical address which were previously mapped via an MmMapIoSpace function call.

Parameters:

BaseAddress - Supplies the base virtual address where the physical address was previously mapped.

NumberOfBytes - Supplies the number of bytes which were mapped.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Environment:

Kernel mode.

8.7 MmGetPhysicalAddress

PHYSICAL_ADDRESS
MmGetPhysicalAddress(
    IN PVOID *BaseAddress*
)

Routine Description:

This function returns the corresponding physical address for a valid virtual address.

Parameters:

BaseAddress - Supplies the virtual address for which to return the physical address.

Return Value:

Returns the corresponding physical address.

Environment:

Kernel mode. APCs disabled.

8.8 MmSizeOfMdl

ULONG
MmSizeOfMdl(
    IN PVOID *Base***,**
    IN ULONG *Length*
)

Routine Description:

This function returns the number of bytes required for an MDL for a given buffer and size.

Parameters:

Base - Supplies the base virtual address for the buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Length - Supplies the size of the buffer in bytes.

Return Value:

Returns the number of bytes required to contain the MDL.

Environment:

Kernel mode.

8.9 MmCreateMdl

```
PMDL
MmCreateMdl(
    IN PMDL MemoryDescriptorList OPTIONAL,
    IN PVOID Base,
    IN ULONG Length
    )
```

Routine Description:

This function optionally allocates and initializes an MDL.

Parameters:

MemoryDescriptorList - Optionally supplies the address of the MDL to initialize. If this address is supplied as NULL an MDL is allocated from non-paged pool and initialized.

Base - Supplies the base virtual address for the buffer.

Length - Supplies the size of the buffer in bytes.

Return Value:

Returns the address of the initialized MDL.

Environment:

Kernel mode.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

9. Is Address Valid

This module contains the pager for memory management.

9.1 MmIsAddressValid

```
BOOLEAN
MmIsAddressValid(
    IN PVOID VirtualAddress
    )
```

Routine Description:

For a given virtual address this function returns TRUE if no page fault will occur for a read operation on the address, FALSE otherwise.

Note that after this routine was called, if appropriate locks are not held, a non-faulting address could fault.

Parameters:

VirtualAddress - Supplies the virtual address to check.

Return Value:

TRUE if a no page fault would be generated reading the virtual address, FALSE otherwise.

Environment:

Kernel mode.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10. Perform Bit Map Operations

This module contains the public data structures and procedure prototypes for the memory management system.

10.1 PAGE_ALIGN

PVOID
PAGE_ALIGN(
    IN PVOID *Va*
)

Routine Description:

The PAGE_ALIGN macro takes a virtual address and returns a page-aligned virtual address for that page.

Parameters:

Va - Virtual address.

Return Value:

Returns the page aligned virtual address.

10.2 BYTES_TO_PAGES

ULONG
BYTES_TO_PAGES(
    IN ULONG *Size*
)

Routine Description:

The BYTES_TO_PAGES macro takes the size in bytes and calculates the number of pages required to contain the bytes.

Parameters:

Size - Size in bytes.

Return Value:

Returns the number of pages required to contain the specified size.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.3 ROUND_TO_PAGES

ULONG
ROUND_TO_PAGES(
    IN ULONG *Size*
)

Routine Description:

The ROUND_TO_PAGES macro takes a size in bytes and rounds it up to a multiple of the page size.

Parameters:

Size - Size in bytes to round up to a page multiple.

Return Value:

Returns the size rounded up to a multiple of the page size.

10.4 BYTE_OFFSET

ULONG
BYTE_OFFSET(
    IN PVOID *Va*
)

Routine Description:

The BYTE_OFFSET macro takes a virtual address and returns the byte offset of that address within the page.

Parameters:

Va - Virtual address.

Return Value:

Returns the byte offset portion of the virtual address.

10.5 ADDRESS_AND_SIZE_TO_SPAN_PAGES

ULONG
ADDRESS_AND_SIZE_TO_SPAN_PAGES(
    IN PVOID *Va***,**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN ULONG *Size*
)

Routine Description:

The ADDRESS_AND_SIZE_TO_SPAN_PAGES macro takes a virtual address and size and returns the number of pages spanned by the size.

Parameters:

Va - Virtual address.

Size - Size in bytes.

Return Value:

Returns the number of pages spanned by the size.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

11. Manage Object Handles and Handle Tables

This module implements a set of functions for supporting handles. Handles are opaque pointers that are implemented as indexes into a handle table.

Access to handle tables is serialized with a mutex. The level number associated with the mutex is specified at the time the handle table is created. Also specified at creation time are the initial size of the handle table, the memory pool type to allocate the table from and the size of each entry in the handle table.

The size of each entry in the handle table is specified as a power of 2. The size specifies how many 32-bit values are to be stored in each handle table entry. Thus a size of zero, specifies 1 (==2**0) 32-bit value. A size of 2 specifies 4 (=2**2) 32-bit values. The ability to support different sizes of handle table entries leads to some polymorphic interfaces.

The polymorphism occurs in two of the interfaces, ExCreateHandle and ExMapHandleToPointer. ExCreateHandle takes a handle table and a pointer. For handle tables whose entry size is one 32-bit value, the pointer parameter will be the value of the created handle. For handle tables whose entry size is more than one, the pointer parameter is a pointer to the 32-bit handle values which will be copied to the newly created handle table entry.

ExMapHandleToPointer takes a handle table and a handle parameter. For handle tables whose entry size is one, it returns the 32-bit value stored in the handle table entry. For handle tables whose entry size is more than one, it returns a pointer to the handle table entry itself. In both cases, ExMapHandleToPointer LEAVES THE HANDLE TABLE LOCKED. The caller must then call the ExUnlockHandleTable function to unlock the table when they are done referencing the contents of the handle table entry.

Free handle table entries are kept on a free list. The head of the free list is in the handle table header. To distinguish free entries from busy entries, the low order bit of the first 32-bit word of a free handle table entry is set to one. This means that the value associated with a handle can't have the low order bit set.

11.1 ExCreateHandleTable

PVOID
ExCreateHandleTable(
    IN ULONG *InitialCountTableEntries*,
    IN ULONG *CountTableEntriesToGrowBy*,
    IN ULONG *LogSizeTableEntry*,
    IN ULONG *TableMutexLevel*,

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN ULONG *SerialNumberMask*
)

Routine Description:

This function creates a handle table for storing opaque pointers. A handle is an index into a handle table.

Parameters:

InitialCountTableEntries - Initial size of the handle table.

CountTableEntriesToGrowBy - Number of entries to grow the handle table by when it becomes full.

LogSizeTableEntry - Log, base 2, of the number of 32-bit values in each handle table entry.

TableMutexLevel - The level number to associated with the mutex that is used to synchronize access to the handle table.

SerialNumberMask - If non-zero then the last 32-bit value in each handle table entry is supposed to contain a serial number and the value of this parameter is used to mask off bits that are not part of the serial number value.

Return Value:

An opaque pointer to the handle table. Returns NULL if an error occurred. The following errors can occur:

- Insufficient memory

11.2 ExLockHandleTable

VOID
ExLockHandleTable(
IN PVOID *HandleTableHandle*
)

Routine Description:

This function acquires the mutex for the specified handle table. After acquiring the mutex, it then acquired the spin lock for the specified handle table and sets

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

the MutexOwned flag in the handle table to TRUE before releasing the spin lock.

The purpose of the dual level locking is so that ExMapHandleToPointer can do it's work by just acquiring the spin lock.

Parameters:

HandleTableHandle - An opaque pointer to a handle table

Return Value:

None.

11.3 ExUnlockHandleTable

VOID
ExUnlockHandleTable(
    IN PVOID *HandleTableHandle*,
    IN BOOLEAN *ReleaseMutex*
)

Routine Description:

This function releases the spin lock associated the specified handle table. If the ReleaseMutex parameter is TRUE then the mutex associated with the handle table is also released, before releasing the spin lock.

Parameters:

HandleTableHandle - An opaque pointer to a handle table

ReleaseMutex - A flag indicated whether or not to release the mutex associated with the specified handle table.

Return Value:

None.

11.4 ExDupHandleTable

PVOID
ExDupHandleTable(
    IN PVOID *HandleTableHandle*,
    IN EX_DUPLICATE_HANDLE_ROUTINE *DupHandleProcedure* **OPTIONAL**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

)

Routine Description:

This function creates a duplicate copy of the specified handle table.

Parameters:

HandleTableHandle - An opaque pointer to a handle table

DupHandleProcedure - A pointer to a procedure to call for each valid handle in the duplicated handle table.

Return Value:

An opaque pointer to the handle table. Returns NULL if an error occurred. The following errors can occur:

- Insufficient memory

11.5 ExDestroyHandleTable

VOID
ExDestroyHandleTable(
IN PVOID *HandleTableHandle***,**
IN EX_DESTROY_HANDLE_ROUTINE *DestroyHandleProcedure* **OPTIONAL**
)

Routine Description:

This function destorys the specified handle table. It first locks the handle table to prevent others from accessing it, and then invalidates the handle table and frees the memory associated with it.

Parameters:

HandleTableHandle - An opaque pointer to a handle table

DestroyHandleProcedure - A pointer to a procedure to call for each valid handle in the handle table being destroyed.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

11.6 ExDumpHandleTable

VOID
ExDumpHandleTable(
    IN PVOID *HandleTableHandle*,
    IN EX_DUMP_HANDLE_ROUTINE *DumpHandleProcedure* **OPTIONAL,**
    IN PVOID *Stream* **OPTIONAL**
)

Routine Description:

This function prints out a formatted dump of the specified handle table.

Parameters:

HandleTableHandle - an opaque pointer to a handle table.

DumpHandleProcedure - A pointer to a procedure to call for each valid handle in the handle table being dumped.

Stream - I/O stream to send the output to. Defaults to stdout.

Return Value:

None.

11.7 ExEnumHandleTable

BOOLEAN
ExEnumHandleTable(
    IN PVOID *HandleTableHandle*,
    IN EX_ENUMERATE_HANDLE_ROUTINE *EnumHandleProcedure*,
    IN PVOID *EnumParameter*,
    OUT PHANDLE *Handle* **OPTIONAL**
)

Routine Description:

This function enumerates all the valid handles in a handle table. For each valid handle in the handle table, this functions calls an enumeration procedure specified by the caller. If the enumeration procedure returns TRUE, then the enumeration is stop, the current handle is returned to the caller via the optional Handle parameter and this function returns TRUE to indicated that the enumeration stopped at a specific handle.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

HandleTableHandle - An opaque pointer to a handle table.

EnumHandleProcedure - A pointer to a procedure to call for each valid handle in the handle table being enumerated.

EnumParameter - An unterpreted 32-bit value that is passed to the EnumHandleProcedure each time it is called.

Handle - An optional pointer to a variable that will receive the Handle value that the enumeration stopped at. Contents of the variable only valid if this function returns TRUE.

Return Value:

TRUE if the enumeration stopped at a specific handle. FALSE otherwise.

11.8 ExCreateHandle

HANDLE
ExCreateHandle(
    IN PVOID *HandleTableHandle*,
    IN PVOID *Pointer*
)

Routine Description:

This function create a handle in the specified handle table. If there is insufficient room in the handle table for a new entry, then the handle table is reallocated to a larger size.

Parameters:

HandleTableHandle - An opaque pointer to a handle table

Pointer - Initial value of the handle table entry if the entry size is one. The low order bit must be zero. If the entry size is not one, then it is a pointer to an array of 32-bit values that are the initial value of the handle table entry. The number of 32-bit values in the array is the size of each handle table entry. The low order bit of the first 32-bit value in the array must be zero.

Return Value:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The handle created or NULL if an error occurred. The following errors can occur:

- Invalid handle table

- Low order bit of the first pointer is not zero

- Insufficient memory

11.9 ExDestroyHandle

BOOLEAN
ExDestroyHandle(
    IN PVOID *HandleTableHandle*,
    IN HANDLE *Handle*
)

Routine Description:

This function removes a handle from a handle table.

Parameters:

HandleTableHandle - An opaque pointer to a handle table

Handle - Handle returned by ExCreateHandle for this handle table

Return Value:

Returns TRUE if the handle was successfully deleted from the handle table. Returns FALSE otherwise.

11.10 ExMapHandleToPointer

BOOLEAN
ExMapHandleToPointer(
    IN PVOID *HandleTableHandle*,
    IN HANDLE *Handle*,
    OUT PVOID *HandleValue*
)

Routine Description:

This function maps a handle into a pointer. It always returns with the handle table locked, so the caller must call ExUnlockHandleTable.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

HandleTableHandle - An opaque pointer to a handle table

Handle - Handle returned by ExCreateHandle for this handle table

HandleValue - A pointer to a variable that is to receive the value of the handle. If the passed handle table has a handle table entry size of one, then HandleValue is the 32-bit value associated with the passed handle. If the handle table entry size is more than one, then HandleValue is a pointer to the handle table entry itself.

Return Value:

This function returns TRUE if the handle table mutex was acquired and FALSE if just the handle table spin lock was acquired. The return value of this function should be passed as the ReleaseMutex parameter to the ExUnlockHandleTable function.

If the returned value is FALSE and the HandleValue variable is set to NULL, then an error occurred. The following errors can occur:

- Invalid handle table

- Invalid handle

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

12. Probe and Validate Arguments

This module contains the routine to probe variable length buffers for read or write accessibility and to ensure correct alignment.

12.1 ProbeForRead

VOID
ProbeForRead(
  IN PVOID *Address***,**
  IN ULONG *Length***,**
  IN ULONG *Alignment*
)

Routine Description:

This function probes a structure for read accessibility and ensures correct alignment of the structure. If the structure is not accessible or has incorrect alignment, then an exception is raised.

Parameters:

Address - Supplies a pointer to the structure to be probed.

Length - Supplies the length of the structure.

Alignment - Supplies the required alignment of the structure expressed as the number of bytes in the primitive datatype (e.g., 1 for char, 2 for short, 4 for long, and 8 for quad).

Return Value:

None.

12.2 ProbeForWrite

VOID
ProbeForWrite(
  IN PVOID *Address***,**
  IN ULONG *Length***,**
  IN ULONG *Alignment*
)

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function probes a structure for write accessibility and ensures correct alignment of the structure. If the structure is not accessible or has incorrect alignment, then an exception is raised.

Parameters:

Address - Supplies a pointer to the structure to be probed.

Length - Supplies the length of the structure.

Alignment - Supplies the required alignment of the structure expressed as the number of bytes in the primitive datatype (e.g., 1 for char, 2 for short, 4 for long, and 8 for quad).

Return Value:

None.

12.3 ProbeAndReadChar

CHAR
ProbeAndReadChar(
  IN PCHAR *Address*
)

12.4 ProbeAndReadUchar

UCHAR
ProbeAndReadUchar(
  IN PUCHAR *Address*
)

12.5 ProbeAndReadShort

SHORT
ProbeAndReadShort(
  IN PSHORT *Address*
)

12.6 ProbeAndReadLong

LONG
ProbeAndReadLong(
  IN PLONG *Address*
)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

12.7 ProbeAndReadUlong

```
ULONG
ProbeAndReadUlong(
        IN PULONG Address
        )
```

12.8 ProbeAndReadQuad

```
QUAD
ProbeAndReadQuad(
        IN PQUAD Address
        )
```

12.9 ProbeAndReadUquad

```
UQUAD
ProbeAndReadUquad(
        IN PUQUAD Address
        )
```

12.10 ProbeAndReadHandle

```
HANDLE
ProbeAndReadHandle(
        IN PHANDLE Address
        )
```

12.11 ProbeAndReadBoolean

```
BOOLEAN
ProbeAndReadBoolean(
        IN PBOOLEAN Address
        )
```

12.12 ProbeForWriteChar

```
CHAR
ProbeForWriteChar(
        IN PCHAR Address
        )
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

12.13 ProbeForWriteUchar

UCHAR
ProbeForWriteUchar(
    IN PUCHAR *Address*
)

12.14 ProbeForWriteShort

SHORT
ProbeForWriteShort(
    IN PSHORT *Address*
)

12.15 ProbeForWriteUshort

USHORT
ProbeForWriteUshort(
    IN PUSHORT *Address*
)

12.16 ProbeForWriteLong

LONG
ProbeForWriteLong(
    IN PLONG *Address*
)

12.17 ProbeForWriteUlong

ULONG
ProbeForWriteUlong(
    IN PULONG *Address*
)

12.18 ProbeForWriteQuad

QUAD
ProbeForWriteQuad(
    IN PQUAD *Address*
)

12.19 ProbeForWriteUquad

UQUAD
ProbeForWriteUquad(

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        IN PUQUAD Address
        )
```

12.20 ProbeForWriteHandle

```
HANDLE
ProbeForWriteHandle(
        IN PHANDLE Address
        )
```

12.21 ProbeForWriteBoolean

```
BOOLEAN
ProbeForWriteBoolean(
        IN PBOOLEAN Address
        )
```

12.22 ProbeAndWriteChar

```
CHAR
ProbeAndWriteChar(
        IN PCHAR Address
        )
```

12.23 ProbeAndWriteUchar

```
UCHAR
ProbeAndWriteUchar(
        IN PUCHAR Address
        )
```

12.24 ProbeAndWriteShort

```
SHORT
ProbeAndWriteShort(
        IN PSHORT Address
        )
```

12.25 ProbeAndWriteUshort

```
USHORT
ProbeAndWriteUshort(
        IN PUSHORT Address
        )
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

12.26 ProbeAndWriteLong

LONG
ProbeAndWriteLong(
  IN PLONG *Address*
)

12.27 ProbeAndWriteUlong

ULONG
ProbeAndWriteUlong(
  IN PULONG *Address*
)

12.28 ProbeAndWriteQuad

QUAD
ProbeAndWriteQuad(
  IN PQUAD *Address*
)

12.29 ProbeAndWriteUquad

UQUAD
ProbeAndWriteUquad(
  IN PUQUAD *Address*
)

12.30 ProbeAndWriteHandle

HANDLE
ProbeAndWriteHandle(
  IN PHANDLE *Address*
)

12.31 ProbeAndWriteBoolean

BOOLEAN
ProbeAndWriteBoolean(
  IN PBOOLEAN *Address*
)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

13. Perform Restricted Interlock Operations

This module implements functions to support interlocked operations in a general way such that all the data that is operated on can be pageable including the locks themselves.

NOTE: The code in this module has been very carefully aligned such that no interlocked routine can cross a page boundary. Care must be taken when making any changes to this module to ensure that a page crossing does not occur.

13.1 ExInterlockedAddLong

LONG
ExInterlockedAddLong(
    IN PLONG *Addend***,**
    IN LONG *Increment***,**
    IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function performs an interlocked add of an increment value to an addend variable of type long. The initial value of the addend variable is returned as the function value.

Parameters:

Addend (r16) - Supplies a pointer to a variable whose value is to be adjusted by the increment value.

Increment (r17) - Supplies the increment value to be added to the addend variable.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the addend variable.

Return Value:

The initial value of the addend variable.

13.2 ExInterlockedAddShort

SHORT
ExInterlockedAddShort(

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        IN PSHORT Addend,
        IN SHORT Increment,
        IN PKSPIN_LOCK Lock
        )
```

Routine Description:

This function performs an interlocked add of an increment value to an addend variable of type short. The initial value of the addend variable is returned as the function value.

Parameters:

Addend (r16) - Supplies a pointer to a variable whose value is to be adjusted by the increment value.

Increment (r17) - Supplies the increment value to be added to the addend variable.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the addend variable.

Return Value:

The initial value of the addend variable.

13.3 ExInterlockedInsertHeadList

```
VOID
ExInterlockedInsertHeadList(
        IN PLIST_ENTRY ListHead,
        IN PLIST_ENTRY ListEntry,
        IN PKSPIN_LOCK Lock
        )
```

Routine Description:

This function inserts an entry at the head of a doubly linked list so that access to the list is synchronized in a multiprocessor system.

Parameters:

ListHead (r16) - Supplies a pointer to the head of the doubly linked list into which an entry is to be inserted.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ListEntry (r17) - Supplies a pointer to the entry to be inserted at the head of the list.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

None.

13.4 ExInterlockedInsertTailList

VOID
ExInterlockedInsertTailList(
  IN PLIST_ENTRY *ListHead*,
  IN PLIST_ENTRY *ListEntry*,
  IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function inserts an entry at the tail of a doubly linked list so that access to the list is synchronized in a multiprocessor system.

Parameters:

ListHead (r16) - Supplies a pointer to the head of the doubly linked list into which an entry is to be inserted.

ListEntry (r17) - Supplies a pointer to the entry to be inserted at the tail of the list.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

None.

13.5 ExInterlockedRemoveHeadList

PLIST_ENTRY
ExInterlockedRemoveHeadList(
  IN PLIST_ENTRY *ListHead*,
  IN PKSPIN_LOCK *Lock*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

)

Routine Description:

This function removes an entry from the head of a doubly linked list so that access to the list is synchronized in a multiprocessor system. If there are no entries in the list, then a value of NULL is returned. Otherwise, the address of the entry that is removed is returned as the function value.

Parameters:

ListHead (r16) - Supplies a pointer to the head of the doubly linked list from which an entry is to be removed.

Lock (r17) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

The address of the entry removed from the list, or NULL if the list is empty.

13.6 ExInterlockedPopEntryList

PSINGLE_LIST_ENTRY
ExInterlockedPopEntryList(
IN PSINGLE_LIST_ENTRY *ListHead*,
IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function removes an entry from the front of a singly linked list so that access to the list is synchronized in a multiprocessor system. If there are no entries in the list, then a value of NULL is returned. Otherwise, the address of the entry that is removed is returned as the function value.

Parameters:

ListHead (r16) - Supplies a pointer to the head of the singly linked list from which an entry is to be removed.

Lock (r17) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The address of the entry removed from the list, or NULL if the list is empty.

13.7 ExInterlockedPushEntryList

VOID
ExInterlockedPushEntryList(
    IN PSINGLE_LIST_ENTRY *ListHead*,
    IN PSINGLE_LIST_ENTRY *ListEntry*,
    IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function inserts an entry at the head of a singly linked list so that access to the list is synchronized in a multiprocessor system.

Parameters:

ListHead (r16) - Supplies a pointer to the head of the singly linked list into which an entry is to be inserted.

ListEntry (r17) - Supplies a pointer to the entry to be inserted at the head of the list.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

14. Allocate and Free Spin Locks

This module implements the executive functions to allocate and free spin locks.

14.1 ExAllocateSpinLock

VOID
ExAllocateSpinLock(
  IN PKSPIN_LOCK *SpinLock*
)

Routine Description:

This function allocates and initializes a spin lock.

Parameters:

SpinLock - Supplies a pointer to a spin lock.

Return Value:

None.

14.2 ExFreeSpinLock

VOID
ExFreeSpinLock(
  IN PKSPIN_LOCK *SpinLock*
)

Routine Description:

This function frees a previously allocated spin lock.

Parameters:

SpinLock - Supplies a pointer to a spin lock.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

15. Perform General Interlocked Operations

This module implements functions to support interlocked operations in a general way such that all the data that is operated on can be pageable including the locks themselves.

NOTE: The code in this module has been very carefully aligned such that no interlocked routine can cross a page boundary. Care must be taken when making any changes to this module to ensure that a page crossing does not occur.

15.1 RtlInterlockedAddLong

LONG
RtlInterlockedAddLong(
    IN PLONG *Addend***,**
    IN LONG *Increment***,**
    IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function performs an interlocked add of an increment value to an addend variable of type long. The initial value of the addend variable is returned as the function value.

Parameters:

Addend (r16) - Supplies a pointer to a variable whose value is to be adjusted by the increment value.

Increment (r17) - Supplies the increment value to be added to the addend variable.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the addend variable.

Return Value:

The initial value of the addend variable.

15.2 RtlInterlockedAddShort

SHORT
RtlInterlockedAddShort(

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN PSHORT *Addend***,**
IN SHORT *Increment***,**
IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function performs an interlocked add of an increment value to an addend variable of type short. The initial value of the addend variable is returned as the function value.

Parameters:

Addend (r16) - Supplies a pointer to a variable whose value is to be adjusted by the increment value.

Increment (r17) - Supplies the increment value to be added to the addend variable.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the addend variable.

Return Value:

The initial value of the addend variable.

15.3 RtlInterlockedInsertHeadList

VOID
RtlInterlockedInsertHeadList(
IN PLIST_ENTRY *ListHead***,**
IN PLIST_ENTRY *ListEntry***,**
IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function inserts an entry at the head of a doubly linked list so that access to the list is synchronized in a multiprocessor system.

Parameters:

ListHead (r16) - Supplies a pointer to the head of the doubly linked list into which an entry is to be inserted.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ListEntry (r17) - Supplies a pointer to the entry to be inserted at the head of the list.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

None.

15.4 RtlInterlockedInsertTailList

VOID
RtlInterlockedInsertTailList(
IN PLIST_ENTRY *ListHead*,
IN PLIST_ENTRY *ListEntry*,
IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function inserts an entry at the tail of a doubly linked list so that access to the list is synchronized in a multiprocessor system.

Parameters:

ListHead (r16) - Supplies a pointer to the head of the doubly linked list into which an entry is to be inserted.

ListEntry (r17) - Supplies a pointer to the entry to be inserted at the tail of the list.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

None.

15.5 RtlInterlockedRemoveHeadList

PLIST_ENTRY
RtlInterlockedRemoveHeadList(
IN PLIST_ENTRY *ListHead*,
IN PKSPIN_LOCK *Lock*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

)

<u>Routine Description:</u>

This function removes an entry from the head of a doubly linked list so that access to the list is synchronized in a multiprocessor system. If there are no entries in the list, then a value of NULL is returned. Otherwise, the address of the entry that is removed is returned as the function value.

<u>Parameters:</u>

ListHead (r16) - Supplies a pointer to the head of the doubly linked list from which an entry is to be removed.

Lock (r17) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

<u>Return Value:</u>

The address of the entry removed from the list, or NULL if the list is empty.

15.6 RtlInterlockedRemoveHeadList

PLIST_ENTRY
RtlInterlockedRemoveHeadList(
IN PLIST_ENTRY *ListHead***,**
IN PKSPIN_LOCK *Lock*
)

<u>Routine Description:</u>

This function removes an entry from the head of a doubly linked list so that access to the list is synchronized in a multiprocessor system. If there are no entries in the list, then a value of NULL is returned. Otherwise, the address of the entry that is removed is returned as the function value.

<u>Parameters:</u>

ListHead (r16) - Supplies a pointer to the head of the doubly linked list from which an entry is to be removed.

Lock (r17) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

<u>Return Value:</u>

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The address of the entry removed from the list, or NULL if the list is empty.

15.7 RtlInterlockedPopEntryList

PSINGLE_LIST_ENTRY
RtlInterlockedPopEntryList(
    IN PSINGLE_LIST_ENTRY *ListHead***,**
    IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function removes an entry from the front of a singly linked list so that access to the list is synchronized in a multiprocessor system. If there are no entries in the list, then a value of NULL is returned. Otherwise, the address of the entry that is removed is returned as the function value.

Parameters:

ListHead (r16) - Supplies a pointer to the head of the singly linked list from which an entry is to be removed.

Lock (r17) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

The address of the entry removed from the list, or NULL if the list is empty.

15.8 RtlInterlockedPushEntryList

VOID
RtlInterlockedPushEntryList(
    IN PSINGLE_LIST_ENTRY *ListHead***,**
    IN PSINGLE_LIST_ENTRY *ListEntry***,**
    IN PKSPIN_LOCK *Lock*
)

Routine Description:

This function inserts an entry at the head of a singly linked list so that access to the list is synchronized in a multiprocessor system.

Parameters:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ListHead (r16) - Supplies a pointer to the head of the singly linked list into which an entry is to be inserted.

ListEntry (r17) - Supplies a pointer to the entry to be inserted at the head of the list.

Lock (r18) - Supplies a pointer to a spin lock to be used to synchronize access to the list.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

16. Perform Operations on Counted Strings

This module defines functions for manipulating counted strings (STRING). A counted string is a data structure containing three fields. The Buffer field is a pointer to the string itself. The MaximumLength field contains the maximum number of bytes that can be stored in the memory pointed to by the Buffer field. The Length field contains the current length, in bytes, of the string pointed to by the Buffer field. Users of counted strings should not make any assumptions about the existence of a null byte at the end of the string, unless the null byte is explicitly included in the Length of the string.

16.1 RtlInitString

VOID
RtlInitString(
OUT PSTRING *DestinationString*,
IN PSZ *SourceString* **OPTIONAL**
)

Routine Description:

The RtlInitString function initializes a Windows NT counted string. The DestinationString is initialized to point to the SourceString and the Length and MaximumLength fields of DestinationString are initialized to the length of the SourceString, which is zero if SourceString is not specified.

Parameters:

DestinationString - Pointer to the counted string to initialize

SourceString - Optional pointer to a null terminated string that the counted string is to point to.

Return Value:

None.

16.2 RtlCopyString

VOID
RtlCopyString(
OUT PSTRING *DestinationString*,
IN PSTRING *SourceString* **OPTIONAL**
)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Routine Description:

The RtlCopyString function copies the SourceString to the DestinationString. If SourceString is not specified, then the Length field of DestinationString is set to zero. The MaximumLength and Buffer fields of DestinationString are not modified by this function.

The number of bytes copied from the SourceString is either the Length of SourceString or the MaximumLength of DestinationString, whichever is smaller.

Parameters:

DestinationString - Pointer to the destination string.

SourceString - Optional pointer to the source string.

Return Value:

None.

16.3 RtlCompareString

LONG
RtlCompareString(
  IN PSTRING *String1*,
  IN PSTRING *String2*,
  IN BOOLEAN *CaseInSensitive*
)

Routine Description:

The RtlCompareString function compares two counted strings. The return value indicates if the strings are equal or String1 is less than String2 or String1 is greater than String2.

The CaseInSensitive parameter specifies if case is to be ignored when doing the comparison.

Parameters:

*String*1 - Pointer to the first string.

*String*2 - Pointer to the second string.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

CaseInsensitive - TRUE if case should be ignored when doing the comparison.

Return Value:

Signed value that gives the results of the comparison:

Zero - String1 equals String2

< Zero - String1 less than String2

> Zero - String1 greater than String2

16.4 RtlEqualString

```
BOOLEAN
RtlEqualString(
    IN PSTRING String1,
    IN PSTRING String2,
    IN BOOLEAN CaseInSensitive
    )
```

Routine Description:

The RtlEqualString function compares two counted strings for equality.

The CaseInSensitive parameter specifies if case is to be ignored when doing the comparison.

Parameters:

*String*1 - Pointer to the first string.

*String*2 - Pointer to the second string.

CaseInsensitive - TRUE if case should be ignored when doing the comparison.

Return Value:

Boolean value that is TRUE if String1 equals String2 and FALSE otherwise.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

17. Debugging Support Functions

This module implements functions to support debugging Windows NT. Each function executes a trap r31,r29,r0 instruction with a special value in R31. The simulator decodes this trap instruction and dispatches to the correct piece of code in the simulator based on the value in R31. See the simscal.c source file in the simulator source directory.

17.1 DbgBreakPoint

VOID
DbgBreakPoint()

Routine Description:

This function executes a breakpoint instruction. Useful for enter the debugger under program control.

Parameters:

None.

Return Value:

None.

17.2 DbgCommand

VOID
DbgCommand(
PCH *Command***,**
ULONG *Parameter*
)

Routine Description:

This function passes a string to the debugger to execute as if it was type by the user.

Parameters:

Command - a pointer to a string that contains one or more debugger commands. Multiple commands are separated by either a semicolon or newline character.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameter - a 32 bit parameter that is stored in $9 simulator variable.

Return Value:

None.

17.3 DbgQueryInstructionCounter

ULONG
DbgQueryInstructionCounter()

Routine Description:

This function returns the current value of the i860 simulator's instruction counter.

Parameters:

None.

Return Value:

32 bit instruction counter.

17.4 DbgPrint

ULONG
DbgPrint(
IN PCH *Format*
)

Routine Description:

This function displays a formatted string on the debugging console. The syntax of it's argments is the same as accepted by the Microsoft C Runtime printf routines with the addition of the following format specifiers:

S - argument is a PSTRING (pointer to STRING)

Parameters:

Format - specifies a pointer to the format string.

Remaining arguments are variable and depend upon the contents of the format string. Maximum of 8 arguments may be specified.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

Number of characters displayed on the debugging console.

17.5 DbgPrompt

ULONG
DbgPrompt(
IN PCH *Prompt*,
OUT PCH *Response*,
IN ULONG *MaximumResponseLength*
)

Routine Description:

This function displays the prompt string on the debugging console and then reads a line of text from the debugging console. The line read is returned in the memory pointed to by the second parameter. The third parameter specifies the maximum number of characters that can be stored in the response area.

Parameters:

Prompt - specifies the text to display as the prompt.

Response - specifies where to store the response read from the debugging console.

Prompt - specifies the maximum number of characters that can be stored in the Response buffer.

Return Value:

Number of characters stored in the Response buffer. Includes the terminating newline character, but not the null character after that.

17.6 DbgLoadImageFileSymbols

ULONG
DbgLoadImageFileSymbols(
IN PCH *FileName*
)

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function attempts to load any symbolic debugging information from an image file into the debugger.

Parameters:

FileName - specifies the name of the image file to load symbols from.

Return Value:

Returns 0 if the image file is not found or is not a valid image file. Otherwise returns the entry point address from the image file header.

17.7 DbgSetDirBaseForImage

VOID
DbgSetDirBaseForImage(
  IN PCH *ImagePathName***,**
  IN ULONG *DirBase*
)

Routine Description:

This function identifies the dirbase value to associate with an image file whose sybols have been loaded with the DbgLoadImageFileSymbols function. The first parameter should point to the path name returned by the DbgLoadImageFileSymbols function. The second parameter is the 20 bit DTB value that is associated with the process into which the image file was loaded.

Parameters:

ImagePathName - specifies the fully qualified path name of the image file that has been loaded into a Windows NT address space.

DirBase - specifies the 20 bit DTB value that is associated with the Windows NT process that will run the image file.

Return Value:

None.

17.8 DbgKillDirBase

VOID
DbgKillDirBase(
  IN ULONG *DirBase*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

)

Routine Description:

This function tells the debugger when a particular process context is being destroyed. This allows the debugger to remove any process specific breakpoints from its breakpoint table.

Parameters:

DirBase - the 20 bit DTB value that is associated with the Windows NT process that is being destroyed.

Return Value:

None.

17.9 DbgCheckpointSimulator

BOOLEAN
DbgCheckpointSimulator(
IN PCH *FileName* **OPTIONAL**
)

Routine Description:

This function saves the entire state of the i860 simulator to the specified file. It returns FALSE when the checkpoint operation is completed. It returns TRUE when the function returns due to having been restarted.

Parameters:

FileName - an optional parameter that specifies the name of the file to save the state of the simulator in. If not specified, then the file name defaults to the image file name with a .CHK extension.

Return Value:

Returns FALSE when the checkpoint is complete. Returns TRUE if the simulator has been restarted from the checkpoint file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 File System Design Note

Author: *Gary D. Kimura*

Revision 1.0, Sep 11, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction..1
 1.1 Function of File System 1
 1.2 NT OS/2 Environment for the File System...1

2. File System Operations...2
 2.1 FSD/FSP Dispatch and Communication..2
 2.2 File System Initialization 3
 2.3 Before the First Operation on a Volume...3
 2.4 Open and Create File Operations 5
 2.5 Read, Write, Set, and Query File Operations..6
 2.6 Close File Operation 7

3. Loose Ends..7

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This design note describes the implementation of an NT OS/2 file system. All NT OS/2 file systems share similar properties in their communication with the NT OS/2 I/O system, and their basic internal flow of control. The information presented here is based on knowledge gained while implementing the FAT file system for NT OS/2, but also contains information applicable to all NT OS/2 file systems.

Before reading this design note, the reader should be familiar with the "NT OS/2 I/O System Specification" document and the section within the "NT OS/2 Memory Management Design Note" regarding I/O support. The I/O system specification presents a high level view of the NT OS/2 user API calls and the support routines it provides for use by the various NT OS/2 file systems and device drivers. This design augments the I/O system document by describing in detail how a file system actually ties into the I/O system, in terms of the I/O system's major data structures and flow of control.

This design note presents a tour of the communication that occurs between the NT OS/2 I/O system and the file system. It is intended as a guide to file system programmers in understanding how the file system interacts with the I/O system, and avoids discussing file system internals, such as its internal data structures or its resource locking mechanism and granularity.

1.1 Function of File System

The task of the file system is to handle user (and system) generated requests to read and write files to or from a disk volume. The user passes all I/O requests to the I/O system (via calls such as NtReadFile) which in turn passes the request to the appropriate file system or device driver in a structure called an I/O request packet (IRP). The IRP contains a function code and parameters appropriate for the function. For example, to open a file the IRP contains a code indicating an open file operation and a file name parameter[1].

In the case of a file system the IRP also contains a pointer to the device object denoting the target device that the file system should direct all sector read and write requests towards. That is, to actually read a sector on a device the file system calls back to the I/O system with information from the IRP indicating which device to read.

In this sense, the file system is not tied to any physical device. From the standpoint of the I/O system, a file system acts more like a filter. In NT OS/2, disk drivers only deal with reading and writing physical device sectors. The primary job of the file system is to add structure to this device. So a user or system request to manipulate a file goes from the I/O system to the file system where it is passed back to the I/O system as actual reads and writes of sectors on a disk.

---

[1]The IRP actually contains additional open file parameters that are not discussed in this example.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.2 NT OS/2 Environment for the File System

An NT OS/2 file system exports three entry points and these entry points are only called by the I/O system (i.e., a user never directly calls the file system). There is an initialization routine, a dispatch routine, and an unload routine.

The initialization routine is called at system start up during I/O initialization. This routine is responsible for creating the file system's device object and for initializing any global data structures and processes/threads used by the file system. The structure of a file system's device object is covered later in this design note.

The dispatch routine is called by the I/O system to process all IRPs targeted for the file system. These include requests to mount a volume, open or create a file, read and write to a file, and close a file. Logically, the dispatch routine either completes the I/O request and then returns control to the I/O system, or it returns to the I/O system immediately and indicates that the I/O will be completed at a later time (i.e., its return status is STATUS_PENDING).

The unload routine is called by the I/O system when the file system is being removed from a running NT OS/2 system. The unload routine is responsible for cleaning up all of its global data structures, processes, and threads. After calling unload, the file system is essentially unavailable for use until its initialization routine is called again.

2. File System Operations

This section describes the implementation details of an NT OS/2 file system. It describes the basic flow of control through the file system and its communication with the I/O system.

2.1 FSD/FSP Dispatch and Communication

The term *File System Driver* (FSD) refers to a set of routines called by the I/O system (via the file system's dispatch routine). These are kernel-mode routines that execute in the same context in which they are called by the I/O system. That is, there is not a special process/thread or context switch associated with executing the FSD. Execution paths through the FSD should be fairly short and not require long waits (e.g., for a disk I/O to complete).

The workhorse of the file system is the *File System Process* (FSP). The term FSP refers to the set of threads executing within a dedicated file system process. Unlike the FSD, the FSP has the entire user address space available for use, and can dedicate threads to systematically attend to the tasks associated with an I/O request. An FSP thread can wait for I/O to complete and can easily maintain context unavailable to the FSD.

Figure 1 illustrates the relationship between the FSD and the FSP. The FSD is called by the I/O system with an IRP containing the requested operation. Then, based on the amount of work and context required by the request, the FSD either sends the necessary read and write requests to the target device object (as associated IRPs) or it enqueues the IRP to the work queue used by the FSP. If the request is queued to the FSP, the FSP dequeues the IRP and performs the work required to complete the request.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Figure 1 - FSD/FSP Layout

As a rule of thumb, file read and write requests can usually go through the FSD directly to the target device object. All other requests, such as file create, that require file system buffering or data structure editing go through the FSP.

Within a single file system, there are multiple work queues and an FSP thread dedicated to each queue. The FSP thread is a normal thread (i.e., it is merely the name of a thread that is handling work queue activity). There is a queue to handle mount requests and a queue for each mounted volume. The construction and use of each work queue is described in the following subsections.

2.2 File System Initialization

The file system initialization procedure is called at system start up during I/O initialization. This procedure is responsible for creating the device object for the file system, creating the FSP process, and starting the FSP mount thread. It also initializes all of the global data structures used by the file system. It is called with a pointer to the driver object for the file system created by the I/O system.

The device object created at initialization is called the *file system device object* and contains the name of the file system (e.g., "\Fat") and a work queue. All mount requests are sent by the FSD dispatcher to this work queue and are processed by a dedicated FSP mount thread. Figure 2 illustrates the relationship between the file system device object and the FSP mount thread.

Figure 2 - File System Initialization

In summary, the initialization phase creates a file system device object and sets a pointer to it in the Driver Object. It also creates the FSP process and FSP mount thread. The FSP mount thread only processes requests that are placed in the file system device object's work queue, and the only type of requests valid in that work queue are mount volume requests.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.3 Before the First Operation on a Volume

When a user calls NtCreateFile, one of the first things the I/O system must decide is whether the volume targeted by the request is mounted. If this is the first operation on the volume, then the volume is not mounted and the I/O system must send an IRP to the file system requesting that the volume be mounted. After the volume is mounted, the I/O system can then forward the create file request.

On a mount request, the I/O system calls the file system FSD dispatch routine passing as input parameters a pointer to the file system device object and an IRP (See Figure 3a). The IRP contains two parameters of interest[2]. There is a pointer to a volume parameter block (VPB) and to a targeted device object. The VPB is a structure used to denote a disk volume. It will contain, after the volume is mounted, the volume label and its serial number. Within the system, there is one VPB for every mounted volume.

The targeted device object is the device object that is to be used by the file system when it issues a read or write sector request. The VPB also contains a pointer to a device object (this field is set by the I/O system) called the real device (not shown in Figure 3). In most cases, the targeted device object and real device are identical and simply denote the disk driver containing the volume (e.g., a floppy or hard disk). However, to accommodate multi-volume disk sets and disk strippers, the targeted device object and the real device can be different device objects. The file system only sends request to the targeted device object and never to the real device.

Figure 3a shows the parameters and data structures sent to the file system FSD dispatch routine on a mount request. When it is called, the FSD dispatcher will enqueue the mount request to the FSP mount thread's work queue. Note that the work queue is located by the FSD dispatcher via the file system device object which is passed in as an input parameter.

Figure 3 - Processing a Mount Request

To process the mount request, the FSP mount thread issues the necessary read sector requests to the targeted device object to decide if the volume belongs to this file system (e.g., whether it is a Fat file system or a Pinball file system). If the mount is successful, the file system creates a new device object for the volume. This device object replaces the device object originally referenced by the VPB. It contains a work queue for the volume, and a volume control block (VCB). Upon

[2]An IRP actually contains more than two parameters, but only these two are germane to this discussion.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

completion of a successful mount operation, the file system also inserts in the VPB the volume label and serial number. Figure 3b shows the results after a successful mount request.

The FSP mount thread also creates a new FSP volume thread to process subsequent I/O requests targeted for the volume. That is, when the I/O system calls the FSD dispatcher to do I/O to a mounted volume, it will pass as input a pointer to the volume device object and not the file system device object. The FSD dispatcher will then send the request to the work queue for the appropriate FSP volume thread.

A slightly different chain of events takes place if the volume being mounted has previously been mounted by the file system. That is, the file system already possesses a VCB for the volume. This situation can occur with removable media where the I/O system is attempting an operation on what it thinks is a new volume, when in reality the volume was mounted in a different drive at an earlier time.

To handle the remount case, the file system will keep track of every device object that is mounted and search this list whenever a new mount request is processed. If a match with a previous volume is found, the file system will then simply replace the new VPB it is given as input with the old VPB. This simply requires changing the real device pointer found in the VPB and the target device object pointer found in the old VCB. The remount process is illustrated in Figure 4.

Figure 4 - Volume Remount

In summary, the file system's mount procedure is used to establish that the target device object contains a proper file system structure. If the file system recognizes the on disk structure, it creates a new device object for the volume in place of the file system device object, sets the volume label and serial number in the VPB, and starts up an FSP volume thread.

2.4 Open and Create File Operations

The open and create file[3] operation takes as input a file object denoting the file being opened and the device object (created by the file system) which is targeted for the operation. The FSD dispatcher queues the open and create file requests to the work queue for the appropriate FSP volume thread. Figure 5a illustrates the input given to the FSD dispatcher.

---

[3]This discussion also applies to opening and creating directories.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Figure 5 - Opening and Creating a File

The file object passed to the file system contains the file name being opened relative to the root of the volume and it contains the requested access mode. It is the job of the file system to open the file on the targeted device object. The information available to the file system is the device object of the volume (passed as input from the I/O system), the VCB and VPB associated with the volume, and a pointer to the targeted device object. The FSP volume thread is free to issue as many read and write sector commands to the targeted device object as necessary to open (create) the file.

When the file is successfully opened, the file system creates a file control block (FCB)[4] and a context control block (CCB) which are both used to store information about the opened file. Both of these records are referenced by the file object through its two FsContext pointers. The file object also contains a section object pointer which points to a reserved longword in the FCB. This longword is used by the memory management system. See Figure 5b.

The FCB and CCB are internal file system control structures. An FCB is created for every opened file on a volume. It contains the file name, mapping information, and a pointer to the VCB containing the file[5]. Only one FCB is created per opened file. The FCB is shared by multiple file objects if the file objects denote the same file. A CCB is created for every file object[6] and contains information specific to the file object, such as current file position information.

In summary, when opening a file, the file system is passed as input a file object denoting the file and a device object denoting the volume where the file is to exist. If the open operation is successful, the file system creates an FCB and a CCB record, both referenced by the file object.

2.5 Read, Write, Set, and Query File Operations

The basic operations on a file, after it is opened, are read, write, set, and query. These operations take as input a file object and a pointer to the device object for the volume. See Figure 6.

---

[4]If the operation opens a directory, then the structure is called a directory control block (DCB).

[5]This is a simplification of the actual FCB structure.

[6]Actually as an optimization it is only created when absolutely necessary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Figure 6 - Read, Write, Set, and Query File Input

When processing an IRP of this type, the FSD dispatcher can either send the IRP to the FSP work queue and let the FSP volume thread handle the request, or it can complete the request itself. The FSP can actually handle all types I/O requests; however, for efficiency, it is better to have the FSD handle the requests when possible. The criterion used to determine if the FSD can handle a request are the amount of time required to process the request and the type of resource locks needed to read or alter the file system's internal data structures. If a lot of extra processing is required or if too many time consuming locks are needed to perform the operation, then the FSP should handle the request.

These operations do not alter the infrastructure connecting the file object, device object, FCB, or CCB. However, they still may need to acquire an exclusive resource lock on the structures in order to alter their other fields (such as setting a delete flag or expanding file allocation).

2.6 Close File Operation

The last operation on a file is the close operation. This operation takes the same input as the preceding read, write, set, and query operations, but unlike those operations, the close operation modifies the infrastructure.

To do a close operation, the file system first performs any necessary I/O to the target device object to either close or delete the file. It then removes the FCB[7] and CCB from its internal data structure. After the close operation the file object is no longer valid and must be removed (or recycled) by the I/O system.

3. Loose Ends

---

[7]The FCB is removed only it if is no longer reference by any file object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Original Draft 1.0, September 7, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 File System Support Routines Specification

Author: *Gary D. Kimura*

Revision 1.0, August 10, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction... 1

2. Miscellaneous Support Macros.. 2
2.1 FsRtlCompleteRequest 2

3. Byte Range File Lock Routines 3
3.1 FsRtlInitializeFileLock 3
3.2 FsRtlUninitializeFileLock.. 4
3.3 FsRtlAreThereCurrentFileLocks 4
3.4 FsRtlProcessFileLock 5
3.5 FsRtlCheckLockForReadAccess 5
3.6 FsRtlCheckLockForWriteAccess 6
3.7 FsRtlGetNextFileLock 6

4. Name Support Routines 8
4.1 FsRtlFirstDbcsCharacter.. 8
4.2 FsRtlDissectDbcs 10
4.3 FsRtlUpcaseDbcs 11
4.4 FsRtlDbcsContainsWildCards 12
4.5 FsRtlCompareDbcs 12
4.6 FsRtlIsDbcsInExpression.. 13
4.7 FsRtlIsNameValid... 14
4.8 FsRtlIsPathValid 14
4.9 FsRtlIsLegalDbcsCharacter 15
4.10 FsRtlToUpperDbcsCharacter... 16

5. Mapped Control Block Routines 17
5.1 FsRtlInitializeMcb 18
5.2 FsRtlUninitializeMcb... 18
5.3 FsRtlAddMcbEntry.. 18
5.4 FsRtlRemoveMcbEntry 19
5.5 FsRtlLookupMcbEntry 20
5.6 FsRtlLookupLastMcbEntry 21
5.7 FsRtlNumberOfRunsInMcb... 21
5.8 FsRtlGetNextMcbEntry 22

6. Volume Mapped Control Block Routines 24
6.1 FsRtlInitializeVmcb... 25
6.2 FsRtlUninitializeVmcb 25
6.3 FsRtlSetMaximumLbnVmcb... 26
6.4 FsRtlAddVmcbMapping.. 26
6.5 FsRtlRemoveVmcbMapping 27
6.6 FsRtlVmcbVbnToLbn 27
6.7 FsRtlVmcbLbnToVbn 28

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.8 FsRtlSetDirtyVmcb.. 28
6.9 FsRtlSetCleanVmcb.. 29
6.10 FsRtlGetDirtySectorsVmcb 29

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes a library of file system support routines for use by the different file systems within **NT OS/2**. They are executive level routines that are too file system specific to belong in the Ex or public Rtl component, and are also inappropriate for the I/O component.

The name of this component is **FsRtl**. Each set of routines within **FsRtl** is tailored for supporting a specific file system function. Most components within **FsRtl** define an abstract data type and routines for manipulating the data. In addition, the header file for **FsRtl** defines some global data types common to multiple file systems.

The global data types defined within **FsRtl** are:

- Logical Block Number (**LBN**). **LBN** is the moniker used to identify physical blocks on the disk. The numbering sequence is from zero to *N*-1 where *N* is the number of sectors on the disk.

- Virtual Block Number (**VBN**). The **VBN** identifies the sectors of a file relative to the start of the file. A value of 0 corresponds to the first sector of data for a file, a value of 1 corresponds to the second sector, and so forth.

The individual categories of support provided by **FsRtl** are:

- Byte Range File Locks (**FILE_LOCK**). This package implements a set of routines for handling byte range file locking. The routines provide a consistent method of all file system to maintain and implement byte range file locks.

- Name Support. This package provides string manipulation routines and macros that are tailored for file system usage.

- Mapped Control Block (**MCB**). This package provides for in-memory retrieval mapping support. Retrieval mapping is the correspondence between LBN's and VBN's for a given file.

- Volume Mapped Control Block (**VMCB**). The Pinball and Fat file systems treat the ancillary structures of the on-disk file system as one large file, called the Volume File. This allows the file systems to utilize memory management for maintaining a cache of these sectors stored in memory. This package provides necessary support for constructing and maintaining an artificial mapping between LBNs and VBNs in the volume file.

- Notify Change Directory Routines. \\ still needs to be added \\

To use the FsRtl component a file system must explicitly include the file <fsrtl.h> (i.e., the file is not automatically included with <ntos.h>).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The remainder of this document describes, in detail, each of the preceding components, and also miscellaneous macros.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2. Miscellaneous Support Macros

This module defines all of the general File System Rtl routines

2.1 FsRtlCompleteRequest

VOID
FsRtlCompleteRequest(
IN PIRP *Irp***,**
IN NTSTATUS *Status*
)

Routine Description:

This routine is used to complete an IRP with the indicated status. It does the necessary raise and lower of IRQL.

Parameters:

Irp - Supplies a pointer to the Irp to complete

Status - Supplies the completion status for the Irp

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. Byte Range File Lock Routines

The file lock package provides a set of routines that allow the caller to handle byte range file lock requests. A variable of type FILE_LOCK is needed for every file with byte range locking. The package provides routines to set and clear locks, and to test for read or write access to a file with byte range locks.

The main idea of the package is to have the file system initialize a FILE_LOCK variable for every data file as its opened, and then to simply call a file lock processing routine to handle all IRP's with a major function code of LOCK_CONTROL. The package is responsible for keeping track of locks and for completing the LOCK_CONTROL IRPS. When processing a read or write request the file system can then call two query routines to check for access.

Most of the code for processing IRPS and checking for access use paged pool and can encounter a page fault, therefore the check routines cannot be called at DPC level. To help servers that do call the file system to do read/write operations at DPC level there is a additional routine that simply checks for the existence of a lock on a file and can be run at DPC level.

Concurrent access to the FILE_LOCK variable must be control by the caller.

The functions provided in this package are as follows:

o FsRtlInitializeFileLock - Initialize a new FILE_LOCK structure.

o FsRtlUninitializeFileLock - Uninitialize an existing FILE_LOCK structure.

o FsRtlProcessFileLock - Process an IRP whose major function code is LOCK_CONTROL.

o FsRtlCheckLockForReadAccess - Check for read access to a range of bytes in a file.

o FsRtlCheckLockForWriteAccess - Check for write access to a range of bytes in a file.

o FsRtlAreThereCurrentFileLocks - Check if there are any locks currently assigned to a file.

o FsRtlGetNextFileLock - This procedure enumerates the current locks of a file lock variable.

3.1 FsRtlInitializeFileLock

VOID
FsRtlInitializeFileLock(
    IN PFILE_LOCK *OpaqueFileLock*,
    IN POOL_TYPE *PoolType*,
    IN PCOMPLETE_LOCK_IRP_ROUTINE *CompleteLockIrpRoutine* **OPTIONAL**
)

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This routine initializes a new FILE_LOCK structure. The caller must supply the memory for the structure. This call must precede all other calls that utilize the FILE_LOCK variable.

Parameters:

OpaqueFileLock - Supplies a pointer to the FILE_LOCK structure to initialize.

PoolType - Supplies the pool type to use when allocating additional internal storage. If nonpaged pool is selected then all of the routines in this package can be called a DPC level. However, using nonpaged pool for storing file lock information is not a wise use of nonpaged pool.

CompleteLockIrpRoutine - Optionally supplies an alternate routine to call for completing IRPs. FsRtlProcessFileLock by default will call IoCompleteRequest to finish up an IRP; however if the caller want to process the completion itself then it needs to specify a completion routine here. This routine will then be called in place of IoCompleteRequest.

Return Value:

None.

3.2 FsRtlUninitializeFileLock

VOID
FsRtlUninitializeFileLock(
IN PFILE_LOCK *OpaqueFileLock*
)

Routine Description:

This routine uninitializes a FILE_LOCK structure. After calling this routine the File lock must be reinitialized before being used again.

This routine will free all files locks and completes any outstanding lock requests as a result of cleaning itself up.

Parameters:

OpaqueFileLock - Supplies a pointer to the FILE_LOCK struture being decommissioned.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.3 FsRtlAreThereCurrentFileLocks

BOOLEAN
FsRtlAreThereCurrentFileLocks(
    IN PFILE_LOCK *OpaqueFileLock*
)

Routine Description:

This routine tells its if caller there are any current file locks for the file. It does this test by simply checking the file lock variable and not by accessing other pieces of memory. Therefore if the FileLock variable is in nonpaged pool then this test can be done at DPC.

Parameters:

FileLock - Supplies the File lock being queried

Return Value:

BOOLEAN - TRUE if there are current locks on the file and FALSE otherwise

3.4 FsRtlProcessFileLock

NTSTATUS
FsRtlProcessFileLock(
    IN PFILE_LOCK *OpaqueFileLock***,**
    IN PIRP *Irp***,**
    IN PVOID *Context* **OPTIONAL**
)

Routine Description:

This routine processes a file lock IRP it does either a lock request, or an unlock request. It also completes the IRP. Once called the user (i.e., File System) has relinquished control of the input IRP.

If pool is not available to store the information this routine will raise a status value indicating insufficient resources.

Parameters:

OpaqueFileLock - Supplies the File lock being modified/queried.

Irp - Supplies the Irp being processed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Context - Optionally supplies a context to use when calling the user alternate IRP completion routine.

<u>Return Value:</u>

NTSTATUS - The return status for the operation.

3.5 FsRtlCheckLockForReadAccess

BOOLEAN
FsRtlCheckLockForReadAccess(
  IN PFILE_LOCK *OpaqueFileLock*,
  IN PIRP *Irp*
)

<u>Routine Description:</u>

This routine checks to see if the caller has read access to the range indicated in the IRP due to file locks. This call does not complete the Irp it only uses it to get the lock information and read information. The IRP must be for a read operation.

<u>Parameters:</u>

OpaqueFileLock - Supplies the File Lock to check.

Irp - Supplies the Irp being processed.

<u>Return Value:</u>

BOOLEAN - TRUE if the indicated user/request has read access to the entire specified byte range, and FALSE otherwise

3.6 FsRtlCheckLockForWriteAccess

BOOLEAN
FsRtlCheckLockForWriteAccess(
  IN PFILE_LOCK *OpaqueFileLock*,
  IN PIRP *Irp*
)

<u>Routine Description:</u>

This routine checks to see if the caller has write access to the indicated range due to file locks. This call does not complete the Irp it only uses it to get the lock information and write information. The IRP must be for a write operation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

OpaqueFileLock - Supplies the File Lock to check.

Irp - Supplies the Irp being processed.

Return Value:

BOOLEAN - TRUE if the indicated user/request has write access to the entire specified byte range, and FALSE otherwise

3.7 FsRtlGetNextFileLock

PFILE_LOCK_INFO
FsRtlGetNextFileLock(
IN PFILE_LOCK *OpaqueFileLock*,
IN BOOLEAN *Restart*
)

Routine Description:

This routine enumerate the file lock current denoted by the input file lock variable. It returns a pointer to the file lock information stored for each lock. The caller is responsible for synchronizing call to this procedure and for not altering any of the data returned by this procedure.

The way a programing will use this procedure to enumerate all of the locks is as follows:

```
for (p = FsRtlGetNextFileLock( FileLock, TRUE );

     p != NULL;

     p = FsRtlGetNextFileLock( FileLock, FALSE )) {

    // Process the lock information referenced by p

  }
```

Parameters:

OpaqueFileLock - Supplies the File Lock to enumerate. The current enumeration state is stored in the file lock variable so if multiple threads are enumerating the lock at the same time the results will be unpredictable.

Restart - Indicates if the enumeration is to start at the beginning of the file lock list or if we are continuing from a previous call.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

PFILE_LOCK_INFO - Either it returns a pointer to the next file lock record for the input file lock or it returns NULL if there are not more locks.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Name Support Routines

The name support package is for manipulating DBCS strings (later this will be extended to also handle UNICODE strings). The routines allow the caller to dissect and compare strings.

There are two exported typedef's defined by this package. The first is a structure called CODEPAGE. Every Dbcs routines takes as input a code page record. This record contains the double byte character and upcase information. If a code page not supplied the routines currently use the default US code page.

We need to work out the routines for a file system to contruct a code page.

The second typedef is an enumerated type called COMPARISON_RESULTS that is used when comparing two strings. It indicates if one string is less than, equal to, or greater than the other. The comparison routines also know how to handle wild cards.

The following routines are provided by this package:

o FsRtlDissectDbcs - This routine takes a path name string and breaks into two parts. The first name in the string and the remainder. It also checks that the first name is valid for an OS/2 file.

o FsRtlUpcaseDbcs - This routines takes a string and computes its upcased equivalent.

o FsRtlDbcsContainsWildCards - This routines tells the caller if a string contains any wildcard characters (i.e., * or ?).

o FsRtlCompareDbcs - This routine compares two strings.

o FsRtlIsDbcsInExpression - This routine is used to compare a string against a template (possibly containing wildcards) to sees if the string is in the language denoted by the template.

o FsRtlIsNameValid - This routine checks to see if a string contains valid characters.

o FsRtlIsPathValid - This routine checks to see if a string contains valid names separated by backslashes.

o FsRtlFirstDbcsCharacter - This routine is used to extract the first character from a DBCS string.

o FsRtlIslegalDbcsCharacter - This routine is used to decide if a DBCS character value is legal.

o FsRtlToUpperDbcsCharacter - This routine is used to upcase a single DBCS character.

4.1 FsRtlFirstDbcsCharacter

USHORT
FsRtlFirstDbcsCharacter(

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN PCODEPAGE *CodePage* **OPTIONAL,**
IN STRING *Name***,**
OUT PSTRING *RemainingName*
)

Routine Description:

This routine takes an input Dbcs string and returns as its function value the first character in the string and as an output parameter the remaining name after the first character. If the name is empty then a value of zero is returned. If the first character in the input name is invalid then an invalid character is returned and the remaining name is advanced by one byte through the input name.

Example of its results are:

```
Name      Function Result    RemainingName

  empty     0                  empty

  A         A                  empty

  ~A        ~                  A (~ denotes an illegal char)

  AB        A                  B
```

Note that given a Dbcs string denoted by a STRING variable Str the 1st, 2nd, and subsequent Dbcs characters can be extracted using the following programming construct

```
while (Str.Length != 0) {

     Dbcs = FsRtlFirstDbcsCharacter( NULL, Str, &Str );

      //  Dbcs now contains the next character in the string.

  }
```

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when examining the input string; otherwise a generic code page is used.

Name - Supplies the input string being examined

RemainingName - Receives the remaining part of the input string after the first character.

Return Value:

USHORT - Receives the first character found in the input string.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2 FsRtlDissectDbcs

BOOLEAN
FsRtlDissectDbcs(
    IN PCODEPAGE *CodePage* **OPTIONAL,**
    IN STRING *InputName***,**
    IN BOOLEAN *Is8dot3***,**
    OUT PSTRING *FirstPart***,**
    OUT PSTRING *RemainingPart*
)

Routine Description:

This routine takes an input Dbcs string and dissects it into two substrings. The first output string contains the name that appears at the beginning of the input string, the second output string contains the remainder of the input string.

In the input string backslashes are used to separate names. The input string must not start with a backslash. Both output strings will not begin with a backslash.

If the input string does not contain any names then both output strings are empty. If the input string contains only one name then the first output string contains the name and the second string is empty.

Note that both output strings use the same string buffer memory of the input string.

This routine returns a function result of TRUE if the input string is well formed (including empty) and contains only valid characters (including wildcards). It returns FALSE if the input string is illformed, contains invalid characters.

Example of its results are:

InputString	FirstPart	RemainingPart	Function Result
empty	empty	empty	TRUE
A	A	empty	TRUE
A\B\C\D\E	A	B\C\D\E	TRUE
*A?	*A?	empty	TRUE
\A	empty	empty	FALSE
A[,]	empty	empty	FALSE
A\\B+;\C	A	\B+;\C	TRUE

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when dissecting the input string; otherwise a generic code page is used.

InputName - Supplies the input string being dissected

*Is*8dot3 - Indicates if the first part of the input name must be 8.3 or can be long file name.

FirstPart - Receives the first name in the input string

RemainingPart - Receives the remaining part of the input string

Return Value:

BOOLEAN - TRUE if the input string is well formed and its first part does not contain any illegal characters, and FALSE otherwise.

4.3 FsRtlUpcaseDbcs

VOID
FsRtlUpcaseDbcs(
    IN PCODEPAGE *CodePage* **OPTIONAL,**
    IN STRING *InputName***,**
    OUT PSTRING *OutputName*
)

Routine Description:

This routine copies and upcases a Dbcs input string into a caller supplied output string according to the following upcase mapping rules.

For character values between 0 and 127, upcase normally.

For character values between 128 and 255 and not DBCS, use the upcase table in the code page to upcase a single character.

For character values between 128 and 255 and DBCS, do not alter.

The first two points above are handled transparently via the Code Page

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when upcasing the input string; otherwise a generic code page is used.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

InputName - Supplies the input string to upcase

OutputName - Receives the output string, the output buffer must already be supplied by the caller

Return Value:

None.

4.4 FsRtlDbcsContainsWildCards

BOOLEAN
FsRtlDbcsContainsWildCards(
IN PCODEPAGE *CodePage* **OPTIONAL,**
IN STRING *Name*
)

Routine Description:

This routine checks if the input Dbcs name contains any wild card characters (i.e., * or ?).

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when examining the input string; otherwise a generic code page is used.

Name - Supplies the name to examine

Return Value:

BOOLEAN - TRUE if the input name contains any wildcard characters and FALSE otherwise.

4.5 FsRtlCompareDbcs

COMPARISON_RESULTS
FsRtlCompareDbcs(
IN PCODEPAGE *CodePage* **OPTIONAL,**
IN STRING *Expression***,**
IN STRING *Name***,**
IN COMPARISON_RESULTS *WildIs***,**
IN BOOLEAN *CaseInsensitive*
)

Routine Description:

This routine compares a Dbcs expression with a Dbcs name lexigraphically for LessThan, EqualTo, or GreaterThan. If the expression does not contain any wildcards, this procedure does

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

a complete comparison. If the expression does contain wild cards, then the comparison is only done up to the first wildcard character. The Name parameter must not contain wild cards. The wildcard character compares as less then all other characters. So the wildcard name "*.*" will always compare less than all other strings.

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when examining the input string; otherwise a generic code page is used.

Expression - Supplies the first name (expression) to compare, optionally with wild cards. (Upcased already if CaseInsensitive is supplied as TRUE.)

Name - Supplies the second name to compare - no wild cards allowed.

WildIs - Determines what Result is returned if a wild card is encountered in the Expression String.

CaseInsensitive - TRUE if Name should be Upcased before comparing.

Return Value:

COMPARISON_RESULTS -

LessThan if Expression < Name

EqualTo if Expression == Name

GreaterThan if Expression > Name

4.6 FsRtlIsDbcsInExpression

BOOLEAN
FsRtlIsDbcsInExpression(
IN PCODEPAGE *CodePage* **OPTIONAL,**
IN STRING *Expression***,**
IN STRING *Name***,**
IN BOOLEAN *CaseInsensitive*
)

Routine Description:

This routine compares a Dbcs name and an expression and tells the caller if the name is equal to or not equal to the expression. The input name cannot contain wildcards, while the expression may contain wildcards.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when examining the input string; otherwise a generic code page is used.

Expression - Supplies the input expression to check against (Caller must already upcase if passing CaseInsensitive TRUE.)

Name - Supplies the input name to check for.

CaseInsensitive - TRUE if Name should be Upcased before comparing.

Return Value:

BOOLEAN - TRUE if Name is an element in the set of strings denoted by the input Expression and FALSE otherwise.

4.7 FsRtlIsNameValid

BOOLEAN
FsRtlIsNameValid(
  IN PCODEPAGE *CodePage* **OPTIONAL,**
  IN STRING *Name***,**
  IN BOOLEAN *Is8dot3***,**
  IN BOOLEAN *CanContainWildCards***,**
  OUT PSTRING *LongestValidPrefix* **OPTIONAL**
)

Routine Description:

This routine scans the input Dbcs name and verifies that if only contains valid characters.

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when examining the input string; otherwise a generic code page is used.

Name - Supplies the input name to check.

*Is*8dot3 - Specifies if the Name must be 8.3 or can be a long file name.

CanContainWildCards - Indicates if the name can contain wild cards (i.e., * and ?).

LongestValidPrefix - This optional output parameter receives a string denoting the largest valid prefix found for the input string. If the input string is completely valid then the longest valid prefix is equal to the input string.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

BOOLEAN - TRUE if the input name is valid and FALSE otherwise.

4.8 FsRtlIsPathValid

BOOLEAN
FsRtlIsPathValid(
IN PCODEPAGE *CodePage* **OPTIONAL,**
IN STRING *Name***,**
IN BOOLEAN *Is8dot3***,**
IN BOOLEAN *MustHaveLeadingBackslash***,**
IN BOOLEAN *CanContainWildCards***,**
OUT PSTRING *LongestValidPrefix* **OPTIONAL**
)

Routine Description:

This routine scans the input Dbcs string and verifies that it is only composed of valid names separated by backslashes.

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when examining the input string; otherwise a generic code page is used.

Name - Supplies the input name to check.

*Is*8dot3 - Specifies if the Name must be 8.3 or can be a long file name.

MustHaveLeadingBackslash - Specifies if the name must start with a leading backslash.

CanContainWildCards - Indicates if the name can contain wild cards (i.e., * and ?).

LongestValidPrefix - This optional output parameter receives a string denoting the largest valid prefix found for the input string. If the input string is completely valid then the longest valid prefix is equal to the input string.

Return Value:

BOOLEAN - TRUE if the input name is valid and FALSE otherwise.

4.9 FsRtlIsLegalDbcsCharacter

BOOLEAN
FsRtlIsLegalDbcsCharacter(

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN PCODEPAGE *CodePage* **OPTIONAL,**
IN USHORT *DbcsCharacter*
)

Routine Description:

This routine takes an input character (either double byte or single byte) and indicates to the caller if the character is legal. The input to this procedure should be the return value of having called FsRtlFirstDbcsCharacter.

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when examining the input character; otherwise a generic code page is used.

DbcsCharacter - Supplies the input character being examined

Return Value:

BOOLEAN - TRUE if the input character is legal and FALSE otherwise

4.10 FsRtlToUpperDbcsCharacter

USHORT
FsRtlToUpperDbcsCharacter(
IN PCODEPAGE *CodePage* **OPTIONAL,**
IN USHORT *DbcsCharacter*
)

Routine Description:

This routine takes an input character (either double byte or single byte) and returns its upcased equivalent character. The input to this procedure should be the return value of having called FsRtlFirstDbcsCharacter.

Parameters:

CodePage - Is an optional input parameter. If supplied it specifies the code page to use when examining the input character; otherwise a generic code page is used.

DbcsCharacter - Supplies the input character being upcased

Return Value:

USHORT - Recieves the upcased character

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Mapped Control Block Routines

The MCB routines provide support for maintaining an in-memory copy of the retrieval mapping information for a file. The general idea is to have the file system lookup the retrieval mapping for a VBN once from the disk, add the mapping to the MCB structure, and then utilize the MCB to retrieve the mapping for subsequent accesses to the file. A variable of type MCB is used to store the mapping information.

The routines provided here allow the user to incrementally store some or all of the retrieval mapping for a file and to do so in any order. That is, the mapping can be inserted to the MCB structure all at once starting from the beginning and working to the end of the file, or it can be randomly scattered throughout the file.

The package identifies each contiguous run of sectors mapping VBNs and LBNs indenpendent of the order they are added to the MCB structure. For example a user can define a mapping between VBN sector 0 and LBN sector 107, and between VBN sector 2 and LBN sector 109. The mapping now contains two runs each one sector in length. Now if the user adds an additional mapping between VBN sector 1 and LBN sector 106 the MCB structure will contain only one run 3 sectors in length.

Concurrent access to the MCB structure is control by this package.

The following routines are provided by this package:

o FsRtlInitializeMcb - Initialize a new MCB structure. There should be one MCB for every opened file. Each MCB structure must be initialized before it can be used by the system.

o FsRtlUninitializeMcb - Uninitialize an MCB structure. This call is used to cleanup any anciallary structures allocated and maintained by the MCB. After being uninitialized the MCB must again be initialized before it can be used by the system.

o FsRtlAddMcbEntry - This routine adds a new range of mappings between LBNs and VBNs to the MCB structure.

o FsRtlRemoveMcbEntry - This routines removes an existing range of mappings between LBNs and VBNs from the MCB structure.

o FsRtlLookupMcbEntry - This routine returns the LBN mapped to by a VBN, and indicates, in sectors, the length of the run.

o FsRtlLookupLastMcbEntry - This routine returns the mapping for the largest VBN stored in the structure.

o FsRtlNumberOfRunsInMcb - This routine tells the caller total number of discontiguous sectors runs stored in the MCB structure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o FsRtlGetNextMcbEntry - This routine returns the the caller the starting VBN and LBN of a given run stored in the MCB structure.

5.1 FsRtlInitializeMcb

VOID
FsRtlInitializeMcb(
  IN PMCB *OpaqueMcb*,
  IN POOL_TYPE *PoolType*
)

Routine Description:

This routine initializes a new Mcb structure. The caller must supply the memory for the Mcb structure. This call must precede all other calls that set/query the Mcb structure.

If pool is not available this routine will raise a status value indicating insufficient resources.

Parameters:

OpaqueMcb - Supplies a pointer to the Mcb structure to initialize.

PoolType - Supplies the pool type to use when allocating additional internal Mcb memory.

Return Value:

None.

5.2 FsRtlUninitializeMcb

VOID
FsRtlUninitializeMcb(
  IN PMCB *OpaqueMcb*
)

Routine Description:

This routine uninitializes an Mcb structure. After calling this routine the input Mcb structure must be re-initialized before being used again.

Parameters:

OpaqueMcb - Supplies a pointer to the Mcb structure to uninitialize.

Return Value:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

None.

5.3 FsRtlAddMcbEntry

BOOLEAN
FsRtlAddMcbEntry(
    IN PMCB *OpaqueMcb*,
    IN VBN *Vbn*,
    IN LBN *Lbn*,
    IN ULONG *SectorCount*
)

Routine Description:

This routine is used to add a new mapping of VBNs to LBNs to an existing Mcb. The information added will map

Vbn to Lbn,

Vbn+1 to Lbn+1,...

Vbn+(SectorCount-1) to Lbn+(SectorCount-1).

The mapping for the VBNs must not already exist in the Mcb. If the mapping continues a previous run, then this routine will actually coalesce them into 1 run.

If pool is not available to store the information this routine will raise a status value indicating insufficient resources.

An input Lbn value of zero is illegal (i.e., the Mcb structure will never map a Vbn to a zero Lbn value).

Parameters:

OpaqueMcb - Supplies the Mcb in which to add the new mapping.

Vbn - Supplies the starting Vbn of the new mapping run to add to the Mcb.

Lbn - Supplies the starting Lbn of the new mapping run to add to the Mcb.

SectorCount - Supplies the size of the new mapping run (in sectors).

Return Value:

BOOLEAN - TRUE if the mapping was added successfully (i.e., the new Vbns did not collide with existing Vbns), and FALSE otherwise. If FALSE is returned then the Mcb is not changed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.4 FsRtlRemoveMcbEntry

VOID
FsRtlRemoveMcbEntry(
    IN PMCB *OpaqueMcb***,**
    IN VBN *Vbn***,**
    IN ULONG *SectorCount*
)

Routine Description:

This routine removes a mapping of VBNs to LBNs from an Mcb. The mappings removed are for

Vbn,

Vbn+1, to

Vbn+(SectorCount-1).

The operation works even if the mapping for a Vbn in the specified range does not already exist in the Mcb. If the specified range of Vbn includes the last mapped Vbn in the Mcb then the Mcb mapping shrinks accordingly.

If pool is not available to store the information this routine will raise a status value indicating insufficient resources.

Parameters:

OpaqueMcb - Supplies the Mcb from which to remove the mapping.

Vbn - Supplies the starting Vbn of the mappings to remove.

SectorCount - Supplies the size of the mappings to remove (in sectors).

Return Value:

None.

5.5 FsRtlLookupMcbEntry

BOOLEAN
FsRtlLookupMcbEntry(
    IN PMCB *OpaqueMcb***,**
    IN VBN *Vbn***,**
    OUT PLBN *Lbn***,**
    OUT PULONG *SectorCount* **OPTIONAL**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
    )
```

Routine Description:

This routine retrieves the mapping of a Vbn to an Lbn from an Mcb. It indicates if the mapping exists and the size of the run.

Parameters:

OpaqueMcb - Supplies the Mcb being examined.

Vbn - Supplies the Vbn to lookup.

Lbn - Receives the Lbn corresponding to the Vbn. A value of zero is returned if the Vbn does not have a corresponding Lbn.

SectorCount - Receives the number of sectors that map from the Vbn to contiguous Lbn values beginning with the input Vbn.

Return Value:

BOOLEAN - TRUE if the Vbn is within the range of VBNs mapped by the MCB (even if it corresponds to a hole in the mapping), and FALSE if the Vbn is beyond the range of the MCB's mapping.

For example, if an MCB has a mapping for VBNs 5 and 7 but not for 6, then a lookup on Vbn 5 or 7 will yield a non zero Lbn and a sector count of 1. A lookup for Vbn 6 will return TRUE with an Lbn value of 0, and lookup for Vbn 8 or above will return FALSE.

5.6 FsRtlLookupLastMcbEntry

BOOLEAN
FsRtlLookupLastMcbEntry(
    IN PMCB *OpaqueMcb***,**
    OUT PVBN *Vbn***,**
    OUT PLBN *Lbn*
)

Routine Description:

This routine retrieves the last Vbn to Lbn mapping stored in the Mcb. It returns the mapping for the last sector or the last run in the Mcb. The results of this function is useful when extending an existing file and needing to a hint on where to try and allocate sectors on the disk.

Parameters:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

OpaqueMcb - Supplies the Mcb being examined.

Vbn - Receives the last Vbn value mapped.

Lbn - Receives the Lbn corresponding to the Vbn.

Return Value:

BOOLEAN - TRUE if there is a mapping within the Mcb and FALSE otherwise (i.e., the Mcb does not contain any mapping).

5.7 FsRtlNumberOfRunsInMcb

ULONG
FsRtlNumberOfRunsInMcb(
IN PMCB *OpaqueMcb*
)

Routine Description:

This routine returns to the its caller the number of distinct runs mapped by an Mcb. Holes (i.e., Vbns that map to Lbn=0) are counted as runs. For example, an Mcb containing a mapping for only Vbns 0 and 3 will have 3 runs, one for the first mapped sector, a second for the hole covering Vbns 1 and 2, and a third for Vbn 3.

Parameters:

OpaqueMcb - Supplies the Mcb being examined.

Return Value:

ULONG - Returns the number of distinct runs mapped by the input Mcb.

5.8 FsRtlGetNextMcbEntry

BOOLEAN
FsRtlGetNextMcbEntry(
IN PMCB *OpaqueMcb***,**
IN ULONG *RunIndex***,**
OUT PVBN *Vbn***,**
OUT PLBN *Lbn***,**
OUT PULONG *SectorCount*
)

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This routine returns to its caller the Vbn, Lbn, and SectorCount for distinct runs mapped by an Mcb. Holes are counted as runs. For example, to construct to print out all of the runs in a a file is:

```
for (i = 0; FsRtlGetNextMcbEntry(Mcb,i,&Vbn,&Lbn,&Count); i++){

     // print out vbn, lbn, and count

     }
```

Parameters:

OpaqueMcb - Supplies the Mcb being examined.

RunIndex - Supplies the index of the run (zero based) to return to the caller.

Vbn - Receives the starting Vbn of the returned run, or zero if the run does not exist.

Lbn - Recieves the starting Lbn of the returned run, or zero if the run does not exist.

SectorCount - Receives the number of sectors within the returned run, or zero if the run does not exist.

Return Value:

BOOLEAN - TRUE if the specified run (i.e., RunIndex) exists in the Mcb, and FALSE otherwise. If FALSE is returned then the Vbn, Lbn, and SectorCount parameters receive zero.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6. Volume Mapped Control Block Routines

The VMCB routines provide support for maintaining a mapping between LBNs and VBNs for a virtual volume file. The volume file is all of the sectors that make up the on-disk structures. A file system uses this package to map LBNs for on-disk structure to VBNs in a volume file. This when used in conjunction with Memory Management and the Cache Manager will treat the volume file as a simple mapped file. A variable of type VMCB is used to store the mapping information and one is needed for every mounted volume.

The main idea behind this package is to allow the user to dynamically read in new disk structure sectors (e.g., FNODEs). The user assigns the new sector a VBN in the Volume file and has memory management fault the page containing the sector into memory. To do this Memory management will call back into the file system to read the page from the volume file passing in the appropriate VBN. Now the file system takes the VBN and maps it back to its LBN and does the read.

The granularity of mapping is one a per page basis. That is if a mapping for LBN 8 is added to the VMCB structure and the page size is 8 sectors then the VMCB routines will actually assign a mapping for LBNS 8 through 15, and they will be assigned to a page aligned set of VBNS. This function is needed to allow us to work efficiently with memory management. This means that some sectors in some pages might actually contain regular file data and not volume information, and so when writing the page out we must only write the sectors that are really in use by the volume file. To help with this we provide a set of routines to keep track of dirty volume file sectors. That way, when the file system is called to write a page to the volume file, it will only write the sectors that are dirty.

Concurrent access the VMCB structure is control by this package.

The functions provided in this package are as follows:

o FsRtlInitializeVmcb - Initialize a new VMCB structure.

o FsRtlUninitializeVmcb - Uninitialize an existing VMCB structure.

o FsRtlSetMaximumLbnVmcb - Sets/Resets the maximum allowed LBN for the specified VMCB structure.

o FsRtlAddVmcbMapping - This routine takes an LBN and assigns to it a VBN. If the LBN already was assigned to an VBN it simply returns the old VBN and does not do a new assignemnt.

o FsRtlRemoveVmcbMapping - This routine takes an LBN and removes its mapping from the VMCB structure.

o FsRtlVmcbVbnToLbn - This routine takes a VBN and returns the LBN it maps to.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o FsRtlVmcbLbnToVbn - This routine takes an LBN and returns the VBN its maps to.

o FsRtlSetDirtyVmcb - This routine is used to mark sectors dirty in the volume file.

o FsRtlSetCleanVmcb - This routine is used to mark sectors clean in the volume file.

o FsRtlGetDirtySectorsVmcb - This routine is used to retrieve the dirty sectors for a page in the volume file.

6.1 FsRtlInitializeVmcb

VOID
FsRtlInitializeVmcb(
    IN PVMCB *OpaqueVmcb***,**
    IN POOL_TYPE *PoolType***,**
    IN ULONG *MaximumLbn*
)

Routine Description:

This routine initializes a new Vmcb Structure. The caller must supply the memory for the structure. This must precede all other calls that set/query the volume file mapping.

If pool is not available this routine will raise a status value indicating insufficient resources.

Parameters:

OpaqueVmcb - Supplies a pointer to the volume file structure to initialize.

PoolType - Supplies the pool type to use when allocating additional internal structures.

MaximumLbn - Supplies the maximum Lbn value that is valid for this volume.

Return Value:

None

6.2 FsRtlUninitializeVmcb

VOID
FsRtlUninitializeVmcb(
    IN PVMCB *OpaqueVmcb*
)

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This routine uninitializes an existing VMCB structure. After calling this routine the input VMCB structure must be re-initialized before being used again.

Parameters:

OpaqueVmcb - Supplies a pointer to the VMCB structure to uninitialize.

Return Value:

None.

6.3 FsRtlSetMaximumLbnVmcb

VOID
FsRtlSetMaximumLbnVmcb(
    IN PVMCB *OpaqueVmcb*,
    IN ULONG *MaximumLbn*
)

Routine Description:

This routine sets/resets the maximum allowed LBN for the specified Vmcb structure. The Vmcb structure must already have been initialized by calling FsRtlInitializeVmcb.

Parameters:

OpaqueVmcb - Supplies a pointer to the volume file structure to initialize.

MaximumLbn - Supplies the maximum Lbn value that is valid for this volume.

Return Value:

None

6.4 FsRtlAddVmcbMapping

BOOLEAN
FsRtlAddVmcbMapping(
    IN PVMCB *OpaqueVmcb*,
    IN LBN *Lbn*,
    IN ULONG *SectorCount*,
    OUT PVBN *Vbn*
)

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This routine adds a new LBN to VBN mapping to the VMCB structure. When a new LBN is added to the structure it does it only on page aligned boundaries.

If pool is not available to store the information this routine will raise a status value indicating insufficient resources.

Parameters:

OpaqueVmcb - Supplies the VMCB being updated.

Lbn - Supplies the starting LBN to add to VMCB.

SectorCount - Supplies the number of Sectors in the run

Vbn - Receives the assigned VBN

Return Value:

BOOLEAN - TRUE if this is a new mapping and FALSE if the mapping for the LBN already exists. If it already exists then the sector count for this new addition must already be in the VMCB structure

6.5 FsRtlRemoveVmcbMapping

VOID
FsRtlRemoveVmcbMapping(
    IN PVMCB *OpaqueVmcb*,
    IN VBN *Vbn*,
    IN ULONG *SectorCount*
)

Routine Description:

This routine removes a Vmcb mapping.

If pool is not available to store the information this routine will raise a status value indicating insufficient resources.

Parameters:

OpaqueVmcb - Supplies the Vmcb being updated.

Vbn - Supplies the VBN to remove

SectorCount - Supplies the number of sectors to remove.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

None.

6.6 FsRtlVmcbVbnToLbn

```
BOOLEAN
FsRtlVmcbVbnToLbn(
    IN PVMCB OpaqueVmcb,
    IN VBN Vbn,
    IN PLBN Lbn,
    OUT PULONG SectorCount OPTIONAL
    )
```

Routine Description:

This routine translates a VBN to an LBN.

Parameters:

OpaqueVmcb - Supplies the VMCB structure being queried.

Vbn - Supplies the VBN to translate from.

Lbn - Receives the LBN mapped by the input Vbn. This value is only valid if the function result is TRUE.

SectorCount - Optionally receives the number of sectors corresponding to the run.

Return Value:

BOOLEAN - TRUE if he Vbn has a valid mapping and FALSE otherwise.

6.7 FsRtlVmcbLbnToVbn

```
BOOLEAN
FsRtlVmcbLbnToVbn(
    IN PVMCB OpaqueVmcb,
    IN LBN Lbn,
    OUT PVBN Vbn,
    OUT PULONG SectorCount OPTIONAL
    )
```

Routine Description:

This routine translates an LBN to a VBN.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameters:

OpaqueVmcb - Supplies the VMCB structure being queried.

Lbn - Supplies the LBN to translate from.

Vbn - Recieves the VBN mapped by the input LBN. This value is only valid if the function result is TRUE.

SectorCount - Optionally receives the number of sectors corresponding to the run.

Return Value:

BOOLEAN - TRUE if the mapping is valid and FALSE otherwise.

6.8 FsRtlSetDirtyVmcb

VOID
FsRtlSetDirtyVmcb(
  IN PVMCB *OpaqueVmcb***,**
  IN ULONG *LbnPageNumber***,**
  IN ULONG *Mask*
)

Routine Description:

This routine sets the sectors within a page as dirty based on the input mask.

If pool is not available to store the information this routine will raise a status value indicating insufficient resources.

Parameters:

OpaqueVmcb - Supplies the Vmcb being manipulated.

LbnPageNumber - Supplies the Page Number (LBN based) of the page being modified. For example, with a page size of 8 a page number of 0 corresponds to LBN values 0 through 7, a page number of 1 corresponds to 8 through 15, and so on.

Mask - Supplies the mask of dirty sectors to set for the Page (a 1 bit means to set it dirty). For example to set LBN 9 dirty on a system with a page size of 8 the LbnPageNumber will be 1, and the mask will be 0x00000002.

Return Value:

None.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.9 FsRtlSetCleanVmcb

VOID
FsRtlSetCleanVmcb(
    IN PVMCB *OpaqueVmcb***,**
    IN ULONG *LbnPageNumber*
)

Routine Description:

This routine sets all of the sectors within a page as clean. All of the sectors in a page whether they are dirty or not are set clean by this procedure.

Parameters:

OpaqueVmcb - Supplies the Vmcb being manipulated.

LbnPageNumber - Supplies the Page Number (Lbn based) of page being modified. For example, with a page size of 8 a page number of 0 corresponds to LBN values 0 through 7, a page number of 1 corresponds to 8 through 15, and so on.

Return Value:

None.

6.10 FsRtlGetDirtySectorsVmcb

ULONG
FsRtlGetDirtySectorsVmcb(
    IN PVMCB *OpaqueVmcb***,**
    IN ULONG *LbnPageNumber*
)

Routine Description:

This routine returns to its caller a mask of dirty sectors within a page.

Parameters:

OpaqueVmcb - Supplies the Vmcb being manipulated

LbnPageNumber - Supplies the Page Number (Lbn based) of page being modified. For example, with a page size of 8 a page number of 0 corresponds to LBN values 0 through 7, a page number of 1 corresponds to 8 through 15, and so on.

Return Value:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG - Receives a mask of dirty sectors within the specified page. (a 1 bit indicates that the sector is dirty).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Product Description and Implementation Plan

Author: *David N. Cutler*

Revision 0.1, October 24, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Executive Summary 1

2. Overall Goals 2

3. Major Milestones, Implementation Strategy, and Overall Schedule 3

4. Self Hosted Development Environment 4
- 4.1. ANSI Terminal Based 386/486 Self Hosted System 4
- 4.2. ANSI Terminal Based MIPS Self Hosted System 6
- 4.3. Windows Based Self Hosted 386/486 and MIPS System 7

5. Beta Test SDK 8

6. Product Descriptions 8
- 6.1. Power PC Workstation Release 8
  - 6.1.1 Deliverables 8
  - 6.1.2 Base System 8
  - 6.1.3 Windows 8
  - 6.1.4 Network 8
  - 6.1.5 Schedule 8
  - 6.1.6 Dependencies 9
  - 6.1.7 Issues 9
- 6.2. Multiprocessor Server Release 9
  - 6.2.1 Deliverables 9
  - 6.2.2 Base System 9
  - 6.2.3 Windows 9
  - 6.2.4 Network 9
  - 6.2.5 Schedule 9
  - 6.2.6 Dependencies 9
  - 6.2.7 Issues 9
- 6.3. Full Workstation Release 9
  - 6.3.1 Deliverables 9
  - 6.3.2 Base System 9
  - 6.3.3 Windows 9
  - 6.3.4 Network 9
  - 6.3.5 Schedule 9
  - 6.3.6 Dependencies 10
  - 6.3.7 Issues 10

7. Product/Major Milestone Descriptions and Schedules 10

8. Project Goals 10

9. Dependencies 10
- 9.1. Languages Group 10

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

9.2. LanMan Group 11
9.3. Testing Group 11
9.4. User Ed Group 11

10. Hardware Plans 12

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Executive Summary

NT OS/2 , here referred to simply as **NT**, is a new operating system product being developed by **Microsoft** which is portable and supports the Windows 32-bit base system APIs, graphical user interface, and window management software.

This document describes implementation plans for the **NT** operating system and contains product descriptions, projected release dates, an overall schedule, a summary of the work items to be performed, and a list of external dependencies.

Development on **NT** actually began approximately two years ago and has progressed to the point where significant system functionality is operational on both 386/486 and MIPS RISC platforms.

In addition to the development of the **NT** operating system, **Microsoft** is also developing a reference implementation for RISC PCs based on the MIPS R4000 microprocessor chip. This hardware architecture will be the main target for the first **NT** product release.

The development and product releases of **NT** will be phased such that new markets are addressed first, followed by high end server markets, and finally the general workstation market.

NT is aimed at the high end of **Microsoft's** systems business and, when running on an 386/486 platform, will share a binary compatible 32-bit programming interface with the low end implementation of the Windows 32-bit operating system environment based on DOS and an extension of Windows 3.0.

On RISC systems, **NT** will provide source level compatiblity with the 386/486 versions of the Windows 32-bit operating system environment, and binary compatibility with other RISC systems of the same architecture.

Typically **NT** will service markets requiring larger memories and higher performance (e.g., greater than 4mb and RISC performance levels), whereas the low end system will service markets requiring smaller memories, lower performance, and x86 binary compatibility (.e.g, less than 4mb of memory and up to 486 performance).

Currently four major product releases are planned, although it is likely that one of more of these releases will be combined.

The first release of **NT** is planned as a workstation product that will provide a strong competitor to UN*X based workstations. It will provide the Windows 32-bit operating system environment, a POSIX compliant execution environment, high integrity, robustnesss, security, and be network enabled as both a client and a server. The primary target for this release is a MIPS based RISC PC, although a 386/486 system will also be developed in parallel and be ready for deployment.

> *\A major issue that needs to get resolved is whether DOS and/or Windows 16-bit emulation needs to be provided on the RISC platform. Another issue relates to whether the 1003.2 tools need to be delivered with the POSIX environment./*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The second release of **NT** is planned as a scalable performance server product and adds multiprocessor support for 486 systems, LanMan 3.0 functionality, an extensive set of network device drivers, and the full services needed to replace OS/2 1.x as the primary **Microsoft** server platform. Its main marketing goal is to provide strong competitor with Novell for server based systems.

The third release of **NT** adds full support for 386/486 workstations and includes DOS emulation, Windows 16-bit emulation, OS/2 32-bit Base APIs, and certified C2 security. It will provide a full PC workstation environment.

The fourth release of **NT** adds support for multiprocessor RISC servers. This release will most lilely be combined with the second release if hardware is available for testing and evaluation.

In addition to the planned product releases, an OAK, DDK, SDK, and source porting kit will be available at the appropriate times.

2. Overall Goals

The overall long term goals for the **NT** project are to:

- o Provide **Microsoft** with a high end Windows 32-bit operating system that is portable, secure, and provides the base technology to compete with UN*X on the desktop, Novell in the network, and provides the advanced features necessary to implement "information at your finger tips".
- o Provide **Microsoft** with a reference implementation of a RISC platform based on the MIPS R4000 microporcessor chip that can be used to facilitate the establishment of standards for the implementation of RISC PCs and servers.
- o Deliver on the above two goals by providing a series of product releases that build functionality, let **Microsoft** address new markets, and provide strong compatibility ties to existing and future low end products.

The specific development goals for **NT** are:

- o Portability - **NT** will be written in C and will be portable to RISC, the 386/486, and other architectures. A typical port to a new architecture should take no longer than six calender months.
- o Security - **NT** will be designed to have pervasive security and will be capable of attaining the "B" levels of security as defined by the U.S. government. Initially it will be certified at the C2 level.
- o Compatibility - **NT** will provide a high degree of compatiblity with other **Microsoft** systems.
  - Window 32-bit Environment - Binary compatibility with the low end implementation of 32-bit Windows environment will be provided when running on a 386/486 system. On

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

RISC platforms, source level compatibility with 386/486 systems will be provided and binary compatibiity with other RISC platforms of the same architecture.

- OS/2 32-bit Base APIs - Binary compatibility will be provided with the OS/2 2.0 32-bit Base APIs when running on a 386/486 platform. On RISC platforms source level compatiblity will be provided.

  OS/2 32-bit Base API binary compatibility is predicated on IBM accepting and implementing all of the NT OS/2 DCRs that were implemented in Cruiser. This includes the image format, structured exception handling, alignment of arguments, and changes to the semantics of muxwait.\

- DOS and Windows 16-bit Environment - Binary compatibility with DOS and 16-bit Windows will be provided when running on a 386/486 system. On RISC platforms, these capabilities will be provided via software emulation of the 8086 instruction set.

  It is not clear how extensive these capabilites will be. The simplest and most straight forward approach is to only run "clean" APPs that do not make arcane use of hardware resources. If all APPS have to be executed without change, then this goal becomes more difficult to achieve. Another question is whether network services need to be available to 16-bit environments.\

- File Systems - Binary compatible on-disk structures will be provided for the FAT, HPFS, and CD-ROM file systems on both 386/486 and RISC platforms.

- Network - LanMan compatible protocols, redirector, server, and network services will be provided.

o Multiprocessors - **NT** will support symmetric multiprocessing and provide scalable performance on 486 and RISC based platforms.

o POSIX - **NT** will provide a POSIX compliant IEEE 1003.1 (FIPS 151-1) POSIX execution environment for deployment in the government marketplace.

o Virtual Memory - **NT** will provide support for a 32-bit flat addressed virtual environment with demand paging, mapped files, and asynchronous I/O.

The 486 and RISC PC platforms will be fully supported by **NT** and will provide a robust and high integrity system. The deficiencies in the 386 memory management architecture, however, will not be fully masked and will result in a 386 based system that is less secure and does not exhibit the same level of integrity and robustness as the 486 and RISC systems. In actual practice this should not be a concern and only represents an exposure in a system under malicious attack.

386 platforms must contain an i386 B6 stepping or above to be supported. Earlier steppings will not be supported and an attempt to boot on such a platform will be rejected by the NT system with an appropriate error message.\

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. Major Milestones, Implementation Strategy, and Overall Schedule

Several major milestones are planned on the road to a the first release of an **NT** product. These milestones lead through a progression of functionality and will increase confidence that the implementation is proceeding according to plan.

Currently **NT** boots and executes user programs on both 386/486 and MIPS R3000 based DECstation 5000s. However, the network software is not complete, the complete set of development tools are not in place, the implementation of the Windows 32-bit base system APIs, graphical user interface, and window management environment are just beginning, and the system is not capable of supporting its own development. In addition, the Jazz hardware is not yet available for software development.

The first major milestone is the finalization of the implemenation plan, product descriptions, and development schedule. This is expected to occur before the end of the year with the first draft completed by **November 30, 1990**.

The main implementation strategy for **NT** is to provide a self hosted development environment on **NT** as quickly as possible. This will provide more testing, force the focus to a stable system that supports its own development, and provide a viable system for ISV and hardware OEM development.

Self hosting will first occur on the 386/486 and be followed shortly thereafter on the Jazz MIPS hardware. The initial self hosted development environment will support network connections to the source code server, character mode development tools, and will require an additional machine for mail and producing word documents.

The target date for self hosting on 386/486 systems is **March 26, 1991** and the target date for the Jazz MIPS system is **April 25, 1991**.

Self hosting with character mode tools will be followed by a windows environment that supports an ANSI terminal window. This will allow the windowing and graphical user interface software to be combined into the system that is running on each developer's desktop.

The target date for a self hosted system on both the 386/486 and Jazz MIPS platforms using the windowing environment is **????**.

The next major milestone is a Beta Test SDK that contains a full Windows 32-bit environment on both 386/486 and Jazz platforms. The target date for the Beta Test SDK is **????**.

It is envisioned that two major updates to the beta-test SDK will be required before the first real product release. These updates will occur at approximately 3 month intervals.

The target for the first product release is **????**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Self Hosted Development Environment

The self hosted development system provides for the development of **NT** on **NT**. There are three self hosted system milestones described below.

4.1. ANSI Terminal Based 386/486 Self Hosted System

The 386/486 self hosted system will occur first and will contain following tools and components:

1. A completely functional **NT** base system with virtual memory, multithreading, process management, image/DLL loader, file system support (FAT and HPFS), and disk driver (ST506).
2. Support for the Windows 32-bit base system API minus the named pipe, sound, and registration APIs.
3. ANSI terminal support for character mode APPs in the keyboard, mouse, and display drivers.
4. A complete C runtime library that uses the Windows 32-bit base system APIs.
5. A command interpreter (CMD.EXE) and the Z-Tools.
6. The source language maintenance utility (SLM).
7. A full screen editor (MEP).
8. The make utility (NMAKE).
9. A native profile utility.
10. A linker that produces executable images and DLLs.
11. An object module conversion utility to convert from the **Microsoft** x86 object format to the COFF format(CVTOMF).
12. The **NT** system build utility (BUILD.EXE).
13. A LanMan redirector that is capable of communicating with and accessing files and printing on a LanMan 2.0 server.
14. A NetBeui transport.
15. The CFRONT C++ preprocessor.
16. An Etherlink II NDIS driver.
17. A 386/486 C compiler with structured exception handling.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

18. A 386/486 assembler.

19. A 386/486 user mode debugger.

20. An OS/2 hosted 386/486 kernel debugger.

The **NT** group will deliver all of these components except the last four items which will be delivered by the **Microsoft** Languages group.

> *\A schedule commmitment is required from the languages group for support of an NT OS/2 hosted 386/486 C compiler, 386/486 assembler, and 386/486 user mode debugger.*

Four people from the **NT** group will be responsible for pulling together the actual system and verifying its operation over a 6-8 week period. These people are tentatively identified as Bryanwi, Stever, Garyki, and Davidtr. Kylesh from the testing group will be the offical build resource and will be responsible for maintaining the build and maintenance trees.

People developing the MIPS self hosted system will not be able to switch their development environment to the 386/486 **NT** system since they will have to be able to continue to compile on the DECstation 5000 systems which are accessed using TCP/IP.

The target data for the self hosted 386/486 system is **March 26, 1991**.

> *\A complete set of Windows 32-bit base system API tests should be operatioinal to check out this system. What other tests should be available?*

> *\A complete set of network aware file tests should be available.*

> *\File system and file server stress tests should be available.*

> *\Are there any doucmentation requirements?*

Documentation will be required for installation and a description of the features that are, and are not, available in the various utilities.

4.2. ANSI Terminal Based MIPS Self Hosted System

The MIPS self hosted system implementation will proceed in parallel with the 386/486 self hosted system support but will not occur until after the 386/486 version. It will contain the following additional tools and components:

1. A port of the **NT** base system from the DECstation 5000 to the R3000 based Jazz system. This requires a new set of device drivers for the SCSI and floopy disks and an update to the original i860 bootstrap code.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2. A port of the **NT** base system from the R3000 based Jazz system to the R4000 base Jazz system. This requires a rewrite of the trap handling code, an update to the memory management code, and an update to the interlocked operations.

3. ANSI terminal support for character mode APPs in the keyboard, mouse, and display drivers.

4. A port and verification of all the above development utilities and tools to the MIPS environment. This includes the 386/486 C compiler and 386/486 assembler.

5. The MIPS C compiler with structured exception handling.

6. The MIPS assembler.

7. A MIPS user mode debugger.

8. An OS/2 hosted MIPS kernel debugger.

9. A Sonic chip NDIS driver.

10. A port and verification of the redirector and NetBeui transport.

The **NT** group will deliver all of these components except the 386/486 C compiler, 386/486 assembler, the MIPS user mode debugger, and the OS/2 hosted MIPS kernel debugger which are being delivered by the **Microsoft** Languages Group.

> *\A schedule commmitment is required from the languages group for support of an NT OS/2 hosted 386/486 C compiler, 386/486 assembler, and a MIPS user mode debugger.*

Four people from the **NT** group will be responsible for pulling together the actual system and verifying its operation over a 6-8 week period. These people are tentatively identified as Markl, Davegi, Tomm, and Larryo. Kylesh from the testing group will be the offical build resource and will be responsible for maintaining the build and maintenance trees.

The target data for the self hosted MIPS system is **April 25, 1991**.

Meeting the target date assumes that a MIPS compiler will be available on the DECstation 5000 by **December 1, 1990** that fully supports structured exception handling and a Microsoft C compatible packed pragma.

Meeting the target date also assumes that the R3000 based Jazz hardware will be available for use by the **NT** software group by **December 1, 1990** and that the R4000 based Jazz system will be available by **February 1, 1991**.

> *\We will have to decide how to split the source tree for multiple targets within one architecture. Currently this is done via conditional compilation, but the differences*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

between the R3000 and R4000 based Jazz system will be too great to use this methodology.

\A ported set of Windows 32-bit base system API tests should be available for testing this system. What other tests should be available?

\A complete set of network aware file tests should be available.

\File system and file server stress tests should be available.

4.3. Windows Based Self Hosted 386/486 and MIPS System

The windows based self hosted system adds Windows support for an ANSI terminal window and allows the phase over from the interim ANSI terminal capabilites to a fully windowed system. This system will be supported on both 386/486 and MIPS platforms and will form the development environment for the components and capabilities needed for the Beta Test SDK system.

The Windows based self hosted system will contain the following additional capabilities and components:

1. ANSI terminal support in a window.
2. The GDI subset required for window support.
3. The user window manager.
4. The program manager in the shell.
5. Kernel and DDI level device drivers for the Jazz and 386/486 display, keyboard, and mouse that have the interim ANSI terminal support removed.
6. The resource compiler.

\What other tools and capabilities are needed?

\What is the debugging environment for windows apps? Is it 3.1 compatible? Does it require a separate terminal?

\The 32-bit thunks kit would help the development of Windows 32-bit APPs before the full windowing environment is available.

The target date for the self hosted windows system is **May 1, 1991**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Beta Test SDK

The Beta Test SDK will be a formally packaged system that is distributed to a selected set of ISVs and hardware OEMs wishing to develop device drivers. It will be supported on a selected 486 platform and the Jazz MIPS platform.

It will contain preliminary installation and configuration management software.

Windows based version of user debugger.

DDK and device driver writers guide.

NDIS driver writers guide.

6. Product Descriptions

This following sections contain a detailed description of the various product releases and schedules.

6.1. Power PC Workstation Release

6.1.1 Deliverables

This section contains a description of the deliverables.

6.1.2 Base System

6.1.3 Windows

6.1.4 Network

6.1.5 Schedule

This section section contains the schedule for major milestones.

6.1.6 Dependencies

This section contains the dependencies on other groups.

User-Ed, Testing, Languages, Lan, Windows-32.

6.1.7 Issues

This section contains any issues that need to be called out.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.2. Multiprocessor Server Release

6.2.1 Deliverables

This section contains a description of the deliverables.

6.2.2 Base System

6.2.3 Windows

6.2.4 Network

6.2.5 Schedule

This section section contains the schedule for major milestones.

6.2.6 Dependencies

This section contains the dependencies on other groups.

User-Ed, Testing, Languages, Lan, Windows-32.

6.2.7 Issues

This section contains any issues that need to be called out.

6.3. Full Workstation Release

6.3.1 Deliverables

This section contains a description of the deliverables.

6.3.2 Base System

6.3.3 Windows

6.3.4 Network

6.3.5 Schedule

This section section contains the schedule for major milestones.

6.3.6 Dependencies

This section contains the dependencies on other groups.

User-Ed, Testing, Languages, Lan, Windows-32.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.3.7 Issues

This section contains any issues that need to be called out.

7. Product/Major Milestone Descriptions and Schedules

The development strategy that is being followed is to

The next major milestone in the development of **NT OS/2** will be the ability of the operating system to host its own development. This is planned to be operational on both the x86 and the MIPS RISC PC in Q1'91.

This milestone will be followed by a beta quality field test SDK that will be available on the MIPS RISC PC and Compaq 486 systems in Q3'91.

The first retail product will be a MIPS RISC PC that supports the Windows 32-bit APIs, is network enabled, provides a robust and secure operating environment, and is capable of competing with UN*X systems. A secondary goal for this product is the support for Compaq 486 systems. The target date for this product is Q1'92.

The second retail product is aimed at providing a robust and secure platform for scalable performance LanMan servers. This release will support multi-processor 486 systems (possibly also MIPS RISC multi-processor systems as well) and will support all the network services and NDIS drivers necessary to compete with Novell on 386 and 486 systems. The target date for this product is H2'92.

The third retail product is aimed at providing a full workstation capability for 368 and 486 systems that is certifiably secure, contains support for DOS and Windows 16-bit applications, and also supports the OS/2 32-bit base system APIs. The target date for this product is somtime in 93.

What products have an OAK? DDK?

8. Project Goals

9. Dependencies

The **NT** operating system products are dependent on several groups to provide necessary components for the various product releases.

9.1. Languages Group

The **NT** effort is dependent on the languages group to deliver the necessary programming tools for the 386, 486, and MIPS platforms to support self hosted development.

Programming tools to be supplied by the languages group are split into two groups; those required for 386 and 486 development, and those required for MIPS development. All tools must be ported to the **NT** environment and run under the **NT** operating system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following is a list of the 386 and 486 tools to be delivered by the languages group:

76. C compiler with structured exception handling.

77. x86 Assembler.

78. Linker capable of producing **NT** format images.

79. User debugger capable of supporting multi-thread debugging.

80. Kernel debugger capable of supporting multi-processor debugging.

The following is a list of the MIPS tools to be delivered by the languages group:

81. User debugger capable of supporting multi-thread debugging.

82. Kernel debugger capable of supporting multi-processor debugging.

The languages group is also planning to deliver a C compiler for MIPS that supports structured exception handling. However, this compiler will not be available for use in time to support the self hosting of **NT** development. Therefore, the MIPS C compiler, which also supports structured exception handling, is being ported to the **NT** environment as a backup.

A linker capable of linking MIPS object modules into an executable image will be provided by the NT Base System group.

The MIPS assembler is being ported to o the NT OS/2 environment to support self hosting and product develoment in assembly language.

C++?? C++ seh??

9.2. LanMan Group

Lan group for UI components, RPC stub compiler and runtime, TCP/IP transport and utiltiies.

9.3. Testing Group

Testing group for ??

9.4. User Ed Group

NT system services manual - who does?

Driver writers course?

NDIS driver writers course.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Where will the documentation for the MIPS compiler and assembler come from? Who will do?

10. Hardware Plans

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT I/O System Specification

Author: *Darryl E. Havens*

Revision 1.7, May 1, 1995

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. INTRODUCTION......1

2. OVERVIEW......1

3. USER APIS......4

3.1 CREATE/OPEN FILE/DEVICE SERVICES......4
3.1.1 Creating and Opening Files......4
3.1.2 Opening Files......14
3.2 FILE DATA SERVICES......20
3.2.1 Reading Files......20
3.2.2 Writing Files......22
3.3 DIRECTORY MANIPULATION SERVICES......25
3.3.1 Enumerating Files in a Directory......25
3.3.2 Enumerating Files in an Ole Directory File......33
3.3.3 Monitoring Directory Modifications......38
3.4 FILE SERVICES......41
3.4.1 Obtaining Information about a File......41
3.4.2 Changing Information about a File......52
3.4.3 Obtaining Extended Attributes for a File......61
3.4.4 Changing Extended Attributes for a File......63
3.4.5 Locking Byte Ranges in Files......64
3.4.6 Unlocking Byte Ranges in Files......66
3.5 FILE SYSTEM SERVICES......67
3.5.1 Obtaining Information about a File System Volume......67
3.5.2 Changing Information about a File System Volume......72
3.5.3 Obtaining Quota Information about a File System Volume......74
3.5.4 Changing Quota Information about a File System Volume......76
3.5.5 Controlling File Systems......77
3.6 MISCELLANEOUS SERVICES......78
3.6.1 Flushing File Buffers......78
3.6.2 Canceling Pending I/O on a File......79
3.6.3 Miscellaneous I/O Control......79
3.6.4 Deleting a File......81
3.6.5 Querying the Attributes of a File......81
3.7 I/O COMPLETION OBJECTS......83
3.7.1 Creating/Opening I/O Completion Objects......83
3.7.2 Operating on I/O Completion Objects......87

4. NAMING CONVENTIONS......89

5. APPENDIX A - TIME FIELD CHANGES......90

5.1 LAST ACCESS TIME......90
5.2 LAST MODIFY TIME......90
5.3 LAST CHANGE TIME......91

6. REVISION HISTORY......92

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the basic overall API for the I/O system of the **Windows NT** operating system. The I/O system is responsible for the management of all input and output operations in the system and for presenting the remainder of the system with a uniform and device-independent view of the various devices connected to the system.

The I/O system provides an interface for the user to perform I/O to various devices attached to the machine. The I/O operations in this API provide the user with a rich set of primitives to manipulate files and devices in such a way as to hide most of the particulars of how the device actually works.

The I/O system also provides system programmers with the ability to write their own device drivers for those devices that **Windows NT** does not support as part of its regular SDK. This part of the I/O system is documented in the *Windows NT Driver Model Specification* and is beyond the scope of this specification.

This specification does not attempt to exhaustively enumerate all error conditions that occur on all paths or indicate the errors that can occur after calling an API.

2. Overview

The user interface model that **Windows NT** uses for I/O consists of several different routines that perform such operations as Open, Read, Write, Close, etc. For other operations that are not included in the general set of routines, there is an **NtDeviceIoControlFile** service. This service allows device-dependent information to be passed to and from the device in a well structured manner. Likewise, the **NtFsControlFile** service which allows file-system-dependent information to be passed to and from the file system in a well structured manner.

The I/O system is designed to support both OS/2 and POSIX I/O operations easily to provide source code compatibility with those standards. This allows users familiar with those systems to continue to program using those interfaces without having to learn a new I/O programming model. The OS/2 and POSIX subsystems emulate the I/O services on top of the **Windows NT** services.

To perform I/O operations in **Windows NT**, a *file handle* must be specified. File handles are obtained by calling the **NtCreateFile** or **NtOpenFile** services. These services either create or open a file and return a handle to it. Alternatively, they may open a device directly and return a handle to the device. In each case the handle is still referred to as a "file handle" throughout the description of the APIs in this specification.

From the point of view of the object management system, a file is a *persistent object*. That is, a *file object* is treated like any other object in the system except that it remains intact across system boots. Handles to file objects, and therefore devices (depending on how the "file" was opened) are usable in the object system.

Some of the I/O interfaces in **Windows NT** are synchronous and others are asynchronous. For the latter type, it is up to the caller to wait for the I/O operation to complete. This may be done in either an alertable or a non-alertable manner. A file object in **Windows NT** is a waitable object and can therefore be used to synchronize completion of an I/O operation on the file. When a request is made to perform an operation on a file, the file object is set to the Not-Signaled state. When the operation completes, the file object is set to the Signaled state.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Each asynchronous I/O service also optionally accepts an event and/or the address of an *Asynchronous Procedure Call* **(APC)** to be executed when the operation completes. If an event is specified, the system sets it to the Not-Signaled state when the I/O operation is requested and sets it to the Signaled state when the I/O operation completes. The system will not normally set both the File object and the event to the Signaled state. That is, if an event is specified, then the event should be used for I/O completion synchronization; otherwise the file object handle should be used.

If an APC is specified, the procedure is invoked when the I/O completes with a parameter that is also supplied to the service. The procedure is also passed the address of the I/O status block discussed below.

Likewise, it is also possible to synchronize the completion of I/O operations through the use of *I/O Completion objects*. An I/O Completion object may be associated with a file such that a pool of threads may wait on the completion of all I/O associated with the object.

All service calls include the address of an *I/O status block*. This variable contains information about the success or failure of the operation once the operation has been completed. This allows the caller to determine the status of the operation once the file object or the event has been set to the Signaled state, or the APC routine has been invoked. Upon completion of the I/O operation the variable may also contain more information that is service-dependent.

It should be noted that performing multiple operations on a file at the same time requires that each operation be synchronized. That is, requesting two asynchronous reads from a file and then waiting on the file object will not guarantee that both operations have completed. In the same manner, using the same event to synchronize these two operations will not work either. Each operation must have its own event associated with it, or the caller must set up an APC which will be able to distinguish between the completion of each request.

Using an I/O system design whose primary data movement operations can be totally asynchronous makes writing faster programs easier. It frees the programmer from inventing methods of passing I/O requests to another thread to gain parallelism. This means that the main loop need not be blocked or concerned with the completion of I/O operations until it absolutely requires the requested data.

This particular design also allows servers and network servers to be written so that it is not necessary to dedicate a thread in the server to each request or to each client. Because the APC routine can be executed any time the server thread is ready for it, a single server thread can potentially perform I/O for an unlimited number of clients using very few system resources.

Since all potentially long I/O operations are asynchronous, a thread that is waiting on an I/O operation in an alertable manner may fall out of the wait. This allows programs to be written so that rundown and cleanup are much easier to control. Likewise, because the user has a choice, programs can still be written to block in a non-alertable manner and simply wait for the I/O operation to complete. More information on alerts can be found in the *Windows NT Process Structure* specification.

The **Windows NT** I/O system provides one optimization that can be used to save extraneous system calls. If the request for an operation is successfully queued to a driver for completion later, then the return status from the service is *STATUS_PENDING*. However, if the operation successfully completes before the service returns because the driver immediately completed the operation, then a status of *STATUS_SUCCESS* is returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

It is also possible to write an application that ignores the fact that the **Windows NT** I/O system is asynchronous by specifying that all I/O calls for a particular file object be performed synchronously. Further, the I/O operations are selectively alertable or non-alertable. This option is requested when the file is opened or created. If the I/O is being performed with alerts enabled, then it is possible for the I/O operation to be interrupted by an alert to the thread. It is also possible to specify that no alerts may be taken during the I/O operation.

If an application is performing I/O to a file in an alertable manner, then it must be written to be prepared for the I/O to fail because an alert occurred or an APC was delivered. In either case the I/O operation must be restarted by invoking the API again.

When the I/O system is performing synchronous I/O on a file object, it also maintains a current file pointer context for the file. This file pointer may be read or written using APIs provided by the I/O system. Furthermore, they are automatically updated whenever the file is read or written according to the number of bytes transferred. It is also possible to set the file pointer context on the read or write operation.

Performing synchronous I/O on a file object also means that the I/O to the file is serialized. That is, if Thread A has issued an I/O operation on a file and Thread B issues an I/O operation using the same file object, then Thread B will wait (alertable or non-alertable, depending on how the file was opened) until Thread A's I/O completes.

All of these features help the user deal with the system and use it to perform I/O the way that he wants to work. He can still take advantage of APC routines, for example, even if he is performing synchronous I/O. However, he doesn't have to if that isn't what he needs.

In order to access a file or a device, the caller must have permission to access the device in the requested manner. For example, some devices are considered single user devices. This is accomplished through the object management system in **Windows NT**. The object that represents a device is called a *device object*. Device objects may be created by device drivers using the *exclusive attribute*. This attribute indicates that only one process may open the object. Any other attempt to open a device from a process other than the "owning" process will fail. This implies that it is possible for a process to "own" a device. Of course, since handles can be inherited by child processes, then children of the owning process may share the device with the parent process.

A file or a device may specify an *Access Control List* **(ACL)**. An ACL is a list of *Access Control Entries* **(ACEs)** that specify what access rights a user has to the file or device. The user must have the requested access in order to successfully perform operations on the object.

Windows NT also provides file sharing among threads within a process and between processes. Because of the object architecture design used in **Windows NT**, it is possible for all of the threads within a process to access a file that one of the threads "opened" by using the returned file handle. Furthermore, a process that is created by one of the threads may also have access to the file if the file object is opened so that its handle is inheritable.

Finally, **Windows NT** provides file sharing by allowing multiple processes to open the same file. A file can be opened so that other processes may read, write, or perform both or neither operation on the file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. User APIs

The following sections present the user interface to the I/O system.

3.1 Create/Open File/Device Services

When a user wishes to access a file or a device, he must create or open it. This causes a handle to be returned that can then be used to manipulate the file or device in subsequent calls.

File handles are closed via the generic **NtClose** service. This service is discussed elsewhere in the **Windows NT** documentation. It should be noted that, just like all other system objects, a file is not actually deleted until all of the valid handles to it are closed and no referenced pointers remain.

The user APIs that supports creating and opening files and opening devices is as follows:

> **NtCreateFile -** Create or open a file and return a file handle.
> **NtOpenFile -** Open a file and return a file handle.

3.1.1 Creating and Opening Files

A file can be created or opened using the **NtCreateFile** service:

NTSTATUS
NtCreateFile(
 OUT PHANDLE *FileHandle,*
 IN ACCESS_MASK *DesiredAccess,*
 IN POBJECT_ATTRIBUTES *ObjectAttributes,*
 OUT PIO_STATUS_BLOCK *IoStatusBlock,*
 IN PLARGE_INTEGER *AllocationSize* **OPTIONAL***,*
 IN ULONG *FileAttributes,*
 IN ULONG *ShareAccess,*
 IN ULONG *CreateDisposition,*
 IN ULONG *CreateOptions,*
 IN PVOID *EaBuffer* **OPTIONAL,**
 IN ULONG *EaLength*
 );

Parameters:

FileHandle - A pointer to a variable that receives the file handle value.

DesiredAccess - Specifies the type of access that the caller requires to the file.

DesiredAccess Flags

SYNCHRONIZE - The file handle may be waited on to synchronize with the completion of the I/O operation.

DELETE - The file may be deleted.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

READ_CONTROL - The ACL and ownership information associated with the file may be read.

WRITE_DAC - The Discretionary ACL associated with the file may be written.

WRITE_OWNER - Ownership information associated with the file may be written.

FILE_READ_DATA - Data may be read from the file.

FILE_WRITE_DATA - Data may be written to the file.

FILE_EXECUTE - Data may be faulted into memory from the file via paging I/O.

FILE_APPEND_DATA - Data may only be appended to the file.

FILE_READ_ATTRIBUTES - File attributes flags may be read.

FILE_WRITE_ATTRIBUTES - File attributes flags may be written.

FILE_READ_EA - Extended attributes associated with the file may be read.

FILE_WRITE_EA - Extended attributes associated with the file may be written.

The three following values are the generic access types that the caller may request. The mapping to specific access rights is given for each:

GENERIC_READ - Maps to *STANDARD_RIGHTS_READ, FILE_READ_DATA, FILE_READ_ATTRIBUTES,* and *FILE_READ_EA*.

GENERIC_WRITE - Maps to *STANDARD_RIGHTS_WRITE, FILE_WRITE_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA,* and *FILE_APPEND_DATA*.

GENERIC_EXECUTE - Maps to *STANDARD_RIGHTS_EXECUTE, SYNCHRONIZE,* and *FILE_EXECUTE*.

For more information about the standard rights accesses, see the *Windows NT Local Security Specification*.

If the file being created or opened is a directory file, as specified in the *CreateOptions* argument, then the following types of access may be requested:

FILE_LIST_DIRECTORY - Files in the directory may be listed.

FILE_TRAVERSE - The directory may be traversed. That is, it may be in the pathname of a file.

FILE_READ_DATA, FILE_WRITE_DATA, FILE_EXECUTE, and *FILE_APPEND_DATA* accesses are not valid when creating or opening a directory file.

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, a security descriptor, a quality of service descriptor, and a set of file object attribute flags.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the file to be created or opened. This file specification must be a fully qualified file specification or the name of a device, unless it is a file relative to the directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the file specified by the *ObjectName* field is a file specification relative to the directory file supplied by this handle.

PSECURITY_DESCRIPTOR *SecurityDescriptor* - Optionally specifies the security descriptor that should be applied to the file. The ACLs specified by the security descriptor are only applied to the file if it is created. If not supplied and the file is created, then the ACL placed on the file is file-system-dependent, but most file systems propagate some part of the ACL from the parent directory file combined with the caller's default ACL.

PSECURITY_QUALITY_OF_SERVICE *SecurityQualityOfService* - Specifies the access a server should be given to the client's security context. This field is only used when a connection to a protected server is established. It allows the caller to control which parts of his security context are made available to the server and whether or not the server may impersonate the caller.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_INHERIT - Indicates that the handle to the file is to be inherited by the new process when an **NtCreateProcess** operation is performed to create a new process.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The actual action taken by the system is written to the *Information* field of this variable.

AllocationSize - Optionally specifies the initial allocation size of the file in bytes. The size has no effect unless the file is created, overwritten, or superseded.

FileAttributes - Specifies the file attributes for the file. Any combination of flags is acceptable except that all other flags override the normal file attribute, *FILE_ATTRIBUTE_NORMAL*. File attributes are only applied to the file if it is created, superseded, or, in some cases, overwritten. See the description in the text below for more details.

FileAttributes Flags

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_ATTRIBUTE_NORMAL - A normal file should be created.

FILE_ATTRIBUTE_READONLY - A read-only file should be created.

FILE_ATTRIBUTE_HIDDEN - A hidden file should be created.

FILE_ATTRIBUTE_SYSTEM - A system file should be created.

FILE_ATTRIBUTE_ARCHIVE - The file should be marked so that it will be archived.

FILE_ATTRIBUTE_TEMPORARY - A temporary should be created.

FILE_ATTRIBUTE_COMPRESSED - A compressed file should be created.

FILE_ATTRIBUTE_OFFLINE - An off-line file should be created.

ShareAccess - Specifies the type of share access that the caller would like to the file.

ShareAccess Flags

FILE_SHARE_READ - Other open operations may be performed on the file for read access.

FILE_SHARE_WRITE - Other open operations may be performed on the file for write access.

FILE_SHARE_DELETE - Other open operations may be performed on the file for delete access.

CreateDisposition - Specifies the actions to be taken if the file does or does not already exist.

CreateDisposition Values

FILE_SUPERSEDE - Indicates that if the file already exists then it should be superseded by the specified file. If it does not already exist then it should be created.

FILE_CREATE - Indicates that if the file already exists then the operation should fail. If the file does not already exist then it should be created.

FILE_OPEN - Indicates that if the file already exists it should be opened rather than creating a new file. If the file does not already exist then the operation should fail.

FILE_OPEN_IF - Indicates that if the file already exists, it should be opened. If the file does not already exist then it should be created.

FILE_OVERWRITE - Indicates that if the file already exists it should be opened and overwritten. If the file does not already exist then the operation should fail.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_OVERWRITE_IF - Indicates that if the file already exists it should be opened and overwritten. If the file does not already exist then it should be created.

CreateOptions - Specifies the options that should be used when creating or opening the file.

CreateOptions Flags

FILE_DIRECTORY_FILE - Indicates that the file being created or opened is a directory file. The *CreateDisposition* parameter must be set to one of *FILE_CREATE, FILE_OPEN,* or *FILE_OPEN_IF*.

FILE_NON_DIRECTORY_FILE - Indicate that the file being opened may not be a directory file.

FILE_WRITE_THROUGH - Indicates that services that write data to the file must actually write the data to the file before the operation is considered to be complete.

FILE_SEQUENTIAL_ONLY - Indicates that the file will only be accessed sequentially.

FILE_RANDOM_ACCESS - Indicates that the file will be accessed randomly so no sequential read ahead operations should be performed on the file.

FILE_NO_INTERMEDIATE_BUFFERING - Indicates that no caching or intermediate buffering is performed for the file.

FILE_SYNCHRONOUS_IO_ALERT - Indicates that all operations on the file are performed synchronously. Any wait being performed on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file position context.

FILE_SYNCHRONOUS_IO_NONALERT - Indicates that all operations on the file are performed synchronously. Waits in the system to synchronize I/O queueing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file position context.

FILE_CREATE_TREE_CONNECTION - Indicates that a tree connection is to be created.

FILE_COMPLETE_IF_OPLOCKED - Indicates that the operation should complete immediately with an alternate success code if the target file is oplocked rather than blocking the caller's thread.

FILE_NO_EA_KNOWLEDGE - Indicates the if the EAs on an existing file being opened indicate that the caller must understand EAs to properly interpret the file, then the file open should fail because the caller does not understand how to deal with EAs.

FILE_DELETE_ON_CLOSE - Indicates that the file should be deleted when the last handle to it is closed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_OPEN_BY_FILE_ID - Indicates that the file name contains the name of the device, and a 64-bit ID that is to be used to open the file.

FILE_OPEN_FOR_BACKUP_INTENT - Indicates that the file is being opened for backup intent, hence, the system should check for **SeBackupPrivilege** or **SeRestorePrivilege** and grant the caller the appropriate accesses to the file before checking the *DesiredAccess* against the file's security descriptor.

FILE_TRANSACTED_MODE - Indicates that the file is to be opened in transacted mode. This specifies that no changes to the file should be visible to other openers of the file until the transaction is committed.

FILE_RESERVE_OPFILTER - Indicates that a filter oplock should be reserved on the file if possible. The first I/O operation on the file must be an oplock request so that the caller can determine whether or not the oplock was granted.

FILE_OPEN_OFFLINE_FILE - Indicates that if the target file has been moved from primary storage and the target file is an off-line file, then the marker itself is to be opened rather than retrieving the actual file.

FILE_STORAGE_TYPE_SPECIFIED - Indicates that this *CreateOptions* parameter specifies a storage type field.

FILE_STORAGE_TYPE_DEFAULT - Create/open a file of default storage type.

FILE_STORAGE_TYPE_DIRECTORY - Create/open an enumerable directory file.

FILE_STORAGE_TYPE_FILE - Create/open normal data file.

FILE_STORAGE_TYPE_DOCFILE - Create/open a document file.

FILE_STORAGE_TYPE_JUNCTION_POINT - Create/open a junction point.

FILE_STORAGE_TYPE_CATALOG - Create/open a summary catalogue.

FILE_STORAGE_TYPE_STRUCTURED_STORAGE - Create/open structured storage.

FILE_STORAGE_TYPE_EMBEDDING - Create/open an embedding.

FILE_STORAGE_TYPE_STREAM - Create/open an alternate data stream on a file.

EaBuffer - Optionally specifies a list of EAs that should be set on the file if it is created. This is done as an atomic operation. That is, if an error occurs setting the EAs on the file, then the file will not be created.

EaLength - Supplies the length of the *EaBuffer*. If no buffer is supplied then this value should be zero.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The I/O status block specified by the *IoStatusBlock* parameter has the following type definition:

```
typedef struct _IO_STATUS_BLOCK {
        NTSTATUS Status;
        ULONG Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

Field	Description
Status	Final status of the operation
Information	Additional information about the operation

The **NtCreateFile** service either causes a new file (or directory) to be created, or it opens an existing file or device. The action taken is dependent on the name of the object being opened, whether the object already existed, and the specified create disposition value. A file handle is returned that can be used by subsequent service calls to manipulate the file itself or the data within the file.

There are two basic ways to specify the name of the file that is to be created/opened:

o - A fully qualified pathname. This method simply supplies the full file specification for the file. This is done using the *ObjectName* field of the *ObjectAttributes* structure. No *RootDirectory* handle may be specified.

o - A relative pathname. This method supplies the name of the file as a relative pathname. The path is relative to the directory file represented by the handle in the *RootDirectory* field of the *ObjectAttributes* structure.

Once the I/O operation is complete, the *Information* field of the I/O status block contains information about the action actually taken by the system. That is, one of *FILE_SUPERSEDED, FILE_CREATED, FILE_OPENED,* or *FILE_OVERWRITTEN,* is returned in this field.

The *SYNCHRONIZE* desired-access flag must be set in order for the caller to wait on the file handle to synchronize I/O completion. If this desired access is not specified, then I/O completion must be synchronized through the use of an event or an APC routine.

If *FILE_EXECUTE* is the only desired-access flag specified other than *SYNCHRONIZE*, then the caller cannot directly read or write any data in the file using the returned file handle. All operations on the file occur through the system pager in response to instruction and data accesses.

If *FILE_APPEND_DATA* is the only desired-access flag specified other than *SYNCHRONIZE*, then the caller can only write to the end of the file. Any offset information on writes to the file is ignored. The file will automatically be extended as necessary for these types of write operations.

Specifying the *FILE_WRITE_DATA* desired-access flag for a file also allows writes beyond the end of the file to occur. The file is also automatically extended for these types of writes as well.

Files may be shared among threads within a process, or among a family of processes through inheritance, by simply opening or creating the file. The file handle can then be used to access the same file. Note that the OBJ_INHERIT object attribute flag must be specified in the *ObjectAttributes* parameter in order for sharing to occur between parent and child processes through use of the file handle.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Access to a file may be shared among separate cooperating processes or threads by requesting that the file system open the file for shared access. This is accomplished through the flags in the *ShareAccess* mode parameter. Provided that both file openers have the privilege to access the file in the specified manner, the file can be successfully opened and shared. If the caller does not specify *FILE_SHARE_READ, FILE_SHARE_WRITE,* or *FILE_SHARE_DELETE,* then no other open operations may be performed on the file.

In order for the file to be successfully opened, the requested access mode to the file must be compatible with the way in which other opens to the file have been made. That is, the desired access mode to the file must not conflict with the accesses that other openers of the file have disallowed.

The *FILE_SUPERSEDE* disposition value specifies that if the file does not already exist, it is to be created. If the file already exists, then it should be superseded. Superseding a file requires that the accessor have delete access to the existing file. That is, the existing file is effectively deleted and then recreated. This implies that if someone else already has the file open, they have specified that the file may be deleted by another file opener. This is done by specifying a *ShareAccess* parameter with the *FILE_SHARE_DELETE* flag set. This type of disposition is consistent with the Unix style of overwriting files.

The *FILE_OVERWRITE_IF* disposition value is much like the *FILE_SUPERSEDE* disposition value. If the file exists, then it will be overwritten; if it does not already exist then it will be created. Overwriting a file is semantically equivalent to a supersede operation except that it requires write access to the file rather than delete access. That is, the requestor must have write access to the file and if someone else already has the file open, they must have specified that the file may be written by another file opener. This is done by specifying a *ShareAccess* parameter with the *FILE_SHARE_WRITE* flag set. Another difference between an overwrite and a supersede is that the specified file attributes are logically OR'd with those already on the file. That is, the caller may not turn off any flags already set in the attributes but may turn others on. This style of overwriting files is consistent with DOS and OS/2.

The *FILE_OVERWRITE* disposition value performs exactly the same operation as a *FILE_OVERWRITE_IF*, except that if the file does not already exist the operation will fail.

The *FILE_DIRECTORY_FILE* option specifies that the file to be created or opened is a directory file. If this option is specified, then the *CreateDisposition* parameter must be set to one of *FILE_CREATE, FILE_OPEN,* or *FILE_OPEN_IF*. Likewise, the only create options that may be specified are *FILE_SYNCHRONOUS_IO_ALERT, FILE_SYNCHRONOUS_IO_NONALERT, FILE_WRITE_THROUGH, FILE_OPEN_FOR_BACKUP_INTENT,* and *FILE_OPEN_BY_FILE_ID*. When a directory file is created, the file system creates an appropriate structure on the disk to represent an empty directory for that particular file system's on-disk structure. If this option was specified and the file being opened is not a directory file, then the API will fail.

Conversely, the *FILE_NON_DIRECTORY_FILE* option specifies that the target file being opened may not be a directory file. It must be a data file, device, volume, etc., or the API is to fail.

It is also possible to further control the type of file, directory, structured storage, etc. that one wishes to create or open by providing the *FILE_STORAGE_TYPE_SPECIFIED* flag. This flag indicates that one of the *FILE_STORAGE_TYPE_xxx* values has been supplied. Note that specifying *FILE_DIRECTORY_FILE* is equivalent to specifying *FILE_STORAGE_TYPE_SPECIFIED* and also specifying *FILE_STORAGE_TYPE_DIRECTORY*. Likewise, specifying

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_NON_DIRECTORY_FILE is equivalent to specifying *FILE_STORAGE_TYPE_SPECIFIED* and also specifying *FILE_STORAGE_TYPE_FILE.*

The *FILE_NO_INTERMEDIATE_BUFFER* option specifies that the file system should not perform any intermediate buffering on behalf of the caller. This causes several restrictions to be placed on the caller's parameters to various service calls.

o - The byte offset parameter to read and write operations must be an integral number of 512-byte blocks.

o - The length of the read or write operation must be an integral number of 512-byte blocks. Note that specifying a read operation to a buffer whose length is 512 bytes may result in a smaller number of significant bytes being transferred to the buffer because the end of the file was reached, however, the driver may still be able to transfer a whole sector of data directly to the buffer.

o - Buffers must be aligned to that of the device. The device alignment requirement can be determined by querying the file.

o - Files opened for this type of access may not be opened for *FILE_APPEND_DATA* access.

o - The *FILE_WRITE_THROUGH* option is automatically set when intermediate buffering is disabled.

o - Calls to set the file position pointer for files opened in this manner may only specify offsets wto 512-byte sector boundaries.

The *FILE_SYNCHRONOUS_IO_ALERT* and *FILE_SYNCHRONOUS_IO_NONALERT* create options allow the caller to specify that all I/O operations on this file are to be performed synchronously as long as they occur through the file object referred to by the returned handle. The system also maintains the current "file pointer context" for the file when the file is opened/created with either of these options. Likewise, all I/O on the file will be serialized across all threads and processes using the returned handle or an inherited copy of the handle. The *SYNCHRONIZE* desired-access flag must also be specified so that the I/O system can use the file object as a synchronization object. Of course, these two options are mutually exclusive.

These two options also imply that the I/O system maintain an internal current file position pointer. This pointer can be used by the read and write services. It can also be set or read by other APIs described later in this document.

The *FILE_CREATE_TREE_CONNECTION* option specifies that a tree connection to a remote node is to be created. For more information, see the *Windows NT LAN Manager Software* specification.

The *FILE_COMPLETE_IF_OPLOCKED* option specifies that if the target file is currently oplocked by another accessor of the file, that the operation should complete immediately anyway without waiting for the oplock break operation to be completed. The call to **NtCreateFile** completes once the oplock break operation has been started, rather than blocking the caller's thread waiting for the break to complete. An alternate success code is returned to the caller if an oplock break is in progress when the service completes. This flag is mutually exclusive with the *FILE_RESERVE_OPFILTER* flag. For more information on oplocks, see the *Windows NT Opportunistic Locking Design Note*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Setting the *FILE_TRANSACTED_MODE* option indicates that the file system and Transaction Manager should work together to only allow other openers of the file to see changes to the file when they are fully committed. This means that other openers will not normally see any writes to the file unless the data has actually been committed.

The *FILE_RESERVE_OPFILTER* option indicates that the caller would like to reserve a filter oplock on the file, if possible. This flag is mutually exclusive with the *FILE_COMPLETE_IF_OPLOCKED* flag. The first I/O request issued on the file must be an oplock *FSCTL* to determine whether or not the oplock was actually reserved. For more information on oplocks, see the *Windows NT Opportunistic Locking Design Note*.

A file is considered to have been moved from primary storage and a marker left in its place if the target file's *FILE_ATTRIBUTE_OFFLINE* attribute bit is set. A normal attempt to open such a file causes the HSM(s) in the system to attempt to retrieve the original file. However, the marker itself can be opened by specifying the *FILE_OPEN_OFFLINE_FILE* option.

If a list of EAs is supplied through specifying an *EaBuffer*, then those EAs are applied to the file as an atomic operation. Note that the EAs are only set on the file if the file is created (this also includes supersede and overwrite operations). If setting the EAs on the file incurs an error, then the file is not created, an appropriate error is returned, and the *Information* field of the *IoStatusBlock* variable is set to the offset into the EA buffer of the EA that caused the error.

The type of the contents of the *EaBuffer* is *FILE_FULL_EA_INFORMATION*. This type has the following definition:

```
typedef struct _FILE_FULL_EA_INFORMATION {
        ULONG NextEntryOffset;
        UCHAR Flags;
        UCHAR EaNameLength;
        USHORT EaValueLength;
        CHAR EaName[];
} FILE_FULL_EA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
Flags	Flags to be associated with the EA
EaNameLength	Length of the EA's name field, excluding null termination character
EaValueLength	Length of the EA's value field
EaName	The name of the EA

The flags currently defined for EAs are:

FILE_NEED_EA- This flag indicates that the caller must understand EAs in order to understand the actual meaning or representation of the file. Files who have an EA with this flag set cannot be seen by callers attempting to access the file with the *FILE_NO_EA_KNOWLEDGE CreateOption* set.

The value field begins after the end of the *EaName* field of the structure, including a single null character. The *EaNameLength* field does not include the null character in the count Each entry in the list must be longword aligned. The *NextEntryOffset* field specifies the number of bytes between the current entry and the next entry in the buffer. If there are no more entries following the current entry, then the value of this field is zero.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

For more information, refer to the **NtSetEaFile** system service documented elsewhere in this specification.

3.1.2 Opening Files

A file can be opened using the **NtOpenFile** service:

NTSTATUS
NtOpenFile(
    OUT PHANDLE *FileHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    IN ULONG *ShareAccess,*
    IN ULONG *OpenOptions*
    );

Parameters:

FileHandle - A pointer to a variable that receives the file handle value.

DesiredAccess - Specifies the type of access that the caller requires to the file.

DesiredAccess Flags

SYNCHRONIZE - The file handle may be waited on to synchronize with the completion of the I/O operation.

DELETE - The file may be deleted.

READ_CONTROL - The ACL and ownership information associated with the file may be read.

WRITE_DAC - The Discretionary ACL associated with the file may be written.

WRITE_OWNER - Ownership information associated with the file may be written.

FILE_READ_DATA - Data may be read from the file.

FILE_WRITE_DATA - Data may be written to the file.

FILE_EXECUTE - Data may be faulted into memory from the file via paging I/O.

FILE_APPEND_DATA - Data may only be appended to the file.

FILE_READ_ATTRIBUTES - File attributes flags may be read.

FILE_WRITE_ATTRIBUTES - File attributes flags may be written.

FILE_READ_EA - Extended attributes associated with the file may be read.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_WRITE_EA - Extended attributes associated with the file may be written.

FILE_LIST_DIRECTORY - Files in the directory may be listed.

FILE_TRAVERSE - The directory may be traversed. That is, it may be in the pathname of a file.

The three following values are the generic access types that the caller may request. The mapping to specific access rights is given for each:

GENERIC_READ - Maps to *STANDARD_RIGHTS_READ, FILE_READ_DATA, FILE_READ_ATTRIBUTES,* and *FILE_READ_EA*.

GENERIC_WRITE - Maps to *STANDARD_RIGHTS_WRITE, FILE_WRITE_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA,* and *FILE_APPEND_DATA*.

GENERIC_EXECUTE - Maps to *STANDARD_RIGHTS_EXECUTE, SYNCHRONIZE,* and *FILE_EXECUTE*.

For more information about standard rights accesses, see the *Windows NT Local Security Specification*.

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, and a set of file object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the file to be opened. This file specification must be a fully qualified file specification or the name of a device, unless it is a file relative to the directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the file specified by the *ObjectName* field is a file specification relative to the directory file supplied by this handle.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_INHERIT - Indicates that the handle to the file is to be inherited by the new process when an **NtCreateProcess** operation is performed to create a new process.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The actual action taken by the system is written to the *Information* field of

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

this variable. For a more information on this parameter see the **NtCreateFile** system service description.

ShareAccess - Specifies the type of share access that the caller would like to the file.

<u>ShareAccess Flags</u>

FILE_SHARE_READ - Other open operations may be performed on the file for read access.

FILE_SHARE_WRITE - Other open operations may be performed on the file for write access.

FILE_SHARE_DELETE - Other open operations may be performed on the file for delete access.

OpenOptions - Specifies the options that should be used when opening the file.

<u>OpenOptions Flags</u>

FILE_DIRECTORY_FILE - Indicates that the file being opened must be a directory file.

FILE_NON_DIRECTORY_FILE - Indicate that the file being opened may not be a directory file.

FILE_WRITE_THROUGH - Indicates that services that write data to the file must actually write the data to the file before the operation is considered to be complete.

FILE_SEQUENTIAL_ONLY - Indicates that the file will only be accessed sequentially.

FILE_RANDOM_ACCESS - Indicates that the file will be access randomly so no read ahead operations should ever be performed on the file.

FILE_NO_INTERMEDIATE_BUFFERING - Indicates that no caching or intermediate buffering is performed for the file.

FILE_SYNCHRONOUS_IO_ALERT - Indicates that all operations on the file are performed synchronously. Any wait being performed on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file position context.

FILE_SYNCHRONOUS_IO_NONALERT - Indicates that all operations on the file are performed synchronously. Waits in the system to synchronize I/O queueing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file position context.

FILE_COMPLETE_IF_OPLOCKED - Indicates that the operation should complete immediately with an alternate success code if the target file is oplocked rather than blocking the caller's thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_NO_EA_KNOWLEDGE - Indicates that if the EAs on an existing file being opened indicate that the caller must understand EAs to properly interpret the file, then the file open should fail because the caller does not understand how to deal with EAs.

FILE_DELETE_ON_CLOSE - Indicates that the file should be deleted when the last handle to it is closed.

FILE_OPEN_BY_FILE_ID - Indicates that the file name contains the name of the device, and a 64-bit ID that is to be used to open the file.

FILE_OPEN_FOR_BACKUP_INTENT - Indicates that the file is being opened for backup intent, hence, the system should check for **SeBackupPrivilege** or **SeRestorePrivilege** and grant the caller the appropriate accesses to the file before checking the *DesiredAccess* against the file's security descriptor.

FILE_TRANSACTED_MODE - Indicates that the file is to be opened in transacted mode. This specifies that no changes to the file should be visible to other openers of the file until the transaction is committed.

FILE_RESERVE_OPFILTER - Indicates that a filter oplock should be reserved on the file if possible. The first I/O operation on the file must be an oplock request so that the caller can determine whether or not the oplock was granted.

FILE_OPEN_OFFLINE_FILE - Indicates that if the target file has been moved from primary storage and the target file is an off-line file, then the marker itself is to be opened rather than retrieving the actual file.

FILE_STORAGE_TYPE_SPECIFIED - Indicates that this *CreateOptions* parameter specifies a storage type field.

FILE_STORAGE_TYPE_DEFAULT - Create/open a file of default storage type.

FILE_STORAGE_TYPE_DIRECTORY - Create/open an enumerable directory file.

FILE_STORAGE_TYPE_FILE - Create/open normal data file.

FILE_STORAGE_TYPE_DOCFILE - Create/open a document file.

FILE_STORAGE_TYPE_JUNCTION_POINT - Create/open a junction point.

FILE_STORAGE_TYPE_CATALOG - Create/open a summary catalogue.

FILE_STORAGE_TYPE_STRUCTURED_STORAGE - Create/open structured storage.

FILE_STORAGE_TYPE_EMBEDDING - Create/open an embedding.

FILE_STORAGE_TYPE_STREAM - Create/open an alternate data stream on a file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **NtOpenFile** service opens an existing file or device. A file handle is returned that can be used by subsequent service calls to manipulate the file itself or the data within the file.

There are two basic ways to specify the name of the file that is to be opened:

> o - A fully qualified pathname. This method simply supplies the full file specification for the file to be opened. This is done using the *ObjectName* field of the *ObjectAttributes* structure. No *RootDirectory* handle may be specified.
>
> o - A relative pathname. This method supplies the name of the file as a relative pathname. The path is relative to the directory file represented by the handle in the *RootDirectory* field of the *ObjectAttributes* structure.

Once the I/O operation is complete, the *Information* field of the I/O status block contains information about the action taken by the system. That is, the *Information* field will contain *FILE_OPENED*.

The *SYNCHRONIZE* desired-access flag must be set in order for the caller to wait on the file handle to synchronize I/O completion. If this desired access is not specified, then I/O completion must be synchronized through the use of an event or an APC routine.

If *FILE_EXECUTE* is the only desired-access flag specified other than *SYNCHRONIZE*, then the caller cannot directly read or write any data in the file using the returned file handle. All operations on the file occur through the system pager in response to instruction and data accesses.

If *FILE_APPEND_DATA* is the only desired-access flag specified other than *SYNCHRONIZE*, then the caller can only write to the end of the file. Any offset information on writes to the file is ignored. The file will automatically be extended as necessary for these types of write operations.

Specifying the *FILE_WRITE_DATA* desired-access flag for a file also allows writes beyond the end of the file to occur. The file is also automatically extended for these types of writes as well.

Files may be shared among threads within a process, or among a family of processes through inheritance, by simply opening or creating the file. The file handle can then be used to access the same file. Note that the OBJ_INHERIT object attribute flag must be specified in the *ObjectAttributes* parameter in order for sharing to occur between parent and child processes through use of the file handle.

Access to a file may be shared among separate cooperating processes or threads by requesting that the file system open the file for shared access. This is accomplished through the flags in the *ShareAccess* mode parameter. Provided that both file openers have the privilege to access the file in the specified manner, the file can be successfully opened and shared. If the caller does not specify *FILE_SHARE_READ, FILE_SHARE_WRITE,* or *FILE_SHARE_DELETE,* then no other open operations may be performed on the file.

In order for the file to be successfully opened, the requested access mode to the file must be compatible with the way in which other opens to the file have been made. That is, the desired access mode to the file must not conflict with the accesses that other openers of the file have disallowed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The *FILE_DIRECTORY_FILE* flag specifies that the file being opened must be a directory file or the service will fail. Likewise, the *FILE_NON_DIRECTORY_FILE* flag specifies that the service will fail if the file being opened is a directory file.

It is also possible to further control the type of file, directory, structured storage, etc. that one wishes to create or open by providing the *FILE_STORAGE_TYPE_SPECIFIED* flag. This flag indicates that one of the *FILE_STORAGE_TYPE_xxx* values has been supplied. Note that specifying *FILE_DIRECTORY_FILE* is equivalent to specifying *FILE_STORAGE_TYPE_SPECIFIED* and also specifying *FILE_STORAGE_TYPE_DIRECTORY*. Likewise, specifying *FILE_NON_DIRECTORY_FILE* is equivalent to specifying *FILE_STORAGE_TYPE_SPECIFIED* and also specifying *FILE_STORAGE_TYPE_FILE.*

The *FILE_NO_INTERMEDIATE_BUFFER* option specifies that the file system should not perform any intermediate buffering on behalf of the caller. This causes several restrictions to be placed on the caller's parameters to various service calls.

o - The byte offset parameter to read and write operations must be an integral number of 512-byte blocks.

o - The length of the read or write operation must be an integral number of 512-byte blocks. Note that specifying a read operation to a buffer whose length is 512 bytes may result in a smaller number of significant bytes being transferred to the buffer because the end of the file was reached, however, the driver may still be able to transfer a whole sector of data directly to the buffer.

o - Buffers must be aligned to that of the device. The device alignment requirement can be determined by querying the file.

o - Files opened for this type of access may not be opened for *FILE_APPEND_DATA* access.

o - The *FILE_WRITE_THROUGH* option is automatically set when intermediate buffering is disabled.

o - Calls to set the file position pointer for files opened in this manner may only specify offsets to 512-byte sector boundaries.

o - All opens of the file must either enable or disable this feature. That is, no mixed opens are permitted.

The *FILE_SYNCHRONOUS_IO_ALERT* and *FILE_SYNCHRONOUS_IO_NONALERT* open options allow the caller to specify that all I/O operations on this file are to be performed synchronously as long as they occur through the file object referred to by the returned handle. The system also maintains the current "file pointer context" for the file when the file is opened with either of these options. Likewise, all I/O on the file will be serialized across all threads and processes using the returned handle or an inherited copy of the handle. The *SYNCHRONIZE* desired access flag must also be specified so that the I/O system can use the file object as a synchronization object.

These two options also imply that the I/O system maintain an internal current file position pointer. This pointer can be used by the read and write services. It can also be set or read by other APIs described later in this document.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The *FILE_COMPLETE_IF_OPLOCKED* option specifies that if the target file is currently oplocked by another accessor of the file, that the operation should complete immediately anyway without waiting for the oplock break operation to be completed. The call to **NtOpenFile** completes once the oplock break operation has been started, rather than blocking the caller's thread waiting for the break to complete. An alternate success code is returned to the caller if an oplock break is in progress when the service completes. For more information on oplocks, see the *Windows NT Opportunistic Locking Design Note*.

Setting the *FILE_TRANSACTED_MODE* option indicates that the file system and Transaction Manager should work together to only allow other openers of the file to see changes to the file when they are fully committed. This means that other openers will not normally see any writes to the file unless the data has actually been committed.

The *FILE_RESERVE_OPFILTER* option indicates that the caller would like to reserve a filter oplock on the file, if possible. The first I/O request issued on the file must be an oplock *FSCTL* to determine whether or not the oplock was actually reserved. For more information on oplocks, see the *Windows NT Opportunistic Locking Design Note*.

A file is considered to have been moved from primary storage and a marker left in its place if the target file’s *FILE_ATTRIBUTE_OFFLINE* attribute bit is set. A normal attempt to open such a file causes the HSM(s) in the system to attempt to retrieve the original file. However, the marker itself can be opened by specifying the *FILE_OPEN_OFFLINE_FILE* option.

3.2 File Data Services

This section presents those services that read data from and write data to files. They provide the functionality to perform I/O to files according to the options provided in the open/create services.

The APIs that support reading and writing files are as follows:

> **NtReadFile** - Read data from a file into a specified buffer.
> **NtWriteFile** - Write data to a file from a specified buffer.

3.2.1 Reading Files

Data can be read from a file with the **NtReadFile** service:

NTSTATUS
NtReadFile(
> **IN HANDLE** *FileHandle,*
> **IN HANDLE** *Event* **OPTIONAL,**
> **IN PIO_APC_ROUTINE** *ApcRoutine* **OPTIONAL,**
> **IN PVOID** *ApcContext* **OPTIONAL,**
> **OUT PIO_STATUS_BLOCK** *IoStatusBlock,*
> **OUT PVOID** *Buffer,*
> **IN ULONG** *Length,*
> **IN PLARGE_INTEGER** *ByteOffset* **OPTIONAL,**
> **IN PULONG** *Key* **OPTIONAL**
> **);**

<u>Parameters:</u>

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileHandle - An open file handle to the file to read.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually read from the file is returned in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer to receive the bytes read from the file.

Length - The length of the specified *Buffer* in bytes. This is the number of bytes that are read from the file unless the end of the file is reached.

ByteOffset - Supplies the starting byte offset within the file where the read begins. An error is returned if an attempt is made to start the read beyond the end of the file.

See the note below about the semantics of this parameter if the I/O system is maintaining the current file pointer position.

Key - Optionally specifies a *Key* that is used to indicate the owner of a byte-range lock. If the value of the *Key* and other conditions are met, then the locked range is read.

The routine specified by the *ApcRoutine* parameter has the following type definition:

```
typedef
VOID
(*PIO_APC_ROUTINE) (
    IN PVOID ApcContext,
    IN PIO_STATUS_BLOCK IoStatusBlock
    );
```

Parameters:

ApcContext - This parameter is the value of *ApcContext* in the call to the I/O system service.

IoStatusBlock - This parameter is the pointer *IoStatusBlock* passed in the call to the I/O system service.

The **NtReadFile** service begins reading from the *ByteOffset* byte within the file into the specified *Buffer*. The read terminates under one of the following conditions:

o - The buffer is full. The number of bytes specified by the *Length* parameter has been read. Therefore, no more data can be placed into the buffer without an overflow.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - During the read operation the end of the file is reached. There is no more data in the file to be placed into the buffer.

If the file was opened or created without intermediate buffering by the file system, there are several restrictions on the parameters supplied to this service. See the descriptions of the **NtCreateFile** and **NtOpenFile** services for more information.

If *FILE_SYNCHRONOUS_IO_ALERT* or *FILE_SYNCHRONOUS_IO_NONALERT* are specified as options when the file is opened/created, then the I/O system maintains the current file position for the file. The caller may specify that the current file pointer position be used instead of a specific byte offset within the file in one of two ways:

o - Specifying a *ByteOffset* parameter whose value is *FILE_USE_FILE_POINTER_POSITION* rather than an actual byte offset within the file.

o - Not specifying the *ByteOffset* parameter at all.

Either of these methods causes the read to occur from the byte offset within the file according to the value of the current file pointer position. Once the read is complete, the pointer position is updated according to the number of bytes that were read from the file.

If the current file position is being maintained by the I/O system, then the caller may still read directly from a location in the file. This automatically changes the current file position to point to that position, performs the read, and then updates the position according to the number of bytes actually read. This gives the caller an atomic "seek and read" service. This is done by supplying the actual byte offset within the file to be read.

The *Key* parameter can optionally be used to specify a key value that is used to determine whether a locked range of bytes can be read by the caller. That is, locked ranges of bytes have a key associated with them using the **NtLockFile** system service. The *Key* parameter is one of the values that must exactly match the key associated with the lock in order to read the locked range of bytes. More information can be found later in this specification on byte range locking.

The **NtReadFile** service is also flexible enough to be invoked for most read functions directly by an RPC stub routine that is emulating a system service on behalf of an emulation subsystem. The OS/2 **DosRead** and POSIX **read** functions, for example, can both be emulated by directly invoking this service.

This service requires **FILE_READ_DATA** access to the file.

Once the data has been read, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.2.2 Writing Files

Data can be written to a file with the **NtWriteFile** service:

NTSTATUS

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NtWriteFile(
IN HANDLE *FileHandle,*
IN HANDLE *Event* **OPTIONAL,**
IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
IN PVOID *ApcContext* **OPTIONAL,**
OUT PIO_STATUS_BLOCK *IoStatusBlock,*
IN PVOID *Buffer,*
IN ULONG *Length,*
IN PLARGE_INTEGER *ByteOffset* **OPTIONAL,**
IN PULONG *Key* **OPTIONAL**
);

Parameters:

FileHandle - An open file handle to the file to write.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually written to the file is returned in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer containing the data that should be written to the file.

Length - The number of bytes to write to the file from the specified *Buffer*.

ByteOffset - Supplies the starting byte offset within the file where the write begins.

The notes below describe other valid values that this parameter can express.

Key - Optionally specifies a *Key* that it used to indicate the owner of a byte range lock. If the value of the *Key* and other conditions are met, then the locked range is written.

The **NtWriteFile** service begins writing *Length* bytes from the specified *Buffer* to the byte within the file specified by the *ByteOffset* parameter.

If the write occurs to a file beyond the current end of file mark, then the file is automatically extended and the end of file mark is updated. Any bytes not explicitly written between the old end of file mark and the new end of file mark are defined to be zero.

If the file is opened with only **FILE_APPEND_DATA** access, then the *ByteOffset* parameter is ignored. The data contained in the *Buffer*, for *Length* bytes, is written to the current end of the file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the file was opened or created without intermediate buffering by the file system, there are several restrictions on the parameters supplied to this service. See the descriptions of the **NtCreateFile** and **NtOpenFile** services for more information.

If *FILE_SYNCHRONOUS_IO_ALERT* or *FILE_SYNCHRONOUS_IO_NONALERT* are specified when the file is opened or created, then the I/O system maintains the current file position pointer. The caller may specify that the current file pointer position be used instead of a specific byte offset within the file in one of two ways:

o - Specifying a *ByteOffset* parameter whose value is *FILE_USE_FILE_POINTER_POSITION* rather than an actual byte offset within the file.

o - Not specifying the *ByteOffset* parameter at all.

Either of these methods causes the write to occur from the byte offset within the file according to the value of the current file pointer position context. Once the write is complete, the pointer position is updated according to the number of bytes that were written to the file.

If the current file position is being maintained by the I/O system, then the caller may still write directly to a location in the file. This automatically changes the current file position to point to that position, performs the write, and then updates the position according to the number of bytes written. This gives the user an atomic "seek and write" service. This is done by supplying the actual byte offset within the file to be written.

It is also possible to cause the write to take place at the current end of file. This can be done regardless of whether the I/O system is maintaining file position information. Specifying a value of *FILE_WRITE_TO_END_OF_FILE* for the *ByteOffset* parameter causes this to occur.

The *Key* parameter can optionally be used to specify a key value that determines whether a locked range of bytes can be written by the caller. That is, locked ranges of bytes have a key associated with them using the **NtLockFile** system service. The *Key* parameter is one of the values that must exactly match the lock specification associated with the lock in order to be able to write the locked range of bytes. More information can be found later in this specification on byte range locking.

The **NtWriteFile** service is also flexible enough to be invoked for most write functions directly by an RPC stub routine executing on behalf of an operating system emulation subsystem. The OS/2 **DosWrite** and POSIX **write** functions, for example, can both be emulated by directly invoking these services.

This service requires either **FILE_WRITE_DATA** or **FILE_APPEND_DATA** access to the file. Note that having only **FILE_APPEND_DATA** access to the file does not allow the caller to write anywhere in the file except at the current end of file mark, while having **FILE_WRITE_DATA** access to a file does not preclude the caller from writing to or beyond the end of the file.

Once the data has been written, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.3 Directory Manipulation Services

This section presents those services that manipulate directories within the file system.

The APIs that permit directory manipulation are as follows:

> **NtQueryDirectoryFile** - Enumerate files within a directory.
> **NtNotifyChangeDirectoryFile** - Monitor directory for modifications.
> **NtQueryOleDirectoryFile** - Enumerate streams and embeddings in the OLE name space.

3.3.1 Enumerating Files in a Directory

The files within a directory can be enumerated using the **NtQueryDirectoryFile** service:

NTSTATUS
NtQueryDirectoryFile(
    IN HANDLE *FileHandle,*
    IN HANDLE *Event* **OPTIONAL,**
    IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
    IN PVOID *ApcContext* **OPTIONAL,**
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    OUT PVOID *FileInformation,*
    IN ULONG *Length,*
    IN FILE_INFORMATION_CLASS *FileInformationClass,*
    IN BOOLEAN *ReturnSingleEntry,*
    IN PUNICODE_STRING *FileName* **OPTIONAL,**
    IN BOOLEAN *RestartScan*
    );

Parameters:

FileHandle - A file handle to an open directory file.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually written to the specified *Buffer* is stored in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FileInformation - A pointer to a buffer to receive information about the files in the directory. The contents of this buffer are defined by the *FileInformationClass* parameter below.

Length - The length of the specified buffer in bytes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileInformationClass - Specifies the type of information that is returned in the *FileInformation* buffer. The type of information in the buffer is defined by the following type codes.

FileInformationClass Values

FileNamesInformation - Specifies that names of files in the directory are written to the *FileInformation* buffer.

FileDirectoryInformation - Specifies that basic directory information about the files is written to the *FileInformation* buffer.

FileFullDirectoryInformation - Specifies that all of the directory information about the files is written to the *FileInformation* buffer.

FileBothDirectoryInformation - Specifies that all of the directory information about the files is written to the *FileInformation* buffer, including both of the file's names.

FileOleDirectoryInformation - Specifies that OLE directory information about the files is written to the *FileInformation* buffer.

ReturnSingleEntry - A BOOLEAN value that, if TRUE, indicates that only a single entry should be returned.

FileName - An optional file name within the specified directory. This parameter may only be specified on the first call to the service. It selects the files in the directory that the query calls return. The specification may contain wildcard characters.

RestartScan - A BOOLEAN value that, if TRUE, indicates that the scan should be restarted from the beginning. This causes the directory operation to restart the scan from the beginning of the directory.

The **NtQueryDirectoryFile** function operates on a directory file specified by the *FileHandle* parameter. The service returns information about files in the specified directory. The *ReturnSingleEntry* parameter specifies that only a single entry should be returned rather than filling the buffer. The actual number of files whose information is returned, is the smallest of the following:

o - One entry, if the *ReturnSingleEntry* parameter is TRUE.

o - The number of files whose information fits into the specified buffer.

o - The number of files that exist in the directory according to the wildcard file specification. This defaults to all of the files in the directory.

File systems supported by **Windows NT** return information about files in directories in either random or alphabetically ascending order. It is possible to receive information about a specific file by specifying the name of the file as the *FileName* parameter without using any wildcard characters.

If information about multiple files is returned, then each entry in the buffer will be aligned on a longword or quadword boundary, depending on the type of information being returned. Each type of information class returned begins with the byte offset required to find the next entry in the buffer. If

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

this value is zero, then there are no more entries following the current entry. Note that there are no entries in the buffer only if the service completes with an error.

The normal operation of this service is to return all of the files in the directory. A wildcard specification may be supplied the first time the service is called to select a subset of the files in the directory. This is done by supplying a wildcard file specification in the *FileName* parameter the first time the service is invoked once the directory file has been opened. Once a wildcard pattern has been supplied, all subsequent **NtQueryDirectoryFile** calls using the same directory handle operate only on those files which match the pattern. That is, restarting the listing will return the first entry in the directory that matches the pattern.

A wildcard file specification may only be supplied the first time that the service is invoked. If no wildcard specification is supplied, the file system assumes all of the files in the directory are selected. Wildcard file specifications must be consistent with those used in **OS/2 V2.0**.

Likewise, the *FileInformationClass* parameter specified the first time indicates the type of information about the files in the directory that is to be returned. Once an information class is established, it may not be changed in subsequent calls to the service. That is, all subsequent calls must pass the same information class as the first call to the service for a given handle.

The information that is returned in the buffer is defined by the following type codes and structures.

FileNamesInformation Format by File Information Class

FileNamesInformation - Data type is *FILE_NAMES_INFORMATION*.

```
typedef struct _FILE_NAMES_INFORMATION {
        ULONG NextEntryOffset;
        ULONG FileIndex;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_NAMES_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	Index in the directory of this entry
FileNameLength	Length of the file name in bytes
FileName	Name of the file

The information returned for this information class is returned longword aligned, and the *FileInformation* buffer itself must be longword aligned.

FileDirectoryInformation - Data type is *FILE_DIRECTORY_INFORMATION*.

```
typedef struct _FILE_DIRECTORY_INFORMATION {
        ULONG NextEntryOffset;
        ULONG FileIndex;
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        LARGE_INTEGER EndOfFile;
        LARGE_INTEGER AllocationSize;
        ULONG FileAttributes;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_DIRECTORY_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The file index of this file in the directory
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Allocated size of all data streams in the file, in bytes
FileAttributes	Attributes of the file
FileNameLength	Length of the name of the file
FileName	Name of the file

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

FileFullDirectoryInformation - Data type is *FILE_FULL_DIR_INFORMATION*.

```
typedef struct _FILE_FULL_DIR_INFORMATION {
        ULONG NextEntryOffset;
        ULONG FileIndex;
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER  LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        LARGE_INTEGER EndOfFile;
        LARGE_INTEGER AllocationSize;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        ULONG FileAttributes;
        ULONG FileNameLength;
        ULONG EaSize;
        WCHAR FileName[];
} FILE_FULL_DIR_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The file index of this file in the directory
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Allocated size of all data streams in the file, in bytes
FileAttributes	Attributes of the file
FileNameLength	Length of the name of the file
EaSize	Size of the EA's associated with the file
FileName	Name of the file

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

FileBothDirectoryInformation - Data type is *FILE_BOTH_DIR_INFORMATION*.

```
typedef struct _FILE_BOTH_DIR_INFORMATION {
        ULONG NextEntryOffset;
        ULONG FileIndex;
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        LARGE_INTEGER EndOfFile;
        LARGE_INTEGER AllocationSize;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        ULONG FileAttributes;
        ULONG FileNameLength;
        ULONG EaSize;
        CCHAR ShortNameLength;
        WCHAR ShortName[12];
        WCHAR FileName[];
} FILE_BOTH_DIR_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The file index of this file in the directory
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Allocated size of all data streams in the file, in bytes
FileAttributes	Attributes of the file
FileNameLength	Length of the name of the file
EaSize	Size of the EA's associated with the file
ShortNameLength	Length of the 8.3 name of the file
ShortName	8.3 name of the file
FileName	Name of the file

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

FileOleDirectoryInformation - Data type is *FILE_OLE_DIR_INFORMATION.*

```
typedef struct _FILE_OLE_DIR_INFORMATION {
        ULONG NextEntryOffset;
        ULONG FileIndex;
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        LARGE_INTEGER EndOfFile;
        LARGE_INTEGER AllocationSize;
        ULONG FileAttributes;
        ULONG FileNameLength;
        FILE_STORAGE_TYPE StorageType;
        GUID OleClassId;
        ULONG OleStateBits;
        BOOLEAN IsExplorable;
        BOOLEAN HasExplorableChildren;
        BOOLEAN ApplicationIsExplorable;
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
        WCHAR FileName[];
} FILE_OLE_DIR_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The index of this file on the volume
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
EndOfFile	Offset to first free byte in the file
AllocationSize	Total allocation size of file, including children
FileAttributes	Attributes of the file
FileNameLength	Length of the name of the file
StorageType	Storage type of the file
OleClassId	OLE class ID
OleStateBits	OLE state bits
IsExplorable	Indicates whether or not object is explorable
HasExplorableChildren	Indicates whether or not object has explorable children
ApplicationHasExplorableChildren	Application-maintained version of above flag
ContentIndexDisable	Indicates whether content indexing has been disabled
InheritContentIndexDisable	Indicates whether content indexing disable is inheritable
FileName	Name of the entry

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

FILE_LIST_DIRECTORY access to the directory is required in order to obtain the above information about files in the specified directory.

As in OS/2 today, users should not depend on any preconceived ideas about the length of file names in **Windows NT**. Because the system supports multiple file system types and will support more in the future, it is difficult to tell just what form file names may take. However, this service guarantees that for **Windows NT V3.1**, a buffer that is large enough to contain at least one *FILE_BOTH_DIR_INFORMATION* structure and has 256 Unicode characters for a file name will be large enough to receive at least one directory entry of any size.

Likewise, a buffer that is large enough to contain at least one name should be at least 256 Unicode characters for the file name itself, plus the size of the remainder of the structure.

Notice that it is legal for the caller to specify the *RestartScan* parameter on a subsequent call to the **NtQueryDirectoryFile** service to have the service restart from the beginning of the directory listing. This causes the scan of the directory to be restarted from the beginning of the list. Notice also that since the file handle may be shared between separate threads within a process, or in threads across processes when the handle is inherited, not all of the directory entries may necessarily be seen by a single thread. That is, the context being maintained to determine which entry should be returned is common among the threads. Therefore, if one thread obtains a directory entry, then the next thread to ask for an entry will obtain the next entry, not the same entry as the first thread.

Once the directory operation has completed, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.3.2 Enumerating Files in an Ole Directory File

The files within an Ole directory file can be enumerate using the **NtQueryOleDirectoryFile** service:

NTSTATUS
NtQueryOleDirectoryFile(
  IN HANDLE *FileHandle*,
  IN HANDLE *Event* **OPTIONAL**,
  IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL**,
  IN PVOID *ApcContext* **OPTIONAL**,
  OUT PIO_STATUS_BLOCK *IoStatusBlock*,
  OUT PVOID *FileInformation*,
  IN ULONG *Length*,
  IN FILE_INFORMATION_CLASS *FileInformationClass*,
  IN BOOLEAN *ReturnSingleEntry*,
  IN PUNICODE_STRING *FileName* **OPTIONAL**,
  IN BOOLEAN *RestartScan*
  );

Parameters:

FileHandle - A file handle to an open container about which information is to be returned.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually written to the specified *Buffer* is stored in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FileInformation - A pointer to a buffer to receive information about the OLE embeddings and streams in the container. The contents of this buffer are defined by the *FileInformationClass* parameter below.

Length - The length of the specified buffer in bytes.

FileInformationClass - Specifies the type of information that is returned in the *FileInformation* buffer. The type of information in the buffer is defined by the following type codes.

FileInformationClass Values

FileDirectoryInformation - Specifies that basic information about the OLE embeddings and streams is written to the *FileInformation* buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileOleDirectoryInformation - Specifies that comprehensive OLE information about the OLE embeddings and streams is written to the *FileInformation* buffer.

ReturnSingleEntry - A BOOLEAN value that, if TRUE, indicates that only a single entry should be returned.

FileName - An optional name within the specified container. This parameter may only be specified on the first call to the service. It selects the embeddings and streams in the container that the query calls return. The specification may contain wildcard characters.

RestartScan - A BOOLEAN value that, if TRUE, indicates that the scan should be restarted from the beginning. This causes the directory operation to restart the scan from the beginning of the container.

The **NtQueryOleDirectoryFile** function operates on a container specified by the *FileHandle* parameter. The service returns information about OLE embeddings and streams in the specified container. The *ReturnSingleEntry* parameter specifies that only a single entry should be returned rather than filling the buffer. The actual number of files whose information is returned, is the smallest of the following:

o - One entry, if the *ReturnSingleEntry* parameter is TRUE.

o - The number of entries whose information fits into the specified buffer.

o - The number of entries that exist in the container according to the wildcard specification. This defaults to all of the entries in the container.

If information about multiple entries is returned, then each entry in the buffer will be aligned on a longword or quadword boundary, depending on the type of information being returned. Each type of information class returned begins with the byte offset required to find the next entry in the buffer. If this value is zero, then there are no more entries following the current entry. Note that there are no entries in the buffer only if the service completes with an error.

The normal operation of this service is to return all of the entries in the container. A wildcard specification may be supplied the first time the service is called to select a subset of the entries in the container. This is done by supplying a wildcard specification in the *FileName* parameter the first time the service is invoked once the container has been opened. Once a wildcard pattern has been supplied, all subsequent **NtQueryOleDirectoryFile** calls using the same handle operate only on those entries which match the pattern. That is, restarting the listing will return the first entry in the container that matches the pattern.

A wildcard specification may only be supplied the first time that the service is invoked. If no wildcard specification is supplied, the file system assumes all of the entries in the container are selected. Wildcard specifications must be consistent with those used in **OS/2 V2.0**.

Likewise, the *FileInformationClass* parameter specified the first time indicates the type of information about the entries in the container that is to be returned. Once an information class is established, it may not be changed in subsequent calls to the service. That is, all subsequent calls must pass the same information class as the first call to the service for a given handle.

The information that is returned in the buffer is defined by the following type codes and structures.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileNamesInformation Format by File Information Class

FileDirectoryInformation - Data type is *FILE_DIRECTORY_INFORMATION*.

```
typedef struct _FILE_DIRECTORY_INFORMATION {
        ULONG NextEntryOffset;
        ULONG FileIndex;
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        LARGE_INTEGER EndOfFile;
        LARGE_INTEGER AllocationSize;
        ULONG FileAttributes;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_DIRECTORY_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The index of this entry in the container
CreationTime	Date/time that the entry was created
LastAccessTime	Date/time that the entry was last accessed
LastWriteTime	Date/time that the entry was last written
ChangeTime	Date/time that the entry was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Total allocated size of the OLE embedding or stream in bytes
FileAttributes	Attributes of the OLE embedding or stream
FileNameLength	Length of the name of the entry
FileName	Name of the entry

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags if the object is an embedding. Otherwise, the file attributes field will be zero.

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileOleDirectoryInformation - Data type is *FILE_OLE_DIR_INFORMATION.*

```
typedef struct _FILE_OLE_DIR_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    FILE_STORAGE_TYPE StorageType;
    GUID OleClassId;
    ULONG OleStateBits;
    BOOLEAN IsExplorable;
    BOOLEAN HasExplorableChildren;
    BOOLEAN ApplicationIsExplorable;
    BOOLEAN ApplicationHasExplorableChildren;
    BOOLEAN ContentIndexDisable;
    BOOLEAN InheritContentIndexDisable;
    WCHAR FileName[];
} FILE_OLE_DIR_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The index of this object on the volume
CreationTime	Date/time that the entry was created
LastAccessTime	Date/time that the entry was last accessed
LastWriteTime	Date/time that the entry was last written
ChangeTime	Date/time that the entry was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Allocated size of the OLE embedding or stream in bytes
FileAttributes	Attributes of the OLE embedding or stream
FileNameLength	Length of the name of the entry
StorageType	Storage type of the entry
OleClassId	OLE class ID
OleStateBits	OLE state bits
IsExplorable	Indicates whether or not object is explorable
HasExplorableChildren	Indicates whether or not object has explorable children
ApplicationHasExplorableChildren	Application-maintained version of above flag
ContentIndexDisable	Indicates whether content indexing has been disabled
InheritContentIndexDisable	Indicates whether CI disable should be inherited
FileName	Name of the entry

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The file attributes field can be a combination of the following flags if the object is an embedding. Otherwise it will be zero.

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

FILE_LIST_DIRECTORY access to the container is required in order to obtain the above information about OLE embeddings and streams in the specified container.

In the case of the **NtQueryOleDirectoryFile**, users can depend on the maximum length of a file name being 31 Unicode characters, because that is the maximum length defined by OLE. Therefore, the name of any stream, property set, embedding, etc., is guaranteed to be a maximum of 31 Unicode characters because this API only operates on OLE objects.

Likewise, a buffer that is large enough to contain at least one name should be at least 31 Unicode characters for the file name itself, plus the size of the remainder of the structure.

Notice that it is legal for the caller to specify the *RestartScan* parameter on a subsequent call to the **NtQueryOleDirectoryFile** service to have the service restart from the beginning of the listing. This causes the scan of the container to be restarted from the beginning of the list. Notice also that since the file handle may be shared between separate threads within a process, or in threads across processes when the handle is inherited, not all of the entries may necessarily be seen by a single thread. That is, the context being maintained to determine which entry should be returned is common among the threads. Therefore, if one thread obtains an entry, then the next thread to ask for an entry will obtain the next entry, not the same entry as the first thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Once the operation has completed, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.3.3 Monitoring Directory Modifications

Directory modifications can be monitored using the **NtNotifyChangeDirectoryFile** service:

NTSTATUS
NtNotifyChangeDirectoryFile(
 IN HANDLE *FileHandle,*
 IN HANDLE *Event* **OPTIONAL,**
 IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
 IN PVOID *ApcContext* **OPTIONAL,**
 OUT PIO_STATUS_BLOCK *IoStatusBlock,*
 OUT PVOID *Buffer,*
 IN ULONG *Length,*
 IN ULONG *CompletionFilter,*
 IN BOOLEAN *WatchTree*
 );

Parameters:

FileHandle - A handle to an open directory file.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A variable to receive the name(s) of the file(s) that changed in the specified target directory.

Length - Specifies the length of the *Buffer*.

CompletionFilter - Specifies a set of flags that indicate the types of operations on the directory or files in the directory that cause the I/O request to complete. The following are the valid flags for this parameter:

CompletionFilter Flags

FILE_NOTIFY_CHANGE_FILE_NAME - Specifies that the I/O operation should be completed if a file is added, deleted, or renamed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_NOTIFY_CHANGE_DIR_NAME - Specifies that the I/O operation should be completed if a subdirectory is added, deleted, or renamed.

FILE_NOTIFY_CHANGE_NAME - Specifies that the I/O operation should be completed if a file or a subdirectory is added, deleted, or renamed.

FILE_NOTIFY_CHANGE_ATTRIBUTES - Specifies that the I/O operation should be completed if the attributes of a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_SIZE - Specifies that the I/O operation should be completed if the allocation size or end of file for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_LAST_WRITE - Specifies that the I/O operation should be completed if the last write date/time for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_LAST_ACCESS - Specifies that the I/O operation should be completed if the last access date/time for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_CREATION - Specifies that the I/O operation should be completed if the creation date/time for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_EA - Specifies that the I/O operation should be completed if the EAs for a file or subdirectory are changed.

FILE_NOTIFY_CHANGE_SECURITY - Specifies that the I/O operation should be completed if the security information for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_STREAM_NAME - Specifies that the I/O operation should be completed if the name of an alternate data stream is changed.

FILE_NOTIFY_CHANGE_STREAM_SIZE - Specifies that the I/O operation should be completed if the size of an alternated data stream is changed.

FILE_NOTIFY_CHANGE_STREAM_WRITE - Specifies that the I/O operation should be completed if an alternate data stream is changed due to a write operation.

WatchTree - A BOOLEAN value that, if TRUE, specifies that all changes to files below the directory should also be reported.

The **NtNotifyChangeDirectoryFile** service notifies the caller when files in the directory or directory tree specified by the *FileHandle* are modified. It also returns the name(s) of the file(s) that changed. All names are specified relative to the directory that the handle represents. The service completes once the directory or directory tree has been modified based on the supplied *CompletionFilter*. The service is a "single shot" and therefore needs to be reinvoked to watch the directory for changes again.

The operation of this service begins by opening a directory for **FILE_LIST_DIRECTORY** access. Once the handle is returned, the **NtNotifyChangeDirectoryFile** service may be invoked to begin watching files and subdirectories in the specified target for changes. The first time the service is invoked, the *Length* parameter supplies the size not only of the user's *Buffer*, but also the buffer that will be used by the file system to store names of files that have changed. Likewise, the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

CompletionFile and *WatchTree* parameters on the first call indicate how notification should operate for all calls using the supplied *FileHandle*. These two parameters are ignored on subsequent calls to the API.

Once a modification is made that should be reported, the system will complete the service. The names of the files that have changed since the last time the service was called will be placed into the caller's output buffer. The *Information* field of the I/O status block indicates the number of bytes that were written to the output buffer. If too many files have changed since the last time the service was called, then zero bytes will be written to the buffer and an alternate status code is returned in the *Status* field of the I/O status block. For the latter case, the application must enumerate the files in the directory or directory tree to note changes.

The format of the data written to the output *Buffer* is defined by the following structure:

```
typedef struct _FILE_NOTIFY_INFORMATION {
        ULONG NextEntryOffset;
        ULONG Action;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_NOTIFY_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
Action	Description of what happened to cause this entry
FileNameLength	Length of the file name that changed
FileName	Name of the file that changed

The value of the *Action* field is defined as one of the following:

Value	Description
FILE_ADDED	The file was added to the directory
FILE_REMOVED	The file was removed from the directory
FILE_MODIFIED	The file was modified
FILE_RENAMED_OLD_NAME	The name of the file that was renamed
FILE_RENAMED_NEW_NAME	The new name of the file that was renamed

When a file is renamed within a single directory, then two entries will be placed into the output buffer: the old name of the file and the new name of the file. If the file is renamed from the directory being monitored to another directory, then only a single entry will be placed into the output buffer with an action type of *Removed*.

This service requires **FILE_LIST_DIRECTORY** access to the directory file that was actually modified. If the operation is watching a directory tree, then the caller must have **FILE_TRAVERSE** access to all intervening directories from the grandparent of the modified file, to the directory specified by the *FileHandle* parameter. It is possible to bypass security checks to all directories if the caller has the **SeNotifyChangePrivilege** privilege.

It should be noted that because of the use of both symbolic and hard links within some file systems, the results of changes to directories within a tree may be unpredictable. That is, some changes may only be seen because the *FileHandle* used refers to a point in the tree through which the change was

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

actually made. Changes made to a point lower in the tree may not be seen because the path used to make the change did not traverse the directory referred to by the *FileHandle*.

It should also be noted that this API may not be implemented by some older network servers. In this case, the API will return a status indicating that it is not implemented. Applications using this API should be prepared to enumerate directories or directory trees in this case.

Once a modification is made to the directory or directory tree, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.4 File Services

This section presents those services that control files and obtain and change information about files.

The APIs that perform these functions are as follows:

NtQueryInformationFile - Obtain information about a file.
NtSetInformationFile - Change information on a file.
NtQueryEaFile - Obtain extended attributes for a file.
NtSetEaFile - Set extended attributes for a file.
NtLockFile - Lock a byte range within a file.
NtUnlockFile - Unlock a byte range within a file.

3.4.1 Obtaining Information about a File

Information about a file may be obtained using the **NtQueryInformationFile** service:

NTSTATUS
NtQueryInformationFile(
  IN HANDLE *FileHandle,*
  OUT PIO_STATUS_BLOCK *IoStatusBlock,*
  OUT PVOID *FileInformation,*
  IN ULONG *Length,*
  IN FILE_INFORMATION_CLASS *FileInformationClass*
  );

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually written to the specified *Buffer* is stored in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FileInformation - A pointer to a buffer to receive the desired information about the file. The contents of this buffer are defined by the *FileInformationClass* parameter described below.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Length - The length of the *FileInformation* buffer in bytes.

FileInformationClass - Specifies the type of information that should be returned about the file. The information returned in the *FileInformation* buffer is defined by the following type codes:

FileInformationClass Values

FileBasicInformation - Returns basic information about the specified file. **FILE_READ_ATTRIBUTES** access to the file is required. Also see the **NtQueryAttributesFile** service description.

FileStandardInformation - Returns standard information about the specified file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileInternalInformation - Returns file system internal information about the file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileEaInformation - Returns the size of the extended attributes structures associated with the file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileAccessInformation - Returns the access that the caller has to the file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileNameInformation - Returns the volume-relative name of the file. No specific access to the file is required; that is, this information is available as long as the file is open.

FilePositionInformation - Returns the current file position for the file. **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file is required.

FileModeInformation - Returns information about how the file is open for the specified file handle. No specific access to the file is required; that is, this information is available as long as the file is open.

FileAlignmentInformation - Returns information about the alignment requirements for buffers being read or written to the file. This is useful when the file has been opened without intermediate buffering enabled. No specific access to the file is required; that is, this information is available as long as the file is open.

FileAllInformation - Returns all of the above information in one structure. **FILE_READ_ATTRIBUTES** access to the file is required to obtain this information. In order for the file position information to be returned, the accessor must have either **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileAlternateNameInformation - Returns the DOS format 8.3 alternate name for the file, if it has one.

FileStreamInformation - Returns the names of the alternate data streams for the file, if any exist.

FileCompressionInformation - Returns the compression information about a file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileOleInformation - Returns the OLE-specific information about a file. **FILE_READ_ATTRIBUTES** access to the file is required to obtain this information.

FileOleAllInformation - Returns the all of the OLE-specific information about a file. **FILE_READ_ATTRIBUTES** access to the file is required to obtain this information. In order for the file position information to be returned, the accessor must have either **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file.

The **NtQueryInformationFile** service returns information about the specified file. The information returned in the buffer is defined by the following type codes and structures. Note that the fields that are not supported for a given device or file system are returned as zero. For example, the FAT file system does not support a creation time, so this field is set to zero.

FileInformation Format by File Information Class

FileBasicInformation - Data type is *FILE_BASIC_INFORMATION*.

```
typedef struct _FILE_BASIC_INFORMATION {
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        ULONG FileAttributes;
} FILE_BASIC_INFORMATION;
```

Field	Description
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
FileAttributes	Attributes of the file

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

Note that the **FILE_ATTRIBUTE_NORMAL** attribute will never be returned in combination with any other attributes, as all other attributes override this attribute. Also see the **NtQueryAttributesFile** service description.

FileStandardInformation - Data type is *FILE_STANDARD_INFORMATION*.

```
typedef struct _FILE_STANDARD_INFORMATION {
        LARGE_INTEGER AllocationSize;
        LARGE_INTEGER EndOfFile;
        DEVICE_TYPE DeviceType;
        ULONG NumberOfLinks;
        BOOLEAN DeletePending;
        BOOLEAN Directory;
} FILE_STANDARD_INFORMATION;
```

Field	Description
AllocationSize	Allocated size of the file in bytes
EndOfFile	Offset to the first free byte in the file
DeviceType	Device type code
NumberOfLinks	Number of hard links to the file
DeletePending	Indicates whether the file is marked for deletion
Directory	Indicates whether the file is a directory

The end of file field specifies the byte offset to the end of the file. Note that because this value is zero-based, it actually refers to the first free byte in the file; that is, it is the offset to the next byte after the last valid byte in the file.

Device types have the following valid values:

FILE_DEVICE_BATTERY
FILE_DEVICE_BEEP
FILE_DEVICE_BUS_EXTENDER
FILE_DEVICE_CD_ROM
FILE_DEVICE_CD_ROM_FILE_SYSTEM
FILE_DEVICE_CONTROLLER
FILE_DEVICE_DATALINK
FILE_DEVICE_DFS
FILE_DEVICE_DISK
FILE_DEVICE_DISK_FILE_SYSTEM
FILE_DEVICE_FILE_SYSTEM
FILE_DEVICE_INPORT_PORT
FILE_DEVICE_KEYBOARD
FILE_DEVICE_MAILSLOT
FILE_DEVICE_MIDI_IN

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_DEVICE_MIDI_OUT
FILE_DEVICE_MOUSE
FILE_DEVICE_MULTI_UNC_PROVIDER
FILE_DEVICE_NAMED_PIPE
FILE_DEVICE_NETWORK
FILE_DEVICE_NETWORK_BROWSER
FILE_DEVICE_NETWORK_FILE_SYSTEM
FILE_DEVICE_NETWORK_REDIRECTOR
FILE_DEVICE_NULL
FILE_DEVICE_PARALLEL_PORT
FILE_DEVICE_PHYSICAL_NETCARD
FILE_DEVICE_PRINTER
FILE_DEVICE_SCANNER
FILE_DEVICE_SCREEN
FILE_DEVICE_SERIAL_MOUSE_PORT
FILE_DEVICE_SERIAL_PORT
FILE_DEVICE_SOUND
FILE_DEVICE_STREAMS
FILE_DEVICE_TAPE
FILE_DEVICE_TAPE_FILE_SYSTEM
FILE_DEVICE_TRANSPORT
FILE_DEVICE_UNKNOWN
FILE_DEVICE_VIDEO
FILE_DEVICE_VIRTUAL_DISK
FILE_DEVICE_WAVE_IN
FILE_DEVICE_WAVE_OUT
FILE_DEVICE_8042_PORT

No specific access is required to obtain this information about the file; that is, this information is obtainable as long as the file is open.

FileInternalInformation - Data type is *FILE_INTERNAL_INFORMATION*.

```
typedef struct _FILE_INTERNAL_INFORMATION {
        LARGE_INTEGER IndexNumber;
} FILE_INTERNAL_INFORMATION;
```

Field	Description
IndexNumber	A file system unique file identifier

No specific access to the file is required to obtain this information about the file; that is, this information is obtainable as long as the file is open.

FileEaInformation - Data type is *FILE_EA_INFORMATION*.

```
typedef struct _FILE_EA_INFORMATION {
        ULONG EaSize;
} FILE_EA_INFORMATION;
```

Field	Description

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

EaSize	Size of file's extended attributes in bytes

No specific access to the file is required to obtain this information about the file; that is, this information is obtainable as long as the file is open.

FileAccessInformation - Data type is *FILE_ACCESS_INFORMATION*.

```
typedef struct _FILE_ACCESS_INFORMATION {
        ACCESS_MASK AccessFlags;
} FILE_ACCESS_INFORMATION;
```

Field	Description
AccessFlags	Access that the caller has to the file

The valid flags that may be set in the **AccessFlags** field are as follows:

SYNCHRONIZE
DELETE
READ_CONTROL
WRITE_DAC
WRITE_OWNER
FILE_READ_EA
FILE_WRITE_EA
FILE_READ_ATTRIBUTES
FILE_WRITE_ATTRIBUTES
FILE_READ_DATA
FILE_WRITE_DATA
FILE_EXECUTE
FILE_APPEND_DATA

If the file is a directory, then the **FILE_READ_DATA** through **FILE_APPEND_DATA** flags are invalid. They are replaced by the following valid values:

FILE_LIST_DIRECTORY
FILE_TRAVERSE

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open.

FileNameInformation - Data type is *FILE_NAME_INFORMATION*.

```
typedef struct _FILE_NAME_INFORMATION {
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_NAME_INFORMATION;
```

Field	Description
FileNameLength	Length of the file name in bytes
FileName	Name of the file

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open.

FilePositionInformation - Data type is *FILE_POSITION_INFORMATION*.

```
typedef struct _FILE_POSITION_INFORMATION {
        LARGE_INTEGER CurrentByteOffset;
} FILE_POSITION_INFORMATION;
```

Field	Description
CurrentByteOffset	Current byte offset within the file

In order for the information to be valid, the file must have been opened or created specifying synchronous I/O.

FILE_READ_DATA or **FILE_WRITE_DATA** access to the file is required to obtain this information about the file.

FileModeInformation - Data type is *FILE_MODE_INFORMATION*.

```
typedef struct _FILE_MODE_INFORMATION {
        ULONG Mode;
} FILE_MODE_INFORMATION;
```

Field	Description
Mode	Current open mode of file handle to the file

The mode flags that may be returned are as follows:

FILE_WRITE_THROUGH
FILE_SEQUENTIAL_ONLY
FILE_NO_INTERMEDIATE_BUFFERING
FILE_SYNCHRONOUS_IO_ALERT
FILE_SYNCHRONOUS_IO_NONALERT
FILE_DELETE_ON_CLOSE

Note that only one of the synchronous I/O flags will be returned.

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open.

FileAlignmentInformation - Data type is *FILE_ALIGNMENT_INFORMATION*.

```
typedef struct _FILE_ALIGNMENT_INFORMATION {
        ULONG AlignmentRequirement;
} FILE_ALIGNMENT_INFORMATION;
```

Field	Description
AlignmentRequirement	Buffer alignment required by device

The value of this field is one of the following:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_BYTE_ALIGNMENT
FILE_WORD_ALIGNMENT
FILE_LONG_ALIGNMENT
FILE_QUAD_ALIGNMENT
FILE_OCTA_ALIGNMENT
FILE_32_BYTE_ALIGNMENT
FILE_64_BYTE_ALIGNMENT
FILE_128_BYTE_ALIGNMENT
FILE_256_BYTE_ALIGNMENT
FILE_512_BYTE_ALIGNMENT

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open.

FileAllInformation - Data type is *FILE_ALL_INFORMATION*.

```
typedef struct _FILE_ALL_INFORMATION {
        FILE_BASIC_INFORMATION BasicInformation;
        FILE_STANDARD_INFORMATION StandardInformation;
        FILE_INTERNAL_INFORMATION InternalInformation;
        FILE_EA_INFORMATION EaInformation;
        FILE_ACCESS_INFORMATION AccessInformation;
        FILE_POSITION_INFORMATION PositionInformation;
        FILE_MODE_INFORMATION ModeInformation;
        FILE_ALIGNMENT_INFORMATION AlignmentInformation;
        FILE_NAME_INFORMATION NameInformation;
} FILE_ALL_INFORMATION;
```

Field	Description
BasicInformation	Basic information
StandardInformation	Standard information
InternalInformation	Internal information
EaInformation	Extended attributes size information
AccessInformation	Access information
PositionInformation	Current position information
ModeInformation	Mode information
AlignmentInformation	Alignment requirement information
NameInformation	File name information

Notice that the position information will be valid only if the file was opened or created using one of the synchronous I/O options.

FILE_READ_ATTRIBUTES access to the file is required to obtain this information. If the file was opened for synchronous I/O, then the position information will only be valid if the accessor has either **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file.

FileAlternateNameInformation - Data type is *FILE_NAME_INFORMATION*.

```
typedef struct _FILE_NAME_INFORMATION {
        ULONG FileNameLength;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
    WCHAR FileName[];
} FILE_NAME_INFORMATION;
```

Field	Description
FileNameLength	Length of the file name in bytes
FileName	Name of the file

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open. Note that some files do not have alternate names.

FileStreamInformation - Data type is *FILE_STREAM_INFORMATION.*

```
typedef struct _FILE_STREAM_INFORMATION {
    ULONG NextEntryOffset;
    ULONG StreamNameLength;
    LARGE_INTEGER StreamSize;
    LARGE_INTEGER StreamAllocationSize;
    WCHAR StreamName;
} FILE_STREAM_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
StreamNameLength	Length of the name of the stream in bytes
StreamSize	Size of the stream
StreamAllocationSize	Allocation size of the stream
StreamName	Name of the stream

No specific access to the file is required to obtain this information about the file; that is, this information is obtainable as long as the file is open.

FileCompressionInformation - Data type is *FILE_COMPRESSION_INFORMATION.*

```
typedef struct _FILE_COMPRESSION_INFORMATION {
    LARGE_INTEGER CompressedFileSize;
    USHORT CompressionFormat;
} FILE_COMPRESSION_INFORMATION;
```

Field	Description
CompressedFileSize	Size of the compressed file in bytes
CompressionFormat	Compression algorithm code

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open. Note that if the file is not compressed, then the *CompressionFormat* field is set to zero.

FileOleInformation - Data type is *FILE_OLE_INFORMATION.*

```
typedef struct _FILE_OLE_INFORMATION {
    FILE_OLE_CLASSID_INFORMATION OleClassIdInformation;
    FILE_OBJECTID_INFORMATION ObjectIdInformation;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        FILE_STORAGE_TYPE StorageType;
        ULONG OleStateBits;
        BOOLEAN ApplicationIsExplorable;
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
} FILE_OLE_INFORMATION;
```

Field	Description
OleClassIdInformation	OLE class ID for the file
ObjectIdInformation	Object ID for the file
OleStateBits	OLE state bits for file
ApplicationIsExplorable	Application-defined notion of explorability
ApplicationHasExplorableChildren	Application-defined notion of children's explorability
ContentIndexDisable	Enable/disable content indexing
InheritContentIndexDisable	Enable/disable content indexing of children

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

FILE_READ_ATTRIBUTES access to the file is required to obtain this information.

FileOleAllInformation - Data type is *FILE_OLE_ALL_INFORMATION*.

```
typedef struct _FILE_OLE_ALL_INFORMATION {
        FILE_BASIC_INFORMATION BasicInformation;
        FILE_STANDARD_INFORMATION StandardInformation;
        FILE_INTERNAL_INFORMATION InternalInformation;
        FILE_EA_INFORMATION EaInformation;
        FILE_ACCESS_INFORMATION AccessInformation;
        FILE_POSITION_INFORMATION PositionInformation;
        FILE_MODE_INFORMATION ModeInformation;
        FILE_ALIGNMENT_INFORMATION AlignmentInformation;
        USN Usn;
        FILE_OLE_CLASSID_INFORMATION OleClassIdInformation;
        FILE_OBJECTID_INFORMATION ObjectIdInformation;
        FILE_STORAGE_TYPE StorageType;
        ULONG OleStateBits;
        ULONG OleId;
        ULONG NumberOfStreamReferences;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        ULONG StreamIndex;
        BOOLEAN IsExplorable;
        BOOLEAN HasExplorableChildren;
        BOOLEAN ApplicationExplorable;
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
        FILE_NAME_INFORMATION NameInformation;
} FILE_OLE_ALL_INFORMATION;
```

Field	Description
BasicInformation	Basic information
StandardInformation	Standard information
InternalInformation	Internal information
EaInformation	Extended attributes size information
AccessInformation	Access information
PositionInformation	Current position information
ModeInformation	Mode information
AlignmentInformation	Alignment requirement information
Usn	Update sequence number
OleClassIdInformation	OLE Class ID for the file
ObjectIdInformation	Object ID for the file
StorageType	Storage type of the file
OleStateBits	OLE state flags
OleId	OLE ID for the file
NumberOfStreamReferences	Reference count for the stream
StreamIndex	Volume index for this stream
IsExplorable	Indicates whether the file is explorable
HasExplorableChildren	Indicates whether the file has explorable children
ApplicationExplorable	Application version of explorable
ApplicationHasExplorableChildren	Application version of explorable children
ContentIndexDisable	Indicates whether content indexing is disabled
InheritContextIndexDisable	Indicates whether CI disable state is inherited
NameInformation	File name information

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

Notice that the position information will be valid only if the file was opened or created using one of the synchronous I/O options.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_READ_ATTRIBUTES access to the file is required to obtain this information. If the file was opened for synchronous I/O, then the position information will only be valid if the accessor has either **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file.

Once the information about the file has been returned, the caller can determine how much information was actually returned by examining the *Information* field of the *IoStatusBlock* variable.

3.4.2 Changing Information about a File

The information about a file may be changed using the **NtSetInformationFile** service:

NTSTATUS
NtSetInformationFile(
  IN HANDLE *FileHandle,*
  OUT PIO_STATUS_BLOCK *IoStatusBlock,*
  IN PVOID *FileInformation,*
  IN ULONG *Length,*
  IN FILE_INFORMATION_CLASS *FileInformationClass*
  );

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

FileInformation - A pointer to a buffer that contains the information about the file to be changed. The contents of this buffer are defined by the *FileInformationClass* parameter described below.

Length - The length of the *FileInformation* buffer in bytes.

FileInformationClass - Specifies the type of information that is contained in the *FileInformation* buffer. The type of information in the buffer is defined by the following type codes.

FileInformationClass Values

FileBasicInformation - Changes the basic information about the specified file. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileRenameInformation - Specifies that the name of the file should be changed to a new name. The caller must be able to remove the directory entry for the file in the current directory and therefore **DELETE** access is required to the file. The caller must also be able to write to the new parent directory. See the notes below for further information.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileLinkInformation - Specifies that a new link be added for the file. The caller must be able to write to the new directory file. See the notes below for further information.

FileDispositionInformation - Specifies that the file should be marked for delete. Once all of the handles to the file have been closed, if the link count for the file is zero, then the file is deleted. Even if the link count is nonzero, at least the directory entry will be deleted. **DELETE** access to the file is required to perform this operation. Also see the **NtDeleteFile** service description.

FilePositionInformation - Specifies a new byte offset as the current position in the file. **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file is required to perform this operation. The file must also have been opened or created using one of the synchronous I/O options.

FileModeInformation - Specifies that a new mode for the specified handle be set. See the notes below for further information.

FileAllocationInformation - Truncates or extends the allocated size of the file. **FILE_WRITE_DATA** access to the file is required to perform this operation. Note that truncating the allocation size of the file may affect the end of file mark for the file as well.

FileEndOfFileInformation - Truncates or extends the amount of valid data in the file by moving the current end of file. **FILE_WRITE_DATA** access to the file is required to perform this operation.

FileCopyOnWrite - Links two streams together until such time as one is written. No specific access right is required to set this information on the file; that is, it is possible to change this information about the file as long as the caller has a valid handle.

FileCompletionInformation - Associates an I/O completion object with the specified file object. This allows synchronization of I/O request completions through the use of an I/O completion object.

FileMoveClusterInformation - Moves data from one file to the end of another file. **FILE_WRITE_DATA** access to the file is required to peform this operation

FileOleClassIdInformation - Sets the OLE class ID for the file. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileOleStateBitsInformation - Sets the OLE state bits for the file. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileApplicationExplorableInformation - Changes the application view of whether or not the object is explorable. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileApplicationExplorableChildrenInformation - Changes the application view of whether or not the object has explorable children. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileObjectIdInformation - Changes the object ID for the file. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileContextIndexInformation - Changes whether or not the file is to be content indexed. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileInheritContentIndexInformation - Changes whether or not the children of this file are to be content indexed. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileOleInformation - Change the OLE information about the file. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

The **NtSetInformationFile** service changes information about a file. The information in the buffer is defined by the following type and structure. Note that the fields that are not supported for a given device or file system are ignored. For example, the FAT file system does not support a creation time, so this field is ignored on an **NtSetInformationFile** service call.

FileInformation Format by File Information Class

FileBasicInformation - Data type is *FILE_BASIC_INFORMATION*.

```
typedef struct _FILE_BASIC_INFORMATION {
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        ULONG FileAttributes;
} FILE_BASIC_INFORMATION;
```

Field	Description
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
FileAttributes	Attributes of the file

All dates and times are specified in the standard **Windows NT** system time format.

The file attributes field can be a combination of the following values:

FILE_ATTRIBUTE_NORMAL

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

Note that the **FILE_ATTRIBUTE_NORMAL** attribute is overridden by all other file attributes flags.

If a field is set to zero, **NtSetInformationFile** does not change the information about the file for that field.

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation.

FileRenameInformation - Data type is *FILE_RENAME_INFORMATION*.

```
typedef struct _FILE_RENAME_INFORMATION {
        BOOLEAN ReplaceIfExists;
        HANDLE RootDirectory;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_RENAME_INFORMATION;
```

Field	Description
ReplaceIfExists	Replace target file if it exists; else fail
RootDirectory	Root directory of target file name
FileNameLength	Length of the file name in bytes
FileName	Name of the file

This operation requires **DELETE** access to the current file so that the directory entry may be removed from the current parent directory. The caller must also have the appropriate access to create the new entry in the new parent directory file.

The file name may be specified in one of three different ways. No wildcards may ever be specified.

o - A simple file name. For this case, the file is simply renamed within the same directory. That is, the name of the file changes but not its location.

o - A fully qualified file name. In this case, the file changes not only its name but its location as well.

o - A relative file name. In this case, the *RootDirectory* field contains a handle to the target directory for the rename operation. The file name itself must be a simple file name.

FileDispositionInformation - Data type is *FILE_DISPOSITION_INFORMATION*.

```
typedef struct _FILE_DISPOSITION_INFORMATION {
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
    BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFORMATION;
```

Field	Description
DeleteFile	Delete the file on close

DELETE access to the file is required to perform this operation.

It should be noted that if the file is deleted, the only legal subsequent operation on the file through the open file handle is to close the file using the NtClose system service.

Also see the **NtDeleteFile** service description.

FileLinkInformation - Data type is *FILE_NAME_INFORMATION*.

```
typedef struct _FILE_NAME_INFORMATION {
    ULONG FileNameLength;
    WCHAR FileName[];
} FILE_NAME_INFORMATION;
```

Field	Description
FileNameLength	Length of the file name in bytes
FileName	Name of the file

No specific access to the file is required to add a link to the file, the file must simply be open. However, the caller must be able to create the new link in the specified target directory.

The file name must be a fully qualified file specification.

FilePositionInformation - Data type is *FILE_POSITION_INFORMATION*.

```
typedef struct _FILE_POSITION_INFORMATION {
    LARGE_INTEGER CurrentByteOffset;
} FILE_POSITION_INFORMATION;
```

Field	Description
CurrentByteOffset	Current byte offset within the file

If the file was opened or created with no intermediate buffering, then the new value of the byte offset must be an integral number of 512 bytes.

FILE_READ_DATA or **FILE_WRITE_DATA** access to the file is required to change this information about the file, and the file must be opened for synchronous I/O.

FileModeInformation - Data type is *FILE_MODE_INFORMATION*.

```
typedef struct _FILE_MODE_INFORMATION {
    ULONG Mode;
} FILE_MODE_INFORMATION;
```

Field	Description

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Mode	Current open mode of file handle to the file

The mode flags that may be changed are as follows:

FILE_WRITE_THROUGH
FILE_SEQUENTIAL_ONLY
FILE_SYNCHRONOUS_IO_ALERT
FILE_SYNCHRONOUS_IO_NONALERT

Note that it is only possible to switch between the two different types of synchronous I/O. It is not possible to either switch to or from synchronous I/O, nor is it possible to specify both types.

If the file has been opened with intermediate buffering disabled, the **FILE_WRITE_THROUGH** flag cannot be turned off. That is, it is forced on by the I/O system. This flag is ignored on a set operation in this case.

Users should be aware that changing this information about the file also changes the access mode for all handles referring to the same file object. That is, all handles referring to the object that are duplicated or inherited are also affected by this access change.

No specific access to the file is required to change this information about the file; that is, this information is available as long as the file is open.

FileAllocationInformation - Data type is *FILE_ALLOCATION_INFORMATION*.

```
typedef struct _FILE_ALLOCATION_INFORMATION {
        LARGE_INTEGER AllocationSize;
} FILE_ALLOCATION_INFORMATION;
```

Field	Description
AllocationSize	The absolute allocation size of the file in bytes

FILE_WRITE_DATA access to the file is required to perform this operation. Setting the allocation size of the file to some number of bytes less than the current end of file mark causes the current end of file mark to be moved to the end of the allocated size of the file.

FileEndOfFileInformation - Data type is *FILE_END_OF_FILE_INFORMATION*.

```
typedef struct _FILE_END_OF_FILE_INFORMATION {
        LARGE_INTEGER EndOfFile;
} FILE_END_OF_FILE_INFORMATION;
```

Field	Description
EndOfFile	The absolute new end of file position

Extending the file beyond the current end of file causes pad bytes of zeroes to be written to the new intermediate bytes.

FILE_WRITE_DATA access to the file is required to perform this operation.

FileCopyOnWrite - Data type is *FILE_COPY_ON_WRITE_INFORMATION*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef struct _FILE_COPY_ON_WRITE_INFORMATION {
        BOOLEAN ReplaceIfExists;
        HANDLE RootDirectory;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_COPY_ON_WRITE_INFORMATION;
```

Field	Description
ReplaceIfExists	Replace the target if it exists, else fail
RootDirectory	Root directory of target file name
FileNameLength	Length of the file name in bytes
FileName	Name of the file

No specific access to the file is required to change this information about the file; that is, it is possible to change this information about the file as long as the caller has a valid handle to the file.

FileCompletionInformation - Data type is *FILE_COMPLETION_INFORMATION.*

```
typedef struct _FILE_COMPLETION_INFORMATION {
        HANDLE Port;
        ULONG Key;
} FILE_COMPLETION_INFORMATION;
```

Field	Description
Port	Handle to the I/O completion object to associate with the file
Key	Caller-defined value to be associated with this completion object

No specific access to the file is required to change this information about the file; that is, it is possible to change this information about the file as long as the caller has a valid handle to the file.

FileMoveClusterInformation - Data type is *FILE_MOVE_CLUSTER_INFORMATION.*

```
typedef struct _FILE_MOVE_CLUSTER_INFORMATION {
        ULONG ClusterCount;
        HANDLE RootDirectory;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_MOVE_CLUSTER_INFORMATION;
```

Field	Description
ClusterCount	Count of clusters to be moved
RootDirectory	Root directory of target file name
FileNameLength	Length of the file name in bytes
FileName	File name of the target

FILE_WRITE_DATA access to the file is required to perform this operation. Setting the move cluster information on a file causes moves *ClusterCount* clusters to the end of the specified target file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileOleClassIdInformation - Data type is *FILE_OLE_CLASS_ID_INFORMATION.*

```
typedef struct _FILE_OLE_CLASS_ID_INFORMATION {
    GUID ClassId;
} FILE_OLE_CLASS_ID_INFORMATION;
```

Field	Description
ClassId	ID of the code that understands this file's format

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the OLE class ID on a file changes the association of application to the file.

FileOleStateBitsInformation - Data type is *FILE_OLE_STATE_BITS_INFORMATION.*

```
typedef struct _FILE_OLE_STATE_BITS_INFORMATION {
    ULONG StateBits;
    ULONG StateBitsMask;
} FILE_OLE_STATE_BITS_INFORMATION;
```

Field	Description
StateBits	OLE state bit information
StateBitsMask	Mask to be applied to state bits

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the OLE state bits on a file changes the value of the file's state bits. The state bits are treated as opaque data to the file system with the exception of the *FILE_ENABLE_DOCFILE_FORMAT* bit which causes a document file to be treated as a single stream rather than as separately addressible streams.

FileApplicationExplorableInformation - Data type is *BOOLEAN.*

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the BOOLEAN flag indicates that the application believes that the object represented by the file handle is explorable.

FileApplicationExplorableChildrenInformation - Data type is *BOOLEAN.*

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the BOOLEAN flag indicates that the application believes that the object represented by the file handle has explorable children.

FileObjectIdInformation - Data type is *FILE_OBJECT_ID_INFORMATION.*

```
typedef struct _FILE_OBJECT_ID_INFORMATION {
    OBJECTID ObjectId;
} FILE_OBJECT_ID_INFORMATION;
```

Field	Description
ObjectId	Object ID for the file

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the Object ID for a file changes the unique ID for the file on the volume.

FileContextIndexInformation - Data type is *BOOLEAN.*

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the BOOLEAN flag disables content indexing for the file.

FileInheritContentIndexInformation - Data type is *BOOLEAN.*

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the BOOLEAN flag disables content indexing for the children of the file.

FileOleInformation - Data type is *FILE_OLE_INFORMATION.*

```
typedef struct _FILE_OLE_INFORMATION {
        FILE_OLE_CLASSID_INFORMATION OleClassIdInformation;
        FILE_OBJECTID_INFORMATION ObjectIdInformation;
        FILE_STORAGE_TYPE StorageType;
        ULONG OleStateBits;
        BOOLEAN ApplicationIsExplorable;
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
} FILE_OLE_INFORMATION;
```

Field	Description
OleClassIdInformation	OLE class ID for the file
ObjectIdInformation	Object ID for the file
OleStateBits	OLE state bits for file
ApplicationIsExplorable	Application-defined notion of explorability
ApplicationHasExplorableChildren	Application-defined notion of children's explorability
ContentIndexDisable	Enable/disable content indexing
InheritContentIndexDisable	Enable/disable content indexing of children

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.4.3 Obtaining Extended Attributes for a File

The extended attributes for a file may be obtained using the **NtQueryEaFile** service:

NTSTATUS
NtQueryEaFile(
    IN HANDLE *FileHandle,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    OUT PVOID *Buffer,*
    IN ULONG *Length,*
    IN BOOLEAN *ReturnSingleEntry,*
    IN PVOID *EaList* **OPTIONAL,**
    IN ULONG *EaListLength,*
    IN PULONG *EaIndex* **OPTIONAL,**
    IN BOOLEAN *RestartScan*
    );

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The length, in bytes, that were written to the *Buffer* is returned in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer to receive extended attributes for the file.

Length - The length of the specified buffer in bytes.

ReturnSingleEntry - A BOOLEAN value that, if TRUE, indicates that only a single entry should be returned.

EaList - An optional list of extended attributes whose name/value pair is returned in the *Buffer*. If this parameter is supplied, only those EAs matching the names of the EAs in the list are returned.

EaListLength - Supplies the length of the *EaList*, if one was specified. If no *EaList* was specified, this parameter should be zero.

EaIndex - An optional index to an EA whose name/value pair is to be returned. The buffer is filled beginning with the EA associated with the index value.

RestartScan - A BOOLEAN value that indicates, if TRUE, that the scan should be restarted from the beginning. This causes the query operation to restart the scan from the beginning of the extended attributes list.

The **NtQueryEaFile** function obtains extended attributes for the file represented by the file handle. Only complete extended attribute name/value pairs are returned. No partial attribute, such as only the name, is ever written into the buffer. The actual number of EAs returned is the smallest of the following:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

o - One entry, if the *ReturnSingleEntry* parameter is TRUE.

o - The number of EAs that fit into the specified buffer.

o - The number of EAs that exist, or the number of EAs that match the list of EAs supplied by the optional *EaList* parameter.

NtQueryEaFile may be invoked multiple times to fill the buffer with EAs from the file. It is possible that the EAs for the file were modified between calls to get more EAs. Due to the sharing semantics defined by OS/2, with which this API is compatible, it is not possible to guarantee that the EAs were not modified.

If the optional *EaList* parameter is specified, then only the information for those EAs specified in the list is returned. Further, if this parameter is specified, then the *EaIndex* parameter is ignored.

The *EaIndex* parameter may optionally be specified to return EAs on the file beginning with an EA other than the first EA in the list.

If multiple EAs are returned, then the structure for each EA in the buffer will be aligned on a longword boundary. Each EA in the list begins with a *NextEntryOffset* field that specifies the number of bytes from the base of the current entry to the start of the next entry. If there are no more entries following the current entry, then the value of this field is zero.

The information that is returned in the *Buffer* is defined by the following structure:

```
typedef struct _FILE_FULL_EA_INFORMATION {
        ULONG NextEntryOffset;
        UCHAR Flags;
        UCHAR EaNameLength;
        USHORT EaValueLength;
        CHAR EaName[];
} FILE_FULL_EA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
Flags	Flags to be associated with the EA
EaNameLength	Length of the EA's name field
EaValueLength	Length of the EA's value field
EaName	The name of the EA

The flags currently defined for EAs are:

FILE_NEED_EA

The value field begins after the end of the *EaName* field of the structure, including a single null character. The null character is not included in the *EaNameLength* field.

The value of the EA can be located then, by adding the length of the EA name to the address of the *EaName* field, and adding one.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The type of the *EaList* parameter is defined by the following structure:

```
typedef struct _FILE_GET_EA_INFORMATION {
        ULONG NextEntryOffset;
        UCHAR EaNameLength;
        CHAR EaName[];
} FILE_GET_EA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
EaNameLength	Length of the EA's name field
EaName	The name of the EA to be retrieved

The *NextEntryOffset* field, like its *FILE_FULL_EA_INFORMATION* counterpart, is the offset in bytes from the current entry in the list to the start of the next entry, if there is one. If there are no more entries in the list, then the value of this field is zero.

The *EaList* parameter defines the list of the EAs whose information is to be returned. This selects a proper subset of the EAs and only those EAs are returned.

FILE_READ_EA access to the file is required in order to obtain information about the extended attributes associated with the file.

If an error, such as an invalid character is found in an EA name field, is encountered, then the *Information* field in the I/O status block contains the byte offset from the base of the *Buffer* to the offending EA entry that caused the failure.

Once extended attributes for the file have been written to the *Buffer*, the *Information* field of the *IoStatusBlock* variable can be examined to determine how many bytes of extended attributes information were actually returned.

3.4.4 Changing Extended Attributes for a File

The extended attributes associated with a file may be changed using the **NtSetEaFile** service:

NTSTATUS
NtSetEaFile(
    IN HANDLE *FileHandle,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    IN PVOID *Buffer,*
    IN ULONG *Length*
    );

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer that contains the extended attributes to be applied to the file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Length - The length of the specified buffer in bytes.

The **NtSetEaFile** service changes the extended attributes on the file using the EAs specified by the *Buffer* parameter.

The information specified by the *Buffer* parameter is defined by the following structure.

```
typedef struct _FILE_FULL_EA_INFORMATION {
        ULONG NextEntryOffset;
        UCHAR Flags;
        UCHAR EaNameLength;
        USHORT EaValueLength;
        CHAR EaName[];
} FILE_FULL_EA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
Flags	Flags to be associated with the EA
EaNameLength	Length of the EA's name field
EaValueLength	Length of the EA's value field
EaName	The name of the EA

The flags currently defined for EAs are:

FILE_NEED_EA

The value field begins after the end of the *EaName* field of the structure, including a single null character. The null character is not included in the *EaNameLength* field.

If multiple EAs are contained in the buffer, then the structure for each entry is longword aligned. The *NextEntryOffset* field contains the byte offset to the start of the next entry in the buffer. If there are no more entries past the current entry, then this field is zero.

EAs are applied to the file such that if the EA does not exist, then it is added. If the EA does exist, it is replaced. An entry whose *EaValueLength* field is zero indicates that the EA whose name matches the entry is to be deleted from the list of EAs on the file.

If an error occurs changing the EAs on the file, then the *Information* field in the I/O status block contains the byte offset from the base of the *Buffer* to the offending EA entry that caused the failure.

FILE_WRITE_EA access to the file is required in order to change the extended attributes associated with the file.

3.4.5 Locking Byte Ranges in Files

A byte range within a file may be locked using the **NtLockFile** service:

NTSTATUS
NtLockFile(
    IN HANDLE *FileHandle,*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN HANDLE *Event* **OPTIONAL,**
IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
IN PVOID *ApcContext* **OPTIONAL,**
OUT PIO_STATUS_BLOCK *IoStatusBlock,*
IN PLARGE_INTEGER *ByteOffset,*
IN PLARGE_INTEGER*Length,*
IN ULONG *Key,*
IN BOOLEAN *FailImmediately,*
IN BOOLEAN *ExclusiveLock*
);

Parameters:

FileHandle - A handle to an open file.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

ByteOffset - Specifies the starting byte offset of the file where the lock should begin.

Length - The length of the byte range to lock, in bytes.

Key - A value to be associated with the lock range for further identification.

FailImmediately - A BOOLEAN value that indicates whether the service will return immediately if the lock cannot be obtained (TRUE), or whether the service will wait indefinitely until the lock is acquired (FALSE).

ExclusiveLock - A BOOLEAN value that indicates the type of lock that is applied to the byte range. If the value is TRUE, then the lock is *exclusive*; otherwise, the lock is *shared*.

The **NtLockFile** service is used to lock the specified byte range for the file. The range locked is for the specified file, and is controlled by the following:

o - the *ByteOffset* of the file

o - the *Length* of the byte range

o - the *Key* value associated with the byte range

o - the invoking process

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Locks are not inherited by child processes when they are created. They are owned by the process that acquired the lock. Locks may be manipulated and "owned" by separate threads within a process as thread-specific locks by specifying non-zero values for the *Key* parameter in each thread.

There are two types of locks on files, shared and exclusive. A shared lock allows read-only access by any process attempting to read the locked range, including the owning process. Shared locks may also overlap. Exclusive locks allow read/write access by only the owning process and by access to any other process. Exclusive locks may not overlap either shared locks or other exclusive locks.

Locks owned by a given process are unlocked once all of the handles to the specified file have been closed by that process. The locks are not released in any particular order.

It is not an error to specify a range that either spans or even begins after the end of the file. These types of locks can be used to synchronize access to the end of the file or for appending data to the file.

FILE_READ_DATA or **FILE_WRITE_DATA** access is required to the file to request a lock.

3.4.6 Unlocking Byte Ranges in Files

A byte range within a file may be unlocked using the **NtUnlockFile** service:

```
NTSTATUS
NtUnlockFile(
        IN HANDLE FileHandle,
        OUT PIO_STATUS_BLOCK IoStatusBlock,
        IN PLARGE_INTEGER ByteOffset,
        IN PLARGE_INTEGER Length,
        IN ULONG Key
        );
```

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

ByteOffset - The byte offset of the file whose corresponding lock is released. This value must exactly match the byte offset of the lock.

Length - The length of the locked byte range that is released. This value must exactly match the length of the lock.

Key - The value associated with the lock range for further identification. This value must exactly match the key of the lock.

The **NtUnlockFile** service is used to unlock the specified byte range for the file. The lock parameters must exactly match those of the acquired lock. If the parameters exactly match those of the locked range, then the lock is released.

Only the process that owns the lock may unlock the byte range.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.5 File System Services

This section presents those services that obtain information about file systems and control them.

The APIs that perform these functions are as follows:

NtQueryVolumeInformationFile - Obtain information about a file system volume.
NtSetVolumeInformationFile - Change information about a file system volume.
NtQueryQuotaInformationFile Obtain quota information about a file system volume.
NtSetQuotaInformationFile - Change quota information about a file system volume.
NtFsControlFile - General file system control interface.

3.5.1 Obtaining Information about a File System Volume

Information about a file system volume may be obtained using the **NtQueryVolumeInformationFile** service:

NTSTATUS
NtQueryVolumeInformationFile(
    IN HANDLE *FileHandle,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    OUT PVOID *FsInformation,*
    IN ULONG *Length,*
    IN FS_INFORMATION_CLASS *FsInformationClass*
    );

Parameters:

FileHandle - A handle to an open file, device, directory, or volume for which volume information is returned.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The length, in bytes, of the data written to the *FsInformation* buffer is returned in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FsInformation - A pointer to a buffer to receive information about the specified volume. The contents of this buffer are defined by the *FsInformationClass* parameter described below.

Length - The length of the *FsInformation* buffer in bytes.

FsInformationClass - Specifies the type of information that should be returned about the volume. The information in the *FsInformation* buffer is defined by the following type codes.

FsInformationClass Values

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileFsVolumeInformation - Returns information about the volume that is currently "mounted" on the specified device. No specific access to the volume is required to obtain this information.

FileFsSizeInformation - Returns information about the size and the free space on the volume. No specific access to the volume is required to obtain this information.

FileFsDeviceInformation - Returns information about the device upon which the volume is actually mounted, or the device to which the handle directly refers. No specific access to the volume is required to obtain this information.

FileFsAttributeInformation - Returns attribute information about the file system responsible for the volume. No specific access to the volume is required to obtain this information.

FileFsControlInformation - Returns file system control information about the volume. No specific access to the volume is require to obtain this information.

The **NtQueryVolumeInformationFile** service returns information about the volume specified by the *FileHandle* parameter. The information returned in the buffer is defined by the following type codes and structures.

FsInformation Format by Fs Information Class

FileFsVolumeInformation - Data type is *FILE_FS_VOLUME_INFORMATION*.

```
typedef struct _FILE_FS_VOLUME_INFORMATION {
        LARGE_INTEGER VolumeCreationTime;
        ULONG VolumeSerialNumber;
        ULONG VolumeLabelLength;
        BOOLEAN SupportsObjects;
        WCHAR VolumeLabel[];
} FILE_FS_VOLUME_INFORMATION;
```

Field	Description
VolumeCreationTime	Date/time the volume was created
VolumeSerialNumber	Serial number of the volume
VolumeLabelLength	Length of the name of the volume
SupportsObjects	File system supports object-oriented file system objects
VolumeLabel	Name of the volume

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

FileFsSizeInformation - Data type is *FILE_FS_SIZE_INFORMATION*.

```
typedef struct _FILE_FS_SIZE_INFORMATION {
        LARGE_INTEGER TotalAllocationUnits;
        LARGE_INTEGER AvailableAllocationUnits;
        ULONG SectorsPerAllocationUnit;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
        ULONG BytesPerSector;
} FILE_FS_SIZE_INFORMATION;
```

Field	Description
TotalAllocationUnits	Total allocation units on volume
AvailableAllocationUnits	Free allocation units on volume
SectorsPerAllocationUnit	Number of sectors in each allocation unit
BytesPerSector	Number of bytes in each sector

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

FileFsDeviceInformation - Data type is *FILE_FS_DEVICE_INFORMATION*.

```
typedef struct _FILE_FS_DEVICE_INFORMATION {
        DEVICE_TYPE DeviceType;
        ULONG Characteristics;
} FILE_FS_DEVICE_INFORMATION;
```

Field	Description
DeviceType	Type of the target device
Characteristics	Characteristcs of the target device

Device types have the following valid values:

FILE_DEVICE_BATTERY
FILE_DEVICE_BEEP
FILE_DEVICE_BUS_EXTENDER
FILE_DEVICE_CD_ROM
FILE_DEVICE_CD_ROM_FILE_SYSTEM
FILE_DEVICE_CONTROLLER
FILE_DEVICE_DATALINK
FILE_DEVICE_DFS
FILE_DEVICE_DISK
FILE_DEVICE_DISK_FILE_SYSTEM
FILE_DEVICE_FILE_SYSTEM
FILE_DEVICE_INPORT_PORT
FILE_DEVICE_KEYBOARD
FILE_DEVICE_MAILSLOT
FILE_DEVICE_MIDI_IN
FILE_DEVICE_MIDI_OUT
FILE_DEVICE_MOUSE
FILE_DEVICE_MULTI_UNC_PROVIDER
FILE_DEVICE_NAMED_PIPE
FILE_DEVICE_NETWORK
FILE_DEVICE_NETWORK_BROWSER
FILE_DEVICE_NETWORK_FILE_SYSTEM
FILE_DEVICE_NETWORK_REDIRECTOR
FILE_DEVICE_NULL
FILE_DEVICE_PARALLEL_PORT

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_DEVICE_PHYSICAL_NETCARD
FILE_DEVICE_PRINTER
FILE_DEVICE_SCANNER
FILE_DEVICE_SCREEN
FILE_DEVICE_SERIAL_MOUSE_PORT
FILE_DEVICE_SERIAL_PORT
FILE_DEVICE_SOUND
FILE_DEVICE_STREAMS
FILE_DEVICE_TAPE
FILE_DEVICE_TAPE_FILE_SYSTEM
FILE_DEVICE_TRANSPORT
FILE_DEVICE_UNKNOWN
FILE_DEVICE_VIDEO
FILE_DEVICE_VIRTUAL_DISK
FILE_DEVICE_WAVE_IN
FILE_DEVICE_WAVE_OUT
FILE_DEVICE_8042_PORT

Device characteristics have the following valid flags:

Flag	Meaning
FILE_REMOVABLE_MEDIA	Device supports removable media
FILE_READ_ONLY_DEVICE	Device is a read-only device
FILE_FLOPPY_DISKETTE	Media in device is a floppy diskette
FILE_WRITE_ONCE_MEDIA	Device supports write once media
FILE_REMOTE_DEVICE	Device is a remote device
FILE_DEVICE_IS_MOUNTED	Device is currently mounted
FILE_VIRTUAL_VOLUME	Device volume is virtual

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

FileFsAttributeInformation - Data type is *FILE_FS_ATTRIBUTE_INFORMATION*.

```
typedef struct _FILE_FS_ATTRIBUTE_INFORMATION {
    ULONG FileSystemAttributes;
    LONG MaximumComponentNameLength;
    ULONG FileSystemNameLength;
    WCHAR FileSystemName;
} FILE_FS_ATTRIBUTE_INFORMATION;
```

Field	Description
FileSystemAttributes	Attributes of the volume's owning file system
MaximumComponentNameLength	Maximum length of each file name component
FileSystemNameLength	The length of the file system's name
FileSystemName	The name of the file system

File system attributes have the following valid flags:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Flag	Meaning
FILE_CASE_SENSITIVE_SEARCH	Supports case sensitive searches
FILE_CASE_PRESERVED_NAMES	Supports preserving name case on disk
FILE_UNICODE_ON_DISK	Stores UNICODE characters on disk
FILE_PERSISTENT_ACLS	Stores ACLs on disk
FILE_FILE_COMPRESSION	Supports file compression
FILE_VOLUME_IS_COMPRESSED	Handle refers to a compressed volume

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

FileFsControlInformation - Data type is *FILE_FS_CONTROL_INFORMATION {*

```
typedef struct _FILE_FS_CONTROL_INFORMATION {
        LARGE_INTEGER FreeSpaceStartFiltering;
        LARGE_INTEGER FreeSpaceThreshold;
        LARGE_INTEGER FreeSpaceStopFiltering;
        LARGE_INTEGER DefaultQuotaThreshold;
        LARGE_INTEGER DefaultQuotaLimit;
        LARGE_INTEGER DeletionLogSizeLimit;
        ULONG FileSystemControlFlags;
} FILE_FS_CONTROL_INFORMATION;
```

Field	Description
FreeSpaceStartFiltering	Amount of space required to begin content indexing
FreeSpaceThreshold	Amount of space remaining to generate popup
FreeSpaceStopFiltering	Amount of space remaining to stop content indexing
DefaultQuotaThreshold	Default quota threshold for volume
DefaultQuotaLimit	Default quota limit for volume
DeletionLogSizeLimit	Size of deletion file log
FileSystemControlFlags	Flags to control this volume

File system control flags consist of the following valid flag values:

Flag	Meaning
FILE_VC_QUOTA_NONE	No quota information maintained
FILE_VC_QUOTA_TRACK	Quotas are being tracked on volume
FILE_VC_QUOTA_ENFORCE	Quotas are being enforced on volume
FILE_VC_QUOTAS_INCOMPLETE	Volume quotas are incomplete
FILE_VC_CONTENT_INDEX_DISABLED	Content indexing disabled
FILE_VC_LOG_QUOTA_THRESHOLD	Log quota threshold reached event
FILE_VC_LOG_QUOTA_LIMIT	Log quota limit reached event
FILE_VC_LOG_VOLUME_THRESHOLD	Log volume free space threshold event
FILE_VC_LOG_VOLUME_LIMIT	Log volume free space limit event

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Once the information about the volume has been returned, the *Information* field of the *IoStatusBlock* variable can be examined to determine the number of bytes of volume information actually written to the *FsInformation* buffer.

3.5.2 Changing Information about a File System Volume

Information about a file system volume may be changed using the **NtSetVolumeInformationFile** service:

NTSTATUS
NtSetVolumeInformationFile(
    IN HANDLE *FileHandle,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    IN PVOID *FsInformation,*
    IN ULONG *Length,*
    IN FS_INFORMATION_CLASS *FsInformationClass*
    );

Parameters:

FileHandle - A handle to an open volume for which information is changed.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

FsInformation - A pointer to a buffer that contains the information about the file system to be changed. The contents of this buffer are defined by the *FsInformationClass* parameter described below.

Length - The length of the *FsInformation* buffer in bytes.

FsInformationClass - Specifies the type of information that should be changed about the file system. The information in the *FsInformation* buffer is defined by the following type codes.

FsInformationClass Values

FileFsLabelInformation - Changes the volume label on the volume that is currently "mounted" on the specified device. **FILE_WRITE_DATA** access to the device or volume is required.

FileFsControlInformation - Changes the file system control information for the volume that is currently “mounted” on the specified device. **FILE_WRITE_DATA** access to the device or volume is required.

The **NtSetVolumeInformationFile** service changes information about the volume "mounted" on the device specified by the *FileHandle* parameter. The information to be changed is in the *FsInformation* buffer. Its contents are defined by the following type codes and structures.

FsInformation Format by Fs Information Class

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FileFsLabelInformation - Data type is *FILE_FS_LABEL_INFORMATION*.

```
typedef struct _FILE_FS_LABEL_INFORMATION {
        ULONG VolumeLabelLength;
        WCHAR VolumeLabel[];
} FILE_FS_LABEL_INFORMATION;
```

Field	Description
VolumeLabelLength	Length of the name of the volume
VolumeLabel	Name of the volume

FILE_WRITE_DATA access to the device or volume is required to change this information.

FileFsControlInformation - Data type is *FILE_FS_CONTROL_INFORMATION {*

```
typedef struct _FILE_FS_CONTROL_INFORMATION {
        LARGE_INTEGER FreeSpaceStartFiltering;
        LARGE_INTEGER FreeSpaceThreshold;
        LARGE_INTEGER FreeSpaceStopFiltering;
        LARGE_INTEGER DefaultQuotaThreshold;
        LARGE_INTEGER DefaultQuotaLimit;
        LARGE_INTEGER DeletionLogSizeLimit;
        ULONG FileSystemControlFlags;
} FILE_FS_CONTROL_INFORMATION;
```

Field	Description
FreeSpaceStartFiltering	Amount of space required to begin content indexing
FreeSpaceThreshold	Amount of space remaining to generate popup
FreeSpaceStopFiltering	Amount of space remaining to stop content indexing
DefaultQuotaThreshold	Default quota threshold for volume
DefaultQuotaLimit	Default quota limit for volume
DeletionLogSizeLimit	Size of deletion file log
FileSystemControlFlags	Flags to control this volume

File system control flags consist of the following valid flag values:

Flag	Meaning
FILE_VC_QUOTA_NONE	No quota information maintained
FILE_VC_QUOTA_TRACK	Quotas are being tracked on volume
FILE_VC_QUOTA_ENFORCE	Quotas are being enforced on volume
FILE_VC_QUOTAS_INCOMPLETE	Volume quotas are incomplete
FILE_VC_CONTENT_INDEX_DISABLED	Content indexing disabled
FILE_VC_LOG_QUOTA_THRESHOLD	Log quota threshold reached event
FILE_VC_LOG_QUOTA_LIMIT	Log quota limit reached event
FILE_VC_LOG_VOLUME_THRESHOLD	Log volume free space threshold event
FILE_VC_LOG_VOLUME_LIMIT	Log volume free space limit event

FILE_WRITE_DATA access to the volume is required in order to change the file system control information.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.5.3 Obtaining Quota Information about a File System Volume

Quota information about a file system volume may be obtained using the **NtQueryQuotaInformationFile** service:

NTSTATUS
NtQueryQuotaInformationFile(
    IN HANDLE *FileHandle*,
    OUT PIO_STATUS_BLOCK *IoStatusBlock*,
    OUT PVOID *Buffer*,
    IN ULONG *Length*,
    IN BOOLEAN *ReturnSingleEntry*,
    IN PVOID *SidList* **OPTIONAL**,
    IN ULONG *SidListLength*,
    IN PSID *StartSid* **OPTIONAL**,
    IN BOOLEAN *RestartScan*
    );

Parameters:

FileHandle - An open handle to an open file, directory, device, or volume whose quota information is to be returned.

IoStatusBlock - A variable to receive the final completion status and information about the operation. Service calls that return information, return the length of the data written to the output buffer in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer to receive the requested quota information about the specified volume.

Length - Specifies the length of the *Buffer* parameter in bytes.

ReturnSingleEntry - A BOOLEAN value that, if TRUE, indicates that only a single quota entry should be returned.

SidList - An optional list of entries whose quota entries are returned in the *Buffer*. If this parameter is supplied, only those quota entries matching the SIDs in the list are returned.

SidListLength - Supplies the length of the *SidList*, if one was specified. If no *SidList* was specified, this parameter should be zero.

StartSid - An optional SID that specifies a quota entry to be rewound to during a *RestartScan* operation. The first quota entry returned is the entry following the entry specified by the SID.

RestartScan - A BOOLEAN value that, if TRUE, indicates that the scan should be restarted from the beginning, or alternately from the entry following the *StartSid* entry. This causes the query to restart the scan from the beginning or from the entry following the quota entry for the specified SID.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **NtQueryQuotaInformationFile** function obtains quota entry information for the volume represented by the file handle. Only complete quota entries are returned. The actual number of quota entries returned is the smallest of the following:

o - One entry, if the *ReturnSingleEntry* parameter is TRUE.

o - One entry, if the only entry visible is the entry for the current thread's SID.

o - The number of quota entries that fit into the specified buffer.

o - The number of quota entries that exist, or the number of entries that match the list of entries supplied by the optional *SidList* parameter.

NtQueryQuotaInformationFile may be invoked multiple times to fill the buffer with quota entries for the volume. It is possible that the quota entries for the volume were modified between calls to get more entries, unless the volume is locked.

If the optional *SidList* parameter is specified, then only the quota information for those SIDs specified in the list is returned. Specifying a SID which has no corresponding quota information on the volume causes an entry to be returned with all zeroes for the quota fields. Futher, if this parameter is specified, then the *StartSid* parameter is ignored. Finally, if a *SidList* is specified, the output buffer will be filled with as many matching entries as possible. If they do not fit, then the caller should invoke the service again, changing the start of the list to the point where the last service left off.

For example, if the caller passed in a *SidList* with entries for SIDs A, B, C, D, and E, and the output buffer was only large enough for the file system to return entries for SIDs A, B, and C, then the caller should invoke the service again specifying SIDs D and E. Because the list is self-describing, this can be easily accomplished by simply changing the starting pointer and adjusting the *SidListLength* parameter.

The *StartSid* parameter may optionally be specified to return quota entries for the volume beginning with an entry other than the first quota entry. If a *StartSid* is specified, and the *RestartScan* parameter is specified, then the quota entries returned will be start with the quota entry for the entry after the one selected by the *StartSid* parameter.

If multiple quota entries are returned, then the structure for each entry in the buffer will be aligned on a longword boundary. Each entry in the list begins with a *NextEntryOffset* field that specifies the number of bytes from the base of the current entry to the start of the next entry. If there are no more entries following the current entry, then the value of this field is zero.

The format of the *SidList* information buffer is defined by the following structure:

```
typedef struct _FILE_GET_QUOTA_INFORMATION {
        ULONG NextEntryOffset;
        ULONG SidLength;
        SID Sid;
} FILE_GET_QUOTA_INFORMATION, *PFILE_GET_QUOTA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SidLength Length, in bytes, of the SID
Sid SID of entry to be returned

No special access to the volume is required in order to obtain quota information about the volume. The *FileHandle* may refer to either the volume, or a file or directory anywhere on the volume to which the caller has some access.

Once quota entries for the volume have been written to the *Buffer*, the *Information* field of the *IoStatusBlock* variable can be examined to determine how many bytes of quota information were actually returned.

3.5.4 Changing Quota Information about a File System Volume

Quota information about a file system volume may be changed using the **NtSetQuotaInformationFile** service:

NTSTATUS
NtSetQuotaInformationFile(
    IN HANDLE *FileHandle*,
    OUT PIO_STATUS_BLOCK *IoStatusBlock*,
    IN PVOID *Buffer*,
    IN ULONG *Length*
    );

Parameters:

FileHandle - A handle to a volume whose quota entries are to be changed.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A opinter to a buffer that contains the quota entry information to be applied to the volume.

Length - The length of the specified buffer in bytes.

The **NtSetQuotaInformationFile** service changes the quota information on a volume using the quota entries specified by the *Buffer* parameter.

The information specified by the *Buffer* parameter is defined by the following structure:

```
typedef struct _FILE_QUOTA_INFORMATION {
    ULONG NextEntryOffset;
    ULONG SidLength;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER QuotaUsed;
    LARGE_INTEGER QuotaThreshold;
    LARGE_INTEGER QuotaLimit;
    SID Sid;
} FILE_QUOTA_INFORMATION, *PFILE_QUOTA_INFORMATION;
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
SidLength	Length, in bytes, of the SID
ChangeTime	Time that the quota entry was last changed
QuotaUsed	Amount of disk space used
QuotaThreshold	Amount of disk space useable without incurring an event
QuotaLimit	Amount of disk space permitted to be used
Sid	SID of this quota entry

If multiple quota entries are contained in the buffer, then the structure for each entry is longword aligned. The *NextEntryOffset* field contains the byte offset to the start of the next entry in the buffer. If there are no more entries past the current entry, then this field is zero.

If an error occurs changing the quotas on the volume, then the *Information* field in the I/O status block contains the byte offset from the vase of the *Buffer* to the offending quota entry that caused the failure.

FILE_WRITE_DATA access to the volume is required in order to change the quota information associated with the volume.

3.5.5 Controlling File Systems

Information may be passed between applications and file systems using the **NtFsControlFile** service:

```
NTSTATUS
NtFsControlFile(
    IN HANDLE FileHandle,
    IN HANDLE Event OPTIONAL,
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
    IN PVOID ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN ULONG FsControlCode,
    IN PVOID InputBuffer OPTIONAL,
    IN ULONG InputBufferLength,
    OUT PVOID OutputBuffer OPTIONAL,
    IN ULONG OutputBufferLength
    );
```

Parameters:

FileHandle - An open file handle to the file or device to whose file system the control information should be given.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IoStatusBlock - A variable to receive the final completion status and information about the operation. Service calls that return information, return the length of the data written to the output buffer in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FsControlCode - A code that indicates which file system control function is to be executed.

InputBuffer - An optional pointer to a buffer that contains the information to be given to the target file system. This information is file-system-specific.

InputBufferLength - The length of the *InputBuffer* in bytes. If the buffer is not supplied, then this value is ignored.

OutputBuffer - An optional pointer to a buffer that is to receive the file-system-dependent return information from the target file system.

OutputBufferLength - The length of the *OutputBuffer* in bytes. If the buffer is not supplied, then this value is ignored.

The **NtFsControlFile** service is a file-system-dependent interface that extends the control that applications have over various components within the system. This API provides a consistent view of the input and output data to the system while still providing the application and file system drivers a file-system-dependent method of specifying a communications interface.

The type of access that the caller needs to the file is dependent on the actual operation being performed.

3.6 Miscellaneous Services

This section presents those service that provide miscellaneous functionality for files and devices.

The APIs that perform these functions are as follows:

NtFlushBuffersFile - Flushes all buffered and cached data out to the file.
NtCancelIoFile - Cancels all I/O operations on a file.
NtDeviceIoControlFile - Miscellaneous device control.

3.6.1 Flushing File Buffers

Buffered data may be flushed out to the file using the **NtFlushBuffersFile** service:

NTSTATUS
NtFlushBuffersFile(
  IN HANDLE *FileHandle,*
  OUT PIO_STATUS_BLOCK *IoStatusBlock*
  );

Parameters:

FileHandle - An open file handle to a file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

The **NtFlushBuffersFile** service causes all buffered data to be written to the file.

FILE_WRITE_DATA or **FILE_APPEND_DATA** access to the file is required to perform this service.

3.6.2 Canceling Pending I/O on a File

Pending I/O operations on a file may be canceled using the **NtCancelIoFile** service:

NTSTATUS
NtCancelIoFile(
IN HANDLE *FileHandle,*
OUT PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

FileHandle - An open file handle to a file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

The **NtCancelIoFile** service causes all pending I/O for the specified file to be marked as canceled. Most types of operations can be canceled immediately, while others may continue toward completion before they are actually canceled. For example, once a DMA disk drive has begun a transfer, the operation cannot be canceled by a device driver, but to the caller it will appear as if the operation had effectively been canceled.

Only those pending operations that were issued by the current thread using the specified handle are canceled. Any operations issued for the file by any other thread or any other process continues normally.

No specific access to the file is required in order to use this service since the caller is only canceling those operations that he requested in the first place.

All pending I/O operations complete with a status that indicates that the operation was canceled.

3.6.3 Miscellaneous I/O Control

Various operations may be performed on files to control the file, or the device associated with the file, using the **NtDeviceIoControlFile** service:

NTSTATUS
NtDeviceIoControlFile(
IN HANDLE *FileHandle,*
IN HANDLE *Event* **OPTIONAL,**
IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
IN PVOID *ApcContext* **OPTIONAL,**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
OUT PIO_STATUS_BLOCK IoStatusBlock,
IN ULONG IoControlCode,
IN PVOID InputBuffer OPTIONAL,
IN ULONG InputBufferLength,
OUT PVOID OutputBuffer OPTIONAL,
IN ULONG OutputBufferLength
);
```

Parameters:

FileHandle - An open file handle to the file or device to which the control information should be given.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. Service calls that return information, return the length of the data written to the output buffer in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

IoControlCode - A code that indicates which device I/O control function is to be executed.

InputBuffer - An optional pointer to a buffer that contains the information to be given to the target device. This information is device-dependent.

InputBufferLength - The length of the *InputBuffer* in bytes. If the buffer is not supplied, then this value is ignored.

OutputBuffer - An optional pointer to a buffer that is to receive the device-dependent return information from the target device.

OutputBufferLength - The length of the *OutputBuffer* in bytes. If the buffer is not supplied, then this value is ignored.

The **NtDeviceIoControlFile** service is a device-dependent interface that extends the control that applications have over various devices within the system. This API provides a consistent view of the input and output data to the system while still providing the application and the driver a device-dependent method of specifying a communications interface.

The type of access that the caller needs to the file is dependent on the actual operation being performed.

Once the service has completed, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.6.4 Deleting a File

A file can be deleted using the **NtDeleteFile** service:

NTSTATUS
NtDeleteFile(
  IN POBJECT_ATTRIBUTES *ObjectAttributes*
  );

Parameters:

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, and a set of file object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the file to be deleted. This file specification must be a fully qualified file specification or the name of a device, unless it is a file relative to the directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the file specified by the *ObjectName* field is a file specification relative to the directory file supplied by this handle.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

The **NtDeleteFile** service allows the caller to delete a file. **DELETE** access to the target file is required. This service is equivalent to calling **NtOpenFile, NtSetInformationFile** with a file information class of *FileDispositionInformation*, and **NtClose**. However, this service is faster because less ring transitions are made.

3.6.5 Querying the Attributes of a File

The attributes of a file can be queried using the **NtQueryAttributesFile** service:

NTSTATUS
NtQueryAttributesFile(
  IN POBJECT_ATTRIBUTES *ObjectAttributes*,
  OUT PFILE_BASIC_INFORMATION *FileInformation*
  );

Parameters:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, and a set of file object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the file to be queried. This file specification must be a fully qualified file specification or the name of a device, unless it is a file relative to the directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the file specified by the *ObjectName* field is a file specification relative to the directory file supplied by this handle.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

FileInformation - A variable to receive the basic information about the file.

The **NtQueryAttributesFile** service allows the caller to query the basic information about a file. **FILE_READ_ATTRIBUTES** access to the target file is required. This service is equivalent to calling **NtOpenFile, NtQueryInformationFile** with a file information class of *FileBasicInformation*, and **NtClose**. However, this service is faster because less ring transitions are made.

The information that is returned in the *FileInformation* buffer is defined by the following structure:

```
typedef struct _FILE_BASIC_INFORMATION {
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    ULONG FileAttributes;
} FILE_BASIC_INFORMATION;
```

Field	Description
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
FileAttributes	Attributes of the file

All dates and times are specified in the standard **Windows NT** system time format.

The file attributes field can be a combination of the following values:

FILE_ATTRIBUTE_NORMAL

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

3.7 I/O Completion Objects

This section describes the creation and use of completion objects.

3.7.1 Creating/Opening I/O Completion Objects

When a user wishes to synchronize the completion of I/O through the use of completion objects, he must first create or open an I/O completion object. Creating or opening a completion object causes the system to return a handle to the specified object.

I/O completion object handles are closed via the generic **NtClose** service. This service is discussed elsewhere in the **Windows NT** documentation. It should be noted that, just like all other system objects, a completion object is not actually deleted until all of the valid handles to it are closed and no referenced pointers remain.

The user APIs that support creating and opening completion objects are as follows:

> **NtCreateIoCompletion** - Create or open an I/O completion object and return a handle.
> **NtOpenIoCompletion** - Open an existing I/O completion object and return a handle.

3.7.1.1 Create/Open I/O Completion Objects

An I/O completion object can be created or opened using the **NtCreateIoCompletion** service:

NTSTATUS
NtCreateIoCompletion(
    OUT PHANDLE *IoCompletionHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL***,*
    IN ULONG *Count* **OPTIONAL**
    );

Parameters:

IoCompletionHandle - A pointer to a variable that receives the I/O completion object handle value.

DesiredAccess - Specifies the type of access that the caller requires to the completion object.

DesiredAccess Flags

SYNCHRONIZE - The completion object handle may be waited.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IO_COMPLETION_QUERY_STATE - The completion object may be queried.

IO_COMPLETION_MODIFY_STATE - The completion object may be modified.

The three following values are the generic access types that the caller may request. The mapping to specific access rights is given for each:

GENERIC_READ - Maps to *STANDARD_RIGHTS_READ* and *IO_COMPLETION_QUERY_STATE*.

GENERIC_WRITE - Maps to *STANDARD_RIGHTS_WRITE* and *IO_COMPLETION_MODIFY_STATE*.

GENERIC_EXECUTE - Maps to *STANDARD_RIGHTS_EXECUTE* and *SYNCHRONIZE*.

ObjectAttributes - A pointer to a structure that specifies the name of completion object, a root directory, a security descriptor, a quality of service descriptor, and a set of completion object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the completion object to be created or opened. This object name specification must be a fully qualified path, unless it is relative to the object directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the completion object specified by the *ObjectName* field is a path specification relative to the directory object supplied by this handle.

PSECURITY_DESCRIPTOR *SecurityDescriptor* - Optionally specifies the security descriptor that should be applied to the I/O completion object. The ACLs specified by the security descriptor are only applied to the object if it is created. If not supplied and the completion object is created, then the ACL placed on the completion object is formed from a combination of the ACL on the parent directory of the object and the current default ACL for the creating process.

PSECURITY_QUALITY_OF_SERVICE *SecurityQualityOfService* - Specifies the access a server should be given to the client's security context. This field is only used when a connection to a protected server is established. It allows the caller to control which parts of his security context are made available to the server and whether or not the server may impersonate the caller.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_INHERIT - Indicates that the handle to the I/O completion object is to be inherited by the new process when an **NtCreateProcess** operation is performed to create a new process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

OBJ_EXCLUSIVE - Indicates that the I/O completion object is to be created such that no other opens to the object may be performed.

OBJ_OPENIF - Indicates that if the I/O completion object already exists then it is to be opened; otherwise it is to be created.

Count - An optional value that supplies the maximum number of threads that should be concurrently active. If this parameter is not specified, then the number of processors is used.

The **NtCreateIoCompletion** service either causes a new I/O completion object to be created, or it opens an existing completion object. The action taken is dependent on the name of the object being opened, and whether the object already existed, and the value of the *OBJ_OPENIF ObjectAttributes* flag. If the object is created, then the maximum target concurrent thread count is set to the value specified by the *Count* parameter. A handle to the I/O completion object with the *DesiredAccess* is returned.

Once the caller has established a handle to an I/O completion object, he can then associate the completion object with a file, via the **NtSetInformationFile** system service. As each request for the file is completed, the I/O system stores a completion message in the I/O completion object.

Each completion message consists of a caller-determined key identifying the target file, a caller-supplied *CompletionContext* pointer, which was passed as *ApcContext* to the asynchronous **Nt...File** service when the request was originally issued, and a pointer to the returned I/O status block for the completed request.

3.7.1.2 Open I/O Completion Objects

An I/O completion object can be opened using the **NtOpenIoCompletion** service:

NTSTATUS
NtOpenIoCompletion(
    OUT PHANDLE *IoCompletionHandle*,
    IN ACCESS_MASK *DesiredAccess*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
    );

Parameters:

IoCompletionHandle - A pointer to a variable that receives the I/O completion object handle value.

DesiredAccess - Specifies the type of access that the caller requires to the completion object.

DesiredAccess Flags

SYNCHRONIZE - The completion object handle may be waited.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IO_COMPLETION_QUERY_STATE - The completion object may be queried.

IO_COMPLETION_MODIFY_STATE - The completion object may be modified.

The three following values are the generic access types that the caller may request. The mapping to specific access rights is given for each:

GENERIC_READ - Maps to *STANDARD_RIGHTS_READ* and *IO_COMPLETION_QUERY_STATE*.

GENERIC_WRITE - Maps to *STANDARD_RIGHTS_WRITE* and *IO_COMPLETION_MODIFY_STATE*.

GENERIC_EXECUTE - Maps to *STANDARD_RIGHTS_EXECUTE* and *SYNCHRONIZE*.

ObjectAttributes - A pointer to a structure that specifies the name of completion object, a root directory, a security descriptor, a quality of service descriptor, and a set of completion object attribute flags.

<u>ObjectAttributes Structure</u>

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the completion object to be opened. This object name specification must be a fully qualified path, unless it is relative to the object directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the completion object specified by the *ObjectName* field is a path specification relative to the directory object supplied by this handle.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_INHERIT - Indicates that the handle to the I/O completion object is to be inherited by the new process when an **NtCreateProcess** operation is performed to create a new process.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

The **NtOpenIoCompletion** service opens an existing I/O completion object and returns a handle to it through the *IoCompletionHandle* parameter.

As with the **NtCreateIoCompletion** service, once the caller has established a handle to an I/O completion object, he can then associate the completion object with a file, via the **NtSetInformationFile** system service. As each request for the file is completed, the I/O system stores a completion message in the I/O completion object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Each completion message consists of a caller-determined key identifying the target file, a caller-supplied *CompletionContext* pointer, which was passed as *ApcContext* to the asynchronous **Nt...File** service when the request was originally issued, and a pointer to the returned I/O status block for the completed request.

3.7.2 Operating on I/O Completion Objects

This section presents those services that manipulate I/O completion objects. The APIs that support operations on I/O completion objects are as follows:

> **NtQueryIoCompletion** - Query the state of an I/O completion object.
> **NtSetIoCompletion** - Inserts a message onto an I/O completion object.
> **NtRemoveIoCompletion** - Removes an entry from an I/O completion object.

3.7.2.1 Querying Completion Objects

The state of an I/O completion object can be queried using the **NtQueryIoCompletion** service:

NTSTATUS
NtQueryIoCompletion(
  IN HANDLE *IoCompletionHandle*,
  IN IO_COMPLETION_INFORMATION_CLASS IoCompletionInformationClass,
  OUT PVOID *IoCompletionInformation*,
  IN ULONG *IoCompletionInformationLength*,
  OUT PULONG *ReturnLength* **OPTIONAL**
  );

Parameters:

IoCompletionHandle - Supplies a handle to an open I/O completion object to be queried.

IoCompletionInformationClass - Specifies the type of information that should be returned about the I/O completion object. The information returned in the *IoCompletionInformation* buffer is defined by the following type codes:

IoCompletionInformationClass Values

IoCompletionBasicInformation - Returns basic information about the specified I/O completion object. **IO_COMPLETION_QUERY_STATE** access to the object is required.

IoCompletionInformation - A pointer to a buffer to receive the desired information about the I/O completion object. The contents of this buffer are defined by the *IoCompletionInformationClass* parameter described above.

IoCompletionInformationLength - The length of the *IoCompletionInformation* buffer in bytes.

ReturnLength - An optional pointer to a variable to receive the actual number of bytes of information returned in the *IoCompletionInformation* buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **NtQueryIoCompletion** service returns information about the specified I/O completion object. The information in the buffer is defined by the following type codes and structures.

IoCompletionInformation Format by I/O Completion Information Class

IoCompletionBasicInformation - Data type is *IO_COMPLETION_BASIC_INFORMATION*.

```
typedef struct _IO_COMPLETION_BASIC_INFORMATION {
    LONG Depth;
} IO_COMPLETION_BASIC_INFORMATION;
```

Field	Description
Depth	Depth, in messages, of the I/O completion object

IO_COMPLETION_QUERY_STATE access to the I/O completion object is required to obtain this information.

Once the information about the object has been returned, the caller can determine how much information was actually returned by examining the variable passed in as the *ReturnLength* parameter, if one was passed.

3.7.2.2 Setting Completion Objects

A completion message can be manually queued to an I/O completion object using the **NtSetIoCompletion** service:

```
NTSTATUS
NtSetIoCompletion(
    IN HANDLE IoCompletionHandle,
    IN ULONG KeyContext,
    IN PVOID ApcContext,
    IN NTSTATUS IoStatus,
    IN ULONG IoStatusInformation
    );
```

Parameters**:**

IoCompletionHandle - A handle to the I/O completion port.

KeyContext - Supplies the key contex that is returned during a call to **NtRemoveIoCompletion**.

ApcContext - Supplies the APC context that is returned during a call to **NtRemoveIoCompletion**.

IoStatus - Supplies the status data that will be returned in the *Status* field of the I/O status block during a call to **NtRemoveIoCompletion**.

IoStatusInformation - Supplies the information data that will be returned in the *Information* field of the I/O status block during a call to **NtRemoveIoCompletion**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **NtSetIoCompletion** service allows the caller to insert an I/O completion message into the completion object manually. This allows threads that are waiting on messages to arrive to be awakened to deal with a particular work item posted by the caller. Note that no I/O was actually performed to cause the completion message to be read by the remover of the item.

3.7.2.3 Removing Messages from Completion Objects

An I/O completion message can be removed from an I/O completion object using the **NtRemoveIoCompletion** service:

NTSTATUS
NtRemoveIoCompletion(
    IN HANDLE *IoCompletionHandle*,
    OUT PVOID **KeyContext*,
    OUT PVOID **ApcContext*,
    OUT PIO_STATUS_BLOCK *IoStatusBlock*,
    IN PLARGE_INTEGER *Timeout* **OPTIONAL**
    );

<u>Parameters:</u>

IoCompletionHandle - A handle to the I/O completion port.

KeyContext - Supplies a pointer to a variable to receive the key contex that was specified when the I/O completion object was associated with a file object.

ApcContext - Supplies a pointer to a variable to receive the context that was specified when the I/O was issued. This value was passed in as the *ApcContext* parameter when the I/O was queued.

IoStatus - Supplies a pointer to a variable that receives the final I/O completion status from the I/O operation.

Timeout - Supplies a pointer to an optional time out value.

The **NtRemoveIoCompletion** service removes a single I/O completion message from the completion object. If an entry is removed, then the *KeyContext, ApcContext,* and *IoStatus* variables receive the information about the I/O operation that was completed. The *Status* field of the *IoStatus* variable indicates whether or not the I/O operation was successfully completed. Note that this is separate from the return value from this service, which indicates whether or not a completion message was successfully removed from the completion object.

If there are no entries in the completion object, or if there are already *Count* threads concurrently ready and/or running due to other completion messages having been removed, then the calling thread will wait for another message according to the *Timeout* parameter. This parameter is treated in the normal manner of all time-out values in *Windows NT.*

4. Naming Conventions

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Devices in **Windows NT** are named according to a very simple set of rules. There are three general rules:

1) If there can only be one device of the specified type in the system, such as the PC subsystem keyboard, then the name of the device is simply the device type name.

2) If there can be more than one device of the specified type in the system, such as a floppy, then the name of the device is the device type name followed by a decimal number that indicates which device of that type it is.

3) For devices such as disks, which can be partitioned, the name of the partition is the name of the device followed by *\Partition* and a decimal number representing which partition on the disk it is. The first partition on a disk is called *\Partition1*. The name that refers to the entire device for partitioned media is *\Partition0.*

For example, the following are valid **Windows NT** device names.

o - *\Device\Floppy2*
o - *\Device\Harddisk1\Partition3*
o - *\Device\Keyboard*
o - *\Device\Mouse*

Note that all of the above device names are in a directory called the *\Device* directory. All device names in **Windows NT** reside in this object directory by convention. Any valid object directory operations can be used to determine the names of the devices on the system, provided the caller has the appropriate privileges and access to the object directory.

5. Appendix A - Time Field Changes

This section contains a list of those APIs that implicitly change the various time fields associated with a file.

5.1 Last Access Time

The Last Access Time field for a file is implicitly changed under the following conditions:

o - **NtQueryDirectoryFile** - The directory file's time field is updated.

o - **NtCreateFile** - The file's time field is set if the file was created.

o - **NtReadFile** - The file's time field is updated.

5.2 Last Modify Time

o - **NtCreateFile** - If the file was created, superseded, or overwritten, then the file's time field is updated. If the file was created or superseded then the parent directory's time field is also updated.

o - **NtSetInformationFile**

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- **FileLinkInformation** - The directory file containing the name of the link's time field is updated.

- **FileDispositionInformation** - The time field of the directory that contains the file is updated.

- **FileRenameInformation** - The old and the new parent directory file's times are updated.

o - **NtWriteFile** - The file's time field is updated.

5.3 Last Change Time

o - **NtCreateFile** - If the file was created, superseded, or overwritten, then the file's time field is updated. If the file was created or superseded then the parent directory's time field is also updated.

o - **NtSetInformationFile**

- **FileLinkInformation** - The time field of both the file and the directory containing the name of the link are updated.

- **FileDispositionInformation** - The time field of both the file and the directory containing the file are updated.

- **FileRenameInformation** - The time field of both the old and the new parent directories are updated.

- **FileAllocationInformation** - The file's time field is updated.

- **FileEndOfFileInformation** - The file's time field is updated.

o - **NtWriteFile** - The file's time field is updated.

o - **NtSetSecurityObject** - The file's time field is updated.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6. Revision History

Original Draft 1.0, March 21, 1989

Revision Draft 1.1, March 31, 1989

- Fixed spelling, grammar and numbering problems.
- Incorporated initial review comments.
- Removed all APIs that didn't use file handles.
- Rewrote overview section dealing with file objects.
- Added access right types to services.
- Redesigned **NtCreateFile** service.
- Removed **NtOpenFile** service.
- Revamped **NtQueryDirectoryFile** service.
- Added more types to **NtQueryInformationFile** service.
- Added more types to **NtSetInformationFile** service.
- Performed general fixup on most other services.
- Added description of DISPATCH_LEVEL driver context.
- Changed device work queues to device queues.
- Redesigned communication region protocol.
- Planned section on volume verification.

Revision Draft 1.2, May 12, 1989

- Allow setting of owner in **NtSetInformationFile.**
- Removed device info from **NtQueryFsInformationFile.**
- Changed **NtQueryFsInformationFile** to **QueryVolume**.
- Changed **NtSetFsInformationFile** to **SetVolume**.
- Added ChangeTime to appropriate structures.
- Added I/O provided time-out functions.
- Remove mount entry point from file systems.
- Fleshed out section on volume verification.
- Wrote section on error logging and handling.
- Wrote section on naming conventions.
- Added "subsystem input" section for terminals.
- Wrote section on network service description.
- Added directory access options.
- Fixed access type names.
- Make all byte offsets block/byte offsets.
    - o Read pointer
    - o Write pointer
    - o File allocation size
    - o End of file marker
- Add new security access types.
- Flesh out Miscellaneous I/O APIs.
- Change FILE_READ and _WRITE back again.

Revision Draft 1.3, October 9, 1989

- Split specification into two separate specs.
- Redo attributes again for security changes (twice).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Add "names" type to **NtQueryDirectoryFile** since other API was dropped by object manager.
- Change APC parameter to context and make PVOID.
- Add AscendingDirectories flag to volume info.
- Make file objects waitable objects.
- Make block/byte values zero-based.
- Add synchronous I/O.
- Only signal file handle if no event specified.
- Fix FILEINFO and FSINFO to be like all other APIs.
- Remove nonsensical directory desired accesses.
- Return actual action in Information on create/open.
- Add FILE_SHARE_ NO_DELETE and NO_RENAME.
- Drop FILE_CREATE_TREE_CONNECTION. Will be service.
- Drop FILE_EXECUTE desired access restrictions.
- Drop FILE_APPEND desired access restrictions.
- Drop or change name of privileges.
- Added time field changes appendix.

Revision Draft 1.4, January 21, 1990

- Added **NtOpenFile** system service.
- Removed **NtQueryAclFile** and **NtSetAclFile** APIs.
- Removed documentation on FileAclInformation.
- Added **NtLockFile** and **NtUnlockFile** services again.
- Change most services to have synchronous APIs.
- Redo attributes again for security changes.
- Revamped structures around security, especially for directories and subdirectories.
- Added EAs to **NtCreateFile**.
- Redo EA APIs and EA structures.
- Added rewind capabilities to EA and directory services.
- Added optional key parameter to **NtReadFile** and **NtWriteFile**.
- Fixed object attributes structure type name and fields.
- Converted APIs from Block and Byte to LARGE_INTEGER.
- Reversed polarity of shared delete and rename flags.
- Expanded type names out to full names.
- Miscellaneous edits and explanation changes.

Revision Draft 1.5, July 9, 1990

- Add EaListLength parameter to **NtQueryEaFile**.
- Removed FILE_MAPPED_IO option.
- Removed FILE_SHARE_RENAME share access.
- Document file sharing semantics.
- Add FileFsSizeInformation to **NtQueryVolumeInformationFile**.
- Removed FileFsBiosInformation from **NtQueryVolumeInformationFile**.
- Add RemovableMedia and SupportsObjects fields for volumes.
- Add FILE_OVERWRITE, FILE_OVERWRITE_IF to **NtCreateFile**.
- Document directory wildcarding.
- Document deleting a file is last valid I/O operation.
- Add FileAlignmentInformation to **NtQueryInformationFile**.
- Replace OBJ_OPEN_LINK with FILE_OPEN_LINK.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Add FILE_TRAVERSE as legal directory access.
- Add FILE_OPEN_UNKNOWN_OBJECT option.
- Add FILE_OPENED_UNKNOWN_OBJECT I/O status block value.
- Replace FILE_DISABLE_CACHING with FILE_NO_INTERMEDIATE_BUFFERING and add requirement restrictions description.
- Add FILE_COMPLETE_IF_OPLOCKED option to create and open.
- Add FileRemainingNameInformation query information type.
- Explicitly state that locking beyond EOF is permissible.
- Switch fields in FILE_FULL_EA_INFORMATION to keep compatibility with OS/2.
- Fixed references to IOSB and PIOSB.
- Removed explicit ACL and owner interfaces and converted to the new security semantics.
- Add ability for synchronous I/O locks to be asynchronous.
- Subsumed **NtSetNewSizeFile** functionality in **NtSetInformationFile**.
- Removed FileOwnerInformation from **NtQueryInformationFile**.
- Removed FileOwnerInformation from **NtSetInformationFile**.
- Removed FILE_OWNER_INFORMATION structure type declaration.

Revision Draft 1.6, July 15, 1993

- Removed outdated "++" notation for subsystems.
- Updated system name from NT OS/2 to Windows NT.
- Removed error ports from all appropriate APIs.
- Added new file attribute definitions for FILE_ATTRIBUTE_TEMPORARY, FILE_ATTRIBUTE_ATOMIC_WRITE, and FILE_ATTRIBUTE_XACTION_WRITE.
- Removed all vestiges of "unknown objects" and all related functionality.
- Replaced old style create/open directory manipulation flags (see next).
- Documented all new Create/Open options:
    - o FILE_DIRECTORY_FILE
    - o FILE_NON_DIRECTORY_FILE
    - o FILE_RANDOM_ACCESS
    - o FILE_NO_EA_KNOWLEDGE
    - o FILE_DELETE_ON_CLOSE
    - o FILE_OPEN_BY_FILE_ID
    - o FILE_OPEN_FOR_BACKUP_INTENT
- Updated all appropriate CHAR's to WCHAR's in accordance w/Unicode changes.
- Updated all STRING's to UNICODE_STRING's in accordance w/Unicode changes.
- Removed source/target process from **NtReadFile** and **NtWriteFile**.
- Removed **NtReadTerminalFile** API.
- Updated all TIME data types to LARGE_INTEGER's.
- Moved FILE_ATTRIBUTE_DIRECTORY flag into attributes for query operations.
- Added FileBothDirectoryInformation file information class to **NtQueryDirectoryFile**.
- Changed Action field of FILE_NOTIFY_INFORMATION to ULONG.
- Added FileAlternateNameInformation to **NtQueryInformationFile**.
- Added FileStreamInformation to **NtQueryInformationFile**.
- Changed FileNameInformation to FileRenameInformation for **NtSetInformationFile**.
- Updated Length parameter to LARGE_INTEGER from ULONG for locking services.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Added FileFsDeviceInformation and FileFsAttributeInformation to **NtQueryVolumeInformation**.

Revision Draft 1.7, May 1, 1995

- Added new FILE_OPEN_TRANSACTED and FILE_RESERVE_OPFILTER create/open options.
- Removed FILE_ATTRIBUTE_ATOMIC_WRITE and FILE_ATTRIBUTE_XACTION_WRITE and added FILE_ATTRIBUTE_COMPRESSED and FILE_ATTRIBUTE_OFFLINE..
- Added new STORAGE_TYPE enumerated type as well as new create/open option fields for storage types.
- Added values for FILE_NOTIFY_CHANGE_STREAM_NAME, FILE_NOTIFY_CHANGE_STREAM_SIZE, and FILE_NOTIFY_CHANGE_STREAM_WRITE.
- Added documentation of file system attributes flags, and included new flags FILE_FILE_COMPRESSED and FILE_VOLUME_IS_COMPRESSED for compression.
- Added FILE_VIRTUAL_VOLUME device characteristic flag for virtual volumes.
- Added the following query and set information class information values and their associated structure type definitions:
  - FileCompressionInformation
  - FileCopyOnWriteInformation
  - FileCompletionInformation
  - FileMoveClusterInformation
  - FileOleClassIdInformation
  - FileOleStateBitsInformation
  - FileApplicationExplorableInformation
  - FileApplicationExplorableChildrenInformation
  - FileObjectIdInformation
  - FileOleAllInformation
  - FileContentIndexInformation
  - FileInheritContentIndexInformation
  - FileOleInformation
- Added new **NtQueryOleDirectoryFile** API description.
- Added new FileOleDirectoryInformation directory information class and its associated structure type definition.
- Added new directory query information class for OLE files.
- Added query and set volume information class information values and its associated type definitions for FileFsControlInformation
- Added new **NtQueryQuotaInformationFile** and **NtSetQuotaInformationFile** API descriptions.
- Added new data structure types (FILE_GET_QUOTA_INFORMATION and FILE_QUOTA_INFORMATION) for the above services.
- Added new **NtDeleteFile** API description.
- Added new **NtQueryAttributesFile** API description.
- Added new I/O completion object section for APIs, access rights, information class values, and data structures.
- Removed old **NtDeviceIoControlFile** and **NtFsControlFile** appendicies to alleviate concerns that they weren't filled in (since they never will be populated).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Added device types for FILE_DEVICE_BATTERY and
  FILE_DEVICE_BUS_EXTENDER.
- Removed POSIX and OS/2 subsystem API implementation sections

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 IRP Language Definition

Author: *Gary D. Kimura*

Revision 1.0x, December 15, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction... 1

2. Valid IRP combination 2
 2.1 Disk Driver IRPs.. 2
 2.2 File System IRPs.. 3
 2.3 Keyboard Driver IRPs.. 5
 2.4 Mouse Driver IRPs 5
 2.5 Network Drivers IRPs.. 6
 2.6 Sound Driver IRPs 6
 2.7 Tape Driver IRPs 6
 2.8 Terminal Driver IRPs... 6
 2.9 Video Driver IRPs.. 6

3. IRP Function Descriptions.. 8
 3.1 Close 8
 3.2 Create 11
 3.3 Device Control 19
 3.4 Directory Control(Notify Change Directory) 19
 3.5 Directory Control(Query Directory) 19
 3.6 File System Control(Dismount Volume) 19
 3.7 File System Control(Lock Volume)... 19
 3.8 File System Control(Mount Volume) 19
 3.9 File System Control(Query Information File System) 19
 3.10 File System Control(Set Information File System) 19
 3.11 File System Control(Unlock Volume) 19
 3.12 File System Control(Verify Volume) 19
 3.13 Internal Device Control.. 19
 3.14 Lock Control(Lock) 19
 3.15 Lock Control(Unlock All).. 19
 3.16 Lock Control(Unlock Single) 20
 3.17 Query Acl... 20
 3.18 Query Ea 20
 3.19 Query Information 20
 3.20 Query Volume Information.. 20
 3.21 Read 20
 3.22 Read Terminal.. 20
 3.23 Set Acl.. 20
 3.24 Set Ea 20
 3.25 Set Information 20
 3.26 Set New Size 20
 3.27 Set Volume Information 20
 3.28 Write 20

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

The purpose of this chapter is to define the semantic contents of an I/O Request Packet (IRP). The information contained here is intended for use mainly by Device Driver and File System developers. The I/O system sends to the various Device Drivers[1] a stream of multiple IRPs that the drivers must interpret and respond to. Figure 1 shows the relationship between the device driver and the I/O system. Communication between the I/O system and the Device Driver is through IRPs. This chapter concentrates on the IRP language.

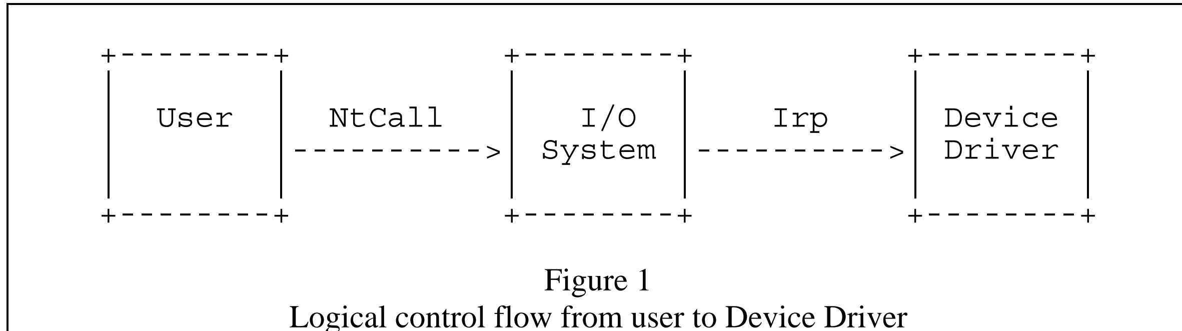


Figure 1
Logical control flow from user to Device Driver

Each IRP has a well defined format and semantic meaning, and the order in which they are sent must adhere to certain rules. The ordering of IRPs and responses form a context sensitive language.

Each IRP contains a common header section followed by one or more function specific records (also called IRP stack locations). From a Device Drivers viewpoint each IRP request is a single record describing one function to perform. That is, the drivers only interpret one function specific record. The additional stack locations are for use when a driver issues subsequent IRPs to a lower level driver and wishes to reuse the original IRP.

Each IRP function is identified by a major and minor function field in the IRP stack location record. The list of possible function combinations are listed below. Each line lists a major function code followed (in paranthesis) by a minor function code. Note that some major functions (e.g., CREATE) do not make use the minor function field.

CLOSE()
CONFIGURATION_CONTROL(...)
CREATE()
DEVICE_CONTROL(...)
DIRECTORY_CONTROL(NOTIFY_CHANGE_DIRECTORY)
DIRECTORY_CONTROL(QUERY_DIRECTORY)
FILE_SYSTEM_CONTROL(DISMOUNT_VOLUME)
FILE_SYSTEM_CONTROL(LOCK_VOLUME)

---

[1]For clarity we will use the term Device Driver to refer to both Device Drivers and File systems.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_SYSTEM_CONTROL(MOUNT_VOLUME)
FILE_SYSTEM_CONTROL(QUERY_INFO_FILE_SYSTEM)
FILE_SYSTEM_CONTROL(SET_INFO_FILE_SYSTEM)
FILE_SYSTEM_CONTROL(UNLOCK_VOLUME)
FILE_SYSTEM_CONTROL(VERIFY_VOLUME)
INTERNAL_DEVICE_CONTROL(...)
LOCK_CONTROL(LOCK)
LOCK_CONTROL(UNLOCK_ALL)
LOCK_CONTROL(UNLOCK_SINGLE)
QUERY_ACL()
QUERY_EA()
QUERY_INFORMATION()
QUERY_VOLUME_INFORMATION()
READ()
READ_TERMINAL()
SET_ACL()
SET_EA()
SET_INFORMATION()
SET_NEW_SIZE()
SET_VOLUME_INFORMATION()
WRITE()

/* We need to define the minor function codes for the configuration, device, and internal device function codes. */

Each Device Driver will only receive a combination of the preceding function codes based on the drivers device type. This means that a file system device driver can expect to receive different functions than the keyboard device driver, or a disk driver. The possible device driver types are:

Disk Driver,
File System (including network redirector),
Keyboard Driver,
Mouse Driver,
Network Drivers,
Sound Driver,
Tape Driver,
Terminal Driver, and
Video Driver,

/* We will need to futher expand on the different network device drivers */

The remainder of this chapter describes the valid combination of IRP function codes that each different device driver can expect to receive. This is followed by a section listing every IRP function code along with a description of the function's parameters, semantics, and I/O completion status codes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2. Valid IRP Function Combinations

The section contains an individual table for each device driver type that lists the set of valid IRP functions that can be sent to the driver and under what conditions the functions are sent.

2.1 Disk Driver IRPs

The set of possible IRPs that can be sent to a disk driver are:

IRP Function	When sent
CLOSE	Anytime.
CREATE	Anytime.
DEVICE_CONTROL (...)	Anytime.
READ	Anytime.
WRITE	Anytime.

2.2 File System IRPs

The set of possible IRPs that can be sent to a file system are:

IRP Function	When sent
CLOSE	Only after a successful CREATE and then only on an opened file. This closes the file so no other operation can be performed on the file other than CREATE.
CREATE	Only after a successful MOUNT_VOLUME and then only on a mounted volume that is not locked. If successful the file is considered opened.
DIRECTORY_CONTROL (NOTIFY_CHANGE_DIRECTORY)	Only after a successful CREATE and then only on an opened directory file.
DIRECTORY_CONTROL (QUERY_DIRECTORY)	Only after a successful CREATE and then only on an opened directory file.
FILE_SYSTEM_CONTROL (DISMOUNT_VOLUME)	Only after a successful MOUNT_VOLUME and then only on a mounted volume. This

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

	dismounts the volume, so no other operation can be performed on the volume other than MOUNT_VOLUME.
FILE_SYSTEM_CONTROL (LOCK_VOLUME)	Only after a successful CREATE and then only on an opened file. This locks the volume containing the file such that no other creates using the same volume will succeed until the volume is unlocked. To be successful, the file used to lock the volume must also be the only opened file on the volume.
FILE_SYSTEM_CONTROL (MOUNT_VOLUME)	Anytime. If the operation is successful then a new device object for the volume is created and the volume is considered mounted and not locked.
FILE_SYSTEM_CONTROL (QUERY_INFO_FILE_SYSTEM)	Only after a successful CREATE and then only on an opened file.
FILE_SYSTEM_CONTROL (SET_INFO_FILE_SYSTEM)	Only after a successful CREATE and then only on an opened file.
FILE_SYSTEM_CONTROL (UNLOCK_VOLUME)	Only after a successful CREATE and then only on a opened file. The file system must handle the situation where the user is attempting to unlock a volume that is not locked. If successful this operation unlocks a previously locked volume so that other creates using the volume can now succeed.
FILE_SYSTEM_CONTROL (VERIFY_VOLUME)	Only after a successful MOUNT_VOLUME and then only on a mounted volume.
LOCK_CONTROL (LOCK)	Only after a successful CREATE and then only on an opened file. If successful this operation locks a range of bytes within a file. The locks remain in affect until they are explicitly unlocked or the file is closed.
LOCK_CONTROL (UNLOCK_ALL)	Only after a successful CREATE and then only on an opened file. The file system must handle the situation where an unlock is received even though there are no outstanding locks for that user.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

LOCK_CONTROL (UNLOCK_SINGLE)	Only after a successful CREATE and then only on an opened file. The file system must handle the situation where an unlock is received even though there is not a corresponding lock.
QUERY_ACL	Only after a successful CREATE and then only on an opened file.
QUERY_EA	Only after a successful CREATE and then only on an opened file.
QUERY_INFORMATION	Only after a successful CREATE and then only on an opened file.
QUERY_VOLUME_INFORMATION	Only after a successful CREATE and then only on an opened file.
READ	Only after a successful CREATE and then only on an opened file.
SET_ACL	Only after a successful CREATE and then only on an opened file.
SET_EA	Only after a successful CREATE and then only on an opened file.
SET_INFORMATION	Only after a successful CREATE and then only on an opened file.
SET_NEW_SIZE	Only after a successful CREATE and then only on an opened file.
SET_VOLUME_INFORMATION	Only after a successful CREATE and then only on an opened file.
WRITE	Only after a successful CREATE and then only on an opened file.

2.3 Keyboard Driver IRPs

The set of possible IRPs that can be sent to the Keyboard driver are:

IRP Function	When sent

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

CLOSE	Anytime.
CREATE	Anytime.
DEVICE_CONTROL (...)	Anytime.
QUERY_INFORMATION	Anytime.
READ	Anytime.
SET_INFORMATION	Anytime.
WRITE	Anytime.

2.4 Mouse Driver IRPs

The set of possible IRPs that can be sent to the Mouse driver are:

IRP Function	When sent
CLOSE	Anytime.
CREATE	Anytime.
DEVICE_CONTROL (...)	Anytime.
QUERY_INFORMATION	Anytime.
READ	Anytime.
SET_INFORMATION	Anytime.
WRITE	Anytime.

2.5 Network Drivers IRPs

The set of possible IRPs that can be sent to the Network drivers are:

IRP Function	When sent

/* This table needs to be filled in */

2.6 Sound Driver IRPs

The set of possible IRPs that can be sent to the Sound driver are:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IRP Function	When sent

/* This table needs to be filled in */

2.7 Tape Driver IRPs

The set of possible IRPs that can be sent to the Tape driver are:

IRP Function	When sent

/* This table needs to be filled in */

2.8 Terminal Driver IRPs

The set of possible IRPs that can be sent to the Terminal driver are:

IRP Function	When sent

/* This table needs to be filled in */

2.9 Video Driver IRPs

The set of possible IRPs that can be sent to the Video driver are:

IRP Function	When sent
CLOSE	Anytime.
CREATE	Anytime.
DEVICE_CONTROL (...)	Anytime.
QUERY_INFORMATION	Anytime.
READ	Anytime.
SET_INFORMATION	Anytime.
WRITE	Anytime.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. IRP Function Descriptions

This section describes the input parameters and semantics for each IRP function code. It also discusses the interactions between the parameters and lists possible return status codes.

The parameter descriptions list all the fields that are used within the IRP by the operation being described. Each parameter is either Read (i.e., used as input to the operation), Set (i.e., used as output for the operation), or Ignored. To help distinguish the parameters we will also use the two terms IrpFlags and FunctionFlags to denote the flags field of the IRP header and the I/O stack location respectively.

In the description of the return status codes we do not include generic values such as STATUS_PENDING or STATUS_INVALID_PARAMETER which can be returned for any IRP. We also do not describe values that can be returned by a lower level device drivers such as STATUS_PARITY_ERROR.

3.1 Close

The close function is used to close a previously opened file or directory. Its two input parameters are a device object and an IRP. The device object parameter points to a volume previously mounted by the Device Driver and is where the file opened file exists. The IRP contains the close function parameters (and are listed below).

Besides closing the file, this function will optionally deletes the file based upon the disposition specified by the caller (See the SET_INFORMATION operation). If this is the last file object with the file opened and the disposition is *delete on close* then the file is removed from the on-disk structure.

Close (
    IN PDEVICE_OBJECT *DeviceObject*,
    IN PIRP *Irp*
    );

Parameters within the IRP:

Parameter type and name	Description
PMDL *MdlAddress*	Ignored.
ULONG *IrpFlags*	Ignored.
STRING *FileObject->FileName*	Ignored.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG *FileObject->RelatedFileObject*	Ignored.
PVOID *FileObject->FsContext*	Read and Set. The driver uses this field to retrieve any private data (established by the CREATE function) that needs to be processed in order to close the file. It is set to NULL upon return from the close function.
PVOID *FileObject->FsContext2*	Read and Set. The driver uses this field to retrieve any private data (established by the CREATE function) that needs to be processed in order to close the file. It is set to NULL upon return from the close function.
PVOID *FileObject->SectionObjectPointer*	Set. The close function must set this field to NULL.
IO_STATUS_BLOCK *IoStatus*	Set. This receives the final return status of the operation. The possible return status values are listed later.
PEPROCESS *AlternateProcess*	Ignored.
KPROCESSOR_MODE *RequestorMode*	Ignored.
PVOID *SystemBuffer*	Ignored.
PIO_STATUS_BLOCK *UserIosb*	Ignored.
PKEVENT *UserEvent*	Ignored.
LARGE_INTEGER *AllocationSize*	Ignored.
PVOID *UserBuffer*	Ignored.

<u>Parameters within the IRP Stack</u>:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Parameter type and name	Description
UCHAR *MajorFunction*	Read. Must be equal to IRP_MJ_CLOSE.
UCHAR *MinorFunction*	Ignored.
UCHAR *FunctionFlags*	Ignored.
UCHAR *Control*	Ignored.

<u>Iosb Return Status and Information</u>:

The following status codes are used to complete the CLOSE function.

Return status followed by information field of IOSB	Description
STATUS_SUCCESS Ignored	Indicates that the opened file has been closed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.2 Create

The create function is used to create or open a file or a directory. Its two input parameters are a device object and an IRP. The device object parameter points to a volume previously mounted by the Device Driver and is where the file will exist. The IRP contains the create function parameters (and are listed below).

Create (
  IN PDEVICE_OBJECT *DeviceObject*,
  IN PIRP *Irp*
  );

Parameters within the IRP:

Parameter type and name	Description
PMDL *MdlAddress*	Ignored.
ULONG *IrpFlags*	Ignored.
STRING *FileObject->FileName*	Read. This is the name of the file being opened.
ULONG *FileObject->RelatedFileObject*	Read. This field is used for path relative file names. If it is null then the file name is relative to the root of the volume (e.g., "\CONFIG.SYS" is the name of the configuration file located in root directory). If is it not null then it points to a previously opened file object representing a directory on the volume, and the file name is relative to the specified directory (e.g., if the related file object is "\NT\SDK" the file name can be "INC\NTIOAPI.H"). Note that path relative file names do not begin with a backslash.
PVOID *FileObject->FsContext*	Set. This is used by the Device Driver to store file object specific information that can be retrieved later when the driver is called to perform subsequent operations on the file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The FAT file system stores in this field a pointer to an internal File Control Block (FCB) structure.

PVOID
FileObject->FsContext2

Set. This is used by the Device Driver to store file object specific information that can be retrieved later when the driver is called to perform subsequent operations on the file.

The FAT file system only uses this field for directories. It is a pointer to an internal Context Control Block (CCB) structure.

PVOID
FileObject->SectionObjectPointer

Set. It is set to the longword context for the file. It is not used for directories. For every opened file the driver allocates a single longword of context for exclusive use by the memory management system. All file objects that denote the same file point to the same longword context.

In FAT this is done by reserving a longword field in the FCB and having each section object pointer point to this field.

IO_STATUS_BLOCK
IoStatus

Set. This receives the final return status of the operation. The possible return status values are listed later.

PEPROCESS
AlternateProcess

Ignored.

KPROCESSOR_MODE
RequestorMode

Read. This is the mode of the requestor. It is used for to help decide if the requestor has the proper access rights to the file.

/**** We also need to pass in the token of the requestor ****/

PVOID
SystemBuffer

Read. This field is only used if the file is being created and then it only specifies the optional extended attributes for the file. If the field is null the file will not be created with extended attributes. The create operation must complete with an error if there are any

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

	problems with the extended attributes. For FAT there is a 64K limit to the size of the extended attributes (as packed on the disk). The create operation will complete with an error if this limit is exceeded.
PIO_STATUS_BLOCK *UserIosb*	Ignored.
PKEVENT *UserEvent*	Ignored.
LARGE_INTEGER *AllocationSize*	Read. This field is only used if the file is being created and is ignored for directories and for open operations. It specifies the initial file allocation in bytes to allocate to the file. This is not the same as the end-of-file location.
PVOID *UserBuffer*	Ignored.

Parameters within the IRP Stack:

Parameter type and name	Description
UCHAR *MajorFunction*	Read. Must be equal to IRP_MJ_CREATE.
UCHAR *MinorFunction*	Ignored.
UCHAR *FunctionFlags*	Ignored.
UCHAR *Control*	Ignored.
ULONG *DesiredAccess*	Read. This is the access mask that the user is trying to acquire to the file. If the user is trying to open a file the mask will be a combination of the following values:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

DELETE,
READ_CONTROL,
WRITE_DAC,
WRITE_OWNER,
SYNCHRONIZE,
FILE_READ_DATA,
FILE_WRITE_DATA,
FILE_APPEND_DATA,
FILE_READ_EA,
FILE_WRITE_EA,
FILE_EXECUTE,
FILE_READ_ATTRIBUTES, and
FILE_WRITE_ATTRIBUTES.

If the user is trying to open a directory the mask will be a combination of the following values:

DELETE,
READ_CONTROL,
WRITE_DAC,
WRITE_OWNER,
SYNCHRONIZE,
FILE_LIST_DIRECTORY,
FILE_ADD_FILE,
FILE_ADD_SUBDIRECTORY,
FILE_READ_EA,
FILE_WRITE_EA,
FILE_TRAVERSE,
FILE_DELETE_CHILD,
FILE_READ_ATTRIBUTES, and
FILE_WRITE_ATTRIBUTES.

The driver must ensure that the combination of the caller's privileges and requestor's mode grants all of the desired accesses that the user is trying to acquire.

ULONG
Options

Read. This field contains all of the different create options and create disposition flags that the user can specify in an NT call. The valid flags and their meanings are listed below:

FILE_CREATE_DIRECTORY

Read. Indicates that the user is creating a

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

new directory.

FILE_OPEN_DIRECTORY	Read. Indicates that the user is opening an existing directory.
FILE_WRITE_THROUGH	Ignored, but saved away for use by subsequent read and write operations to the file object.
FILE_SEQUENTIAL_ONLY	Ignored, but saved away for use by subsequent read and write operations to the file object.
FILE_MAPPED_IO	Ignored, but saved away for use by subsequent read and write operations to the file object.
FILE_DISABLE_CACHING	Ignored, but saved away for use by subsequent read and write operations to the file object.
FILE_SYNCHRONOUS_IO_ALERT	Ignored.
FILE_SYNCHRONOUS_IO_NONALERT	Ignored.
FILE_CREATE_TREE_CONNECTION	Read. Only used by the network. /**** need a complete description of this parameter ****/
FILE_SUPERSEDE << 24[2]	Read. Indicates that if the file already exists it should be superseded, and if the file does not exist it should be created.
FILE_CREATE << 24	Read. Indicates that if the file already exists it is an error, and if the file does not exist it should be created.
FILE_OPEN << 24	Read. Indicates that if the file already exists it is to be opened, and if the file

[2]To test if the flags FILE_SUPERSEDE, FILE_OPEN, FILE_CREATE, and FILE_OPEN_IF are in the options parameter the driver must first shift the flag 24 bits to the left and then do the test (e.g., Option & (FILE_SUPERSEDE << 24)).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

does not exist it is an error.

FILE_OPEN_IF << 24 — Read. Indicates that if the file already exists it is to be opened, and if the file does not exist it should be created.

/**** We need a list of the illegal flag combinations, and state that they will never be seen in an IRP ****/

USHORT *FileAttributes* — Read. This field specifies the DOS file attributes to use when creating or superseding a file, and is ignored when opening an existing file. It is a combination of any of the following flags:

FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_CONTROL, and
FILE_ATTRIBUTE_NORMAL

The flag FILE_ATTRIBUTE_NORMAL overrides all other file attribute flags. (i.e., if the user specifies normal and readonly then the file is created as a normal file and not readonly).

USHORT *ShareAccess* — Read. This field specifies the share mode access between processes trying to open the same file. All users that open a file for shared access must specify the exact same share flags. This is separate from their desired access. For example a file opened shared read, write, and delete, must be opened by all users as shared read, write, and delete even though the desired access might only specify read access.

The valid flags and their meanings are listed below:

FILE_SHARE_READ — Read. Indicates that the file can be opened by others for read access. If the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

file is already opened for shared read access then other users can open it for read access.

FILE_SHARE_WRITE — Read. Indicates that the file can be opened by others for write access. If the file is already opened for shared write access then other users can open it for write access.

FILE_SHARE_DELETE — Read. Indicates that the file can be opened by others for delete access. If the file is already opened for shared delete access then other users can open it for delete access.

FILE_SHARE_RENAME — Read. Indicates that the file can be renamed by others. If the file is already opened for shared renamed access then other users can rename the file.

The test that a user requesting shared read, write, or delete can be done by the Device Driver during the create operation (i.e., a user is allowed read access to a shared file if the shared access flags match, shared read is specified, and the file's security protection allows for read access). The test for rename access must be deferred until the a rename IRP is processed (see the Set Information IRP description).

ULONG
EaLength — Read. This parameter is specified only if the user is creating or superseding a file and has specified an EA for the file. This parameter is then the size, in bytes, of the EA set specified by the user. (i.e., it is the size of the system buffer parameter).

Iosb Return Status and Information:

The following status codes are used to complete the CREATE function.

Return status followed by

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

information field of IOSB	Description
STATUS_SUCCESS FILE_OPENED	Indicates that an existing file has been successfully located and opened.
STATUS_SUCCESS FILE_SUPERSEDED	Indicates that an existing file has been successfully located and superseded.
STATUS_SUCCESS FILE_CREATED	Indicates that an existing file (of the same name) does not exist and that a new file has been successfully created.
STATUS_ACCESS_DENIED Ignored	Indicates that because of protection on the file, parent directory, or volume access has been denied to the file. This can also occur if the caller specified options or share access flags are not compatible with either the file or the previous share access that it was opened with.
STATUS_OBJECT_NAME_INVALID Ignored	Indicates that the last name in the object's file name field does not contain a syntactically valid name (e.g., it's too long or contains invalid characters).
STATUS_OBJECT_NAME_NOT_FOUND Ignored	Indicates that the last name in the object's file name field is not the name of an existing file.
STATUS_OBJECT_PATH_INVALID Ignored	Indicates that a name within the path part of the object's file name field does not contain a syntactically valid name.
STATUS_OBJECT_PATH_NOT_FOUND Ignored	Indicates that a name within the path part of the object's file name field does not contain the name of an existing directory.
STATUS_DISK_FULL_ERROR Ignored	Indicates that because the disk is full the file cannot be created. This can occur when disk space cannot be allocated for the directory entry, file node, or the extended attributes.
STATUS_DISK_FULL_WARNING FILE_SUPERSEDED	Indicates that the file has been superseded but because the disk is full the file cannot be given the user specified file allocation size.
STATUS_DISK_FULL_WARNING	Indictes that the file has been created but because

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_CREATED	the disk is full the file cannot be given ths user specified file allocation size.
STATUS_EA_INVALID Ignored	Indicates that the EA structure passed into this function is syntactically invalid.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.3 Device Control

3.4 Directory Control(Notify Change Directory)

3.5 Directory Control(Query Directory)

3.6 File System Control(Dismount Volume)

3.7 File System Control(Lock Volume)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.8 File System Control(Mount Volume)

The mount function is used mount a new disk volume. Its two input parameters are a device object and an IRP. The device object parameter points to the Device Drivers original device object that is created when the driver is initialized.

The mount operation can handle mounting new volume, and remounting a previously mounted volume. The parameter description that follows assumes that it is processing a new volume. At the end of the description we cover the updating required for the remount case.

Mount (
    IN PDEVICE_OBJECT *DeviceObject*,
    IN PIRP *Irp*
    );

Parameters within the IRP:

Parameter type and name	Description
PMDL *MdlAddress*	Ignored.
ULONG *IrpFlags*	Ignored.
PFILE_OBJECT *FileObject*	Ignored.
IO_STATUS_BLOCK *IoStatus*	Set. This receives the final return status of the operation. The possible return status values are listed later.
PEPROCESS *AlternateProcess*	Ignored.
KPROCESSOR_MODE *RequestorMode*	Ignored.
PVOID *SystemBuffer*	Ignored.
PIO_STATUS_BLOCK *UserIosb*	Ignored.
PKEVENT	Ignored.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

UserEvent	
LARGE_INTEGER *AllocationSize*	Ignored.
PVOID *UserBuffer*	Ignored.

<u>Parameters within the IRP Stack</u>:

Parameter type and name	Description
UCHAR *MajorFunction*	Read. Must be equal to IRP_MJ_FILE_SYSTEM_CONTROL.
UCHAR *MinorFunction*	Read. Must be equal to IRP_MN_MOUNT_VOLUME.
UCHAR *FunctionFlags*	Ignored.
UCHAR *Control*	Ignored.
PDEVICE_OBJECT *Vpb->DeviceObject*	Set. If the mount is successful this field is set the point to the newly allocated device object for the volume. If the mount is unsuccessful or this is a remount then this field is not updated.
ULONG *Vpb->DeviceObject->Flags*	Set. If the mount is successful then the flag DO_DIRECT_IO is set in the newly created device objects flags field. Setting this flag allows the Device Driver to receive unbuffered I/O requests for this volume.
ULONG *Vpb->SerialNumber*	Set. If the mount is successful this field is set to the serial number found on the volume. It is ignored if the mount is unsuccessful or in the case of a remount.
CHAR *Vpb->VolumeName[20]*	Set. If the mount is successful this field is set to the label found on the volume. If the volume does not have a label then this field

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

	should be set to all spaces. For FAT the volume label, if present, is found in the root directory as a special dirent.
PDEVICE_OBJECT *DeviceObject*	Read. This is the device object that the Device Driver is to use when formulating IRPs to read or write to the volume. It is also called the target device object. If the volume is mounted successful this value must be remembered so the driver can handle subsequent requests to the volume.

Iosb Return Status and Information:

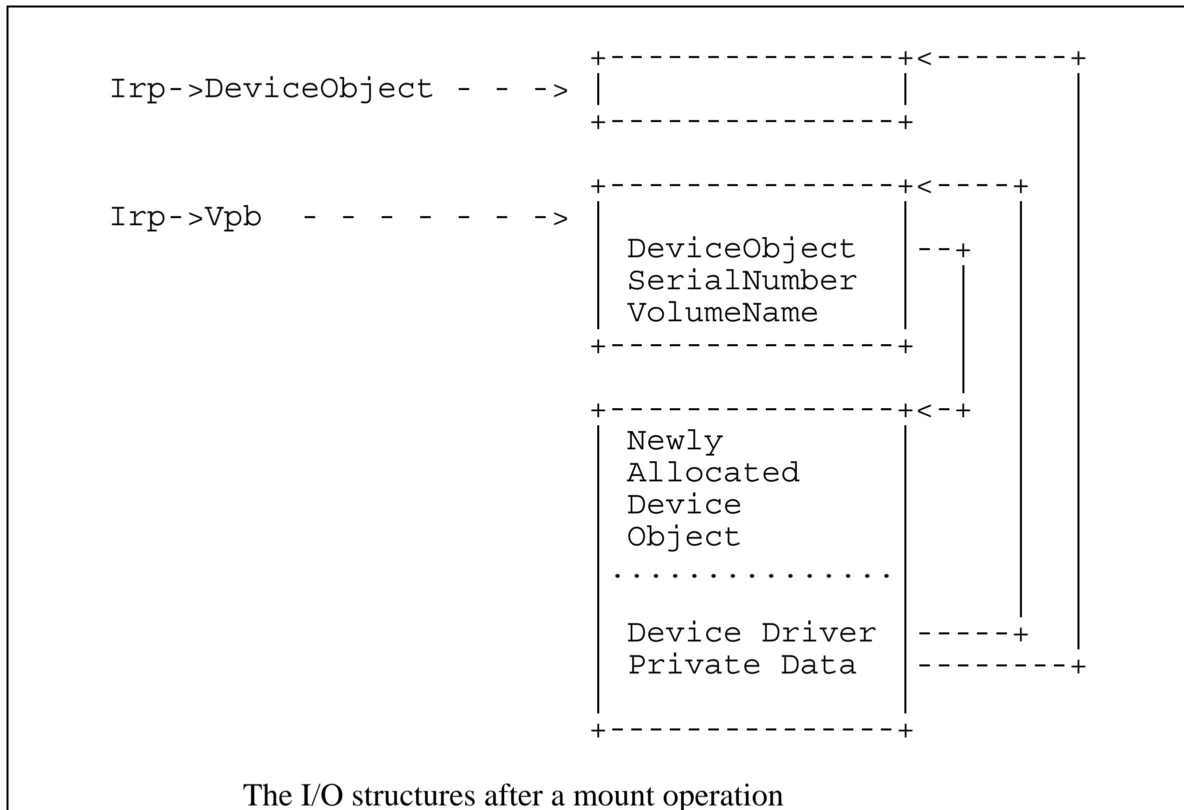
The following status codes are used to complete the MOUNT function.

Return status followed by information field of IOSB	Description
STATUS_SUCCESS Ignored	Indicates that the volume has been successful mounted.
STATUS_WRONG_VOLUME Ignored	Indicates that the volume cannot be mounted either because it does not recognize the on-disk structure or the on-disk structure has been currupted.

Mounting a new volume:

The following figure shows the major I/O structures after processing a successful mount request.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License



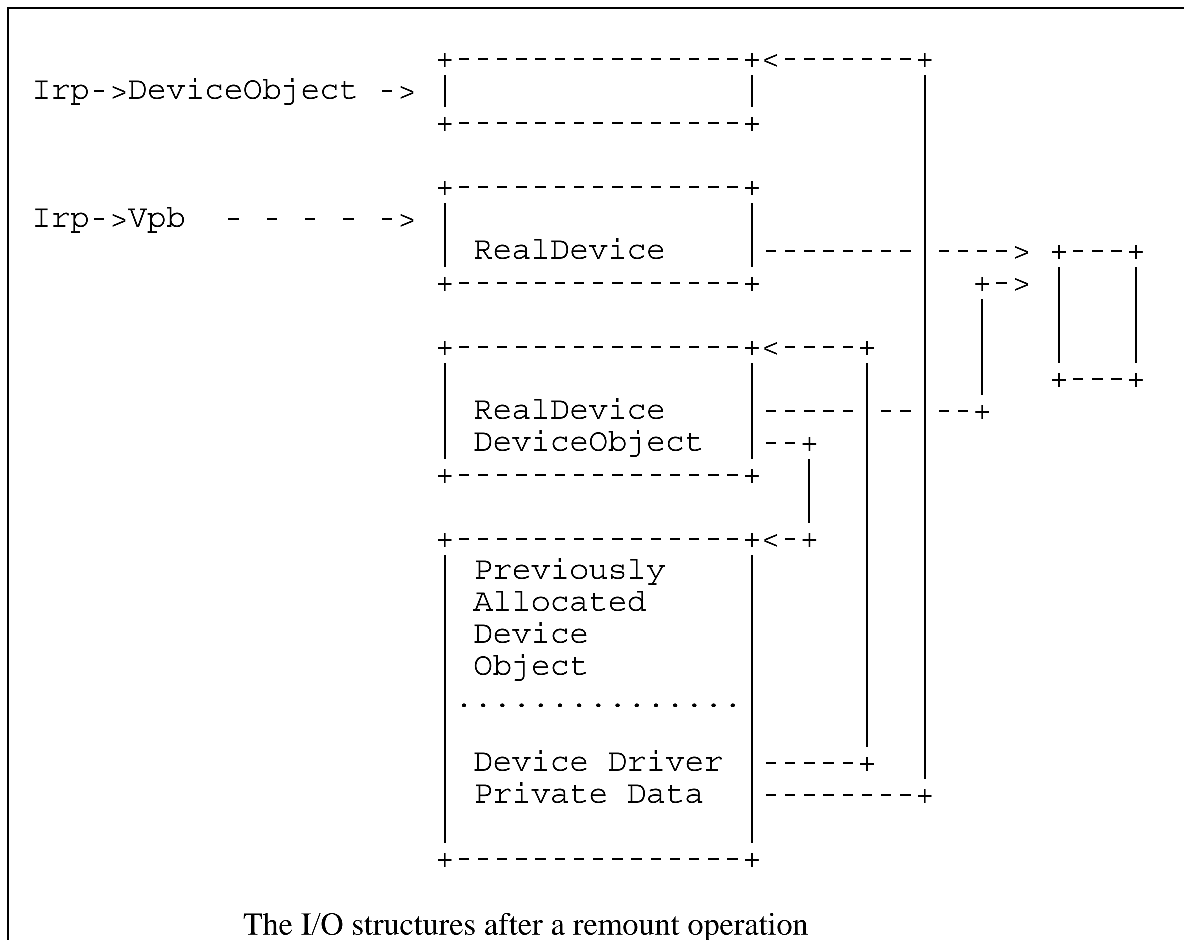
The I/O structures after a mount operation

In the preceding figure the newly allocated device object has immediately following it a Device Driver private data record that is for used only by the driver. This technique should be used in the driver to keep track of the VPB and the device object where it is to send its read and write requests. It should also be used to link together all of the mounted volumes serviced by the driver.

Remounting a volume:

By using the device driver private data record to maintain a link of all mounted volumes a Device Driver can determine if a mount request for a volume matches a previously mounted volume (They match if the both volume have the same serial number and volume label). The following figure shows the major I/O structure after processing a remount.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License



The I/O structures after a remount operation

The remount operation does not allocate any new structures, instead it it performs the following operations:

- o The Device Drivers Private Data pointer to the target device object is changed to point to the new target device object.

- o The RealDevice field of the Vpb that we previously mounted is set to the RealDevice field of the new Vpb that was passed in as a parameter in the IRP.

- o The Irp->Vpb is deallocated from pool by the device driver, and complete the mount request with STATUS_SUCCESS.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.9 File System Control(Query Information File System)

3.10 File System Control(Set Information File System)

3.11 File System Control(Unlock Volume)

3.12 File System Control(Verify Volume)

3.13 Internal Device Control

3.14 Lock Control(Lock)

3.15 Lock Control(Unlock All)

3.16 Lock Control(Unlock Single)

3.17 Query Acl

3.18 Query Ea

3.19 Query Information

3.20 Query Volume Information

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.21 Read

3.22 Read Terminal

3.23 Set Acl

3.24 Set Ea

3.25 Set Information

3.26 Set New Size

3.27 Set Volume Information

3.28 Write

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Original Draft 1.0, December 15, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Kernel Specification

Author: *David N. Cutler,*
Bryan M. Willman

Original Draft 1.0, March 8, 1989
Revision 1.1, March 16, 1989
Revision 1.2, March 29, 1989
Revision 1.3, April 18, 1989
Revision 1.4, May 4, 1989
Revision 1.5, May 8, 1989
Revision 1.6, August 14, 1989
Revision 1.7, November 15, 1989
Revision 1.8, November 16, 1989
Revision 1.9, November 17, 1989
Revision 1.10, January 6, 1990
Revision 1.11, June 6, 1990
Revision 1.12, September 19, 1990
Revision 1.13, March 11, 1991
Revision 1.14, May 2, 1991
Revision 1.15, May 28, 1991
Revision 1.16, June 18, 1991
Revision 1.17, August 7, 1991
Revision 1.18, August 8, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview 1
1.1 Kernel Execution Environment 1
1.2 Kernel Use of Hardware Priority Levels 2
1.3 Primary Kernel Data Structures 3
1.4 Multiprocessor Synchronization 4
1.4.1 Executive Multiprocessor Synchronization 6
1.4.1.1 Acquire Executive Spin Lock 6
1.4.1.2 Release Executive Spin Lock 7
1.5 Dispatching 7
1.5.1 Dispatcher Database 8
1.5.2 Idle Thread 8

2. Kernel Objects 9
2.1 Dispatcher Objects 9
2.1.1 Event Object 9
2.1.1.1 Initialize Event 11
2.1.1.2 Pulse Event 11
2.1.1.3 Read State Event 12
2.1.1.4 Reset Event 12
2.1.1.5 Set Event 12
2.1.2 Mutual Exclusion Objects 13
2.1.2.1 Mutant Object 14
2.1.2.1.1 Initialize Mutant 14
2.1.2.1.2 Read State Mutant 15
2.1.2.1.3 Release Mutant 15
2.1.2.2 Mutex Object 16
2.1.2.2.1 Initialize Mutex 17
2.1.2.2.2 Read State Mutex 17
2.1.2.2.3 Release Mutex 18
2.1.2.2.4 Mutex Contention Data 19
2.1.3 Semaphore Object 19
2.1.3.1 Initialize Semaphore 20
2.1.3.2 Read State Semaphore 20
2.1.3.3 Release Semaphore 20
2.1.4 Thread Object 21
2.1.4.1 Initialize Thread 24
2.1.4.2 Alert Thread 26
2.1.4.3 Alert and Resume Thread 27
2.1.4.4 Confine Thread 27
2.1.4.5 Delay Execution 28
2.1.4.6 Disable Queuing of APCs 29
2.1.4.7 Enable Queuing of APCs 29
2.1.4.8 Force Resumption of Thread 30

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.4.9 Freeze Thread 30
2.1.4.10 Query Data Alignment Mode 31
2.1.4.11 Query Base Priority 32
2.1.4.12 Read State Thread 32
2.1.4.13 Ready Thread 32
2.1.4.14 Resume Thread 33
2.1.4.15 Rundown Thread 33
2.1.4.16 Set Affinity Thread 34
2.1.4.17 Set Data Alignment Mode 34
2.1.4.18 Set Base Priority 35
2.1.4.19 Set Priority Thread 35
2.1.4.20 Suspend Thread 36
2.1.4.21 Terminate Thread 37
2.1.4.22 Test Alert Thread 37
2.1.4.23 Unfreeze Thread 38
2.1.4.24 Thread Performance Data 39
2.1.5 Timer Object 39
2.1.5.1 Initialize Timer 39
2.1.5.2 Cancel Timer 40
2.1.5.3 Read State Timer 40
2.1.5.4 Set Timer 40
2.2 Control Objects 41
2.2.1 Asynchronous Procedure Call (APC) Object 41
2.2.1.1 Initialize APC 43
2.2.1.2 Flush Queue APC 45
2.2.1.3 Insert Queue APC 46
2.2.1.4 Remove Queue APC 47
2.2.2 Deferred Procedure Call (DPC) Object 47
2.2.2.1 Initialize DPC 48
2.2.2.2 Insert Queue DPC 49
2.2.2.3 Remove Queue DPC 49
2.2.3 Device Queue Object 50
2.2.3.1 Initialize Device Queue 51
2.2.3.2 Insert Device Queue 51
2.2.3.3 Insert By Key Device Queue 52
2.2.3.4 Remove Device Queue 52
2.2.3.5 Remove Entry Device Queue 53
2.2.4 Interrupt Object 54
2.2.4.1 Initialize Interrupt 55
2.2.4.2 Connect Interrupt 58
2.2.4.3 Disconnect Interrupt 58
2.2.4.4 Synchronize Execution 59
2.2.5 Power Notify Object 60

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.5.1 Initialize Power Notify 61
2.2.5.2 Insert Power Notify 61
2.2.5.3 Remove Power Notify 62
2.2.6 Power Status Object 62
2.2.6.1 Initialize Power Status 63
2.2.6.2 Insert Power Status 63
2.2.6.3 Remove Power Status 64
2.2.7 Process Object 64
2.2.7.1 Initialize Process 65
2.2.7.2 Attach Process 66
2.2.7.3 Detach Process 67
2.2.7.4 Exclude Process 67
2.2.7.5 Include Process 68
2.2.7.6 Set Priority Process 68
2.2.7.7 Process Accounting Data 69
2.2.8 Profile Object 69
2.2.8.1 Initialize Profile 70
2.2.8.2 Start Profile 70
2.2.8.3 Stop Profile 71
2.2.8.4 Set System Profile Interval 71
2.2.8.5 Query System Profile Interval 72

3. Wait Operations 72
3.1 Wait For Multiple Objects 73
3.2 Wait For Single Object 75

4. Miscellaneous Operations 77
4.1 Bug Check 77
4.2 Context Frame Manipulation 78
4.2.1 Move Machine State To Context Frame 78
4.2.2 Move Machine State From Context Frame 79
4.3 Fill Entry Translation Buffer 80
4.4 Flush Data Cache 81
4.5 Flush Entire Translation Buffer 82
4.6 Flush Instruction Cache 82
4.7 Flush I/O Buffers 83
4.8 Flush Single Translation Buffer Entry 84
4.9 Freeze Execution 86
4.10 Get Current APC Environment 86
4.11 Get Current IRQL 86
4.12 Get Previous Mode 86
4.13 Lower IRQL 87
4.14 Query System Time 87
4.15 Raise IRQL 87

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.16 Run Down Thread 88
4.17 Set System Time........ 88
4.18 Stall Execution........ 89
4.19 Unfreeze Execution 89

5. Intel x86 Specific Functions. 90
5.1 Load an Ldt for a process. 90
5.2 Set and Entry in a Process's Ldt. 90
5.3 Get an Entry from a Thread's Gdt........ 91

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This specification describes the kernel layer of the **NT OS/2** operating system. The kernel is responsible for thread dispatching, multiprocessor synchronization, hardware exception handling, and the implementation of low-level machine dependent functions.

The kernel is used by the executive layer of the system to synchronize its activities and to implement the higher levels of abstraction that are exported in user-level API's.

Generally speaking, the kernel does not implement any policy since this is the province of the executive. However, there are some places where policy decisions are made by the kernel. These include the way in which thread priority is manipulated to maximize responsiveness to dispatching events (e.g., the input of a character from a keyboard).

The kernel executes entirely in kernel mode and is nonpageable. It guards access to critical data by raising the processor Interrupt Request Level (IRQL) to an appropriate level and then acquiring a spin lock.

The primary functions provided by the kernel include:

- Support of kernel objects
- Trap handling and exception dispatching
- Interrupt handling and dispatching
- Multiprocessor coordination and context switching
- Power failure recovery
- Miscellaneous hardware-specific functions

It is estimated that the kernel will be less than 48k bytes of resident nonpageable code exclusive of the IEEE exception handling code.

1.1 Kernel Execution Environment

The kernel executes in the most privileged processor mode, usually at an Interrupt Request Level (IRQL) of DISPATCH_LEVEL. The most privileged processor mode is termed kernel mode.

> \ *On the N10 and the x86 architectures the most privileged processor mode is called supervisor mode. However, in other architectures (e.g., MIPS), the*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

most privileged processor mode is not called supervisor mode. Furthermore, still other architectures include a supervisor mode, but it is not the most privileged mode. Therefore, since it is intended that NT OS/2 be portable and capable of running across several architectures, the most privileged processor mode will be referred to as kernel mode. \

The kernel can execute simultaneously on all processors in a multiprocessor configuration and synchronize access to critical regions as appropriate.

Software within the kernel is not preemptible and, therefore, cannot be context switched, whereas all software outside the kernel is almost always preemptible and context switchable. In general, executive software outside the kernel is not allowed to raise the IRQL above APC_LEVEL. However, device drivers and executive spin lock synchronization are exceptions to this rule.

The kernel is not pageable and cannot take page faults.

Software within the kernel is written in **C** and assembly language. Assembly language is used for:

- Trap handling
- Spin locks
- Context switching
- Interval timer interrupt
- Power failure interrupt
- Interprocessor interrupt
- I/O Interrupt dispatching
- Machine check processing
- Asynchronous Procedure Call dispatching
- Deferred Procedure Call dispatching
- A small piece of thread startup
- A small piece of system initialization

It is estimated that the number of lines of assembly code within the kernel will be less than 3k.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.2 Kernel Use of Hardware Priority Levels

Hardware Interrupt Request Levels (IRQL's) are used to prioritize the execution of the various kernel components. IRQL's are hierarchically ordered and each distinct level disables interrupts on lower levels while the respective level is active. The IRQL is raised when hardware and software interrupt requests are granted and by the kernel when synchronization with the possible occurrence of an interrupt is desired.

The kernel uses the hardware Interrupt Request Levels (IRQL's) as follows:

LOW_LEVEL - Thread execution

APC_LEVEL - Asynchronous Procedure Call interrupt

DISPATCH_LEVEL - Dispatch and Deferred Procedure Call interrupt

WAKE_LEVEL - Wake system debugger interrupt

Device levels - Device interrupts

CLOCK2_LEVEL - Interval timer clock interrupt

IPI_LEVEL - Interprocessor interrupt

POWER_LEVEL - Power failure interrupt

HIGH_LEVEL - Machine check and bus error interrupts

The level LOW_LEVEL is reserved for normal thread execution and enables all other interrupts.

The levels APC_LEVEL and DISPATCH_LEVEL are software interrupts and are requested only by the kernel itself. They are located below all hardware interrupt priority levels.

The level WAKE_LEVEL may or may not be present depending on the host hardware configuration and capabilities. It is intended for use in notifying the kernel debugger.

Device interrupt levels are generally placed between the levels WAKE_LEVEL and CLOCK2_LEVEL.

The levels CLOCK2_LEVEL, IPI_LEVEL, POWER_LEVEL, AND HIGH_LEVEL are the highest priority levels and are the most time critical.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

\ *The exact specification of interrupt levels is dependent on the host system architecture. The above discussion only defines the importance of the various levels, and does not attempt to assign a numeric value of each level.* \

1.3 Primary Kernel Data Structures

The primary kernel data structures include:

- Interrupt Dispatch Table (IDT) - This is a software maintained table that associates an interrupt source with an Interrupt Service Routine (ISR).
- Processor Control Registers (PCR's) - This is a set of four registers that appear in the same physical address on each processor in a multiprocessor configuration. These registers hold a pointer to the Processor Control Block (PRCB), a pointer to the current thread's Thread Environment Block (TEB), a pointer to the currently active thread, and a temporary location used by the trap handler to save the contents of the stack pointer. On a single processor implementation the PCR is located in main memory.
- Processor Control Block (PRCB) - This structure holds per processor information such as a pointer to the next thread selected for execution on the respective processor. There is a PRCB for each processor in a multiprocessor configuration. The address of this structure can always be obtained from a fixed virtual address on any processor.
- An array of pointers to PRCB's - This array is used to address the PRCB of another processor. It is used when another processor must be interrupted to performed some desired operation.
- Kernel objects - These are the data abstractions that are necessary to control processor execution and synchronization (e.g., thread object, mutex object, etc.). Functions are provided to initialize and manipulate these objects in a synchronized fashion.
- Dispatcher database - This is the database that is required to record the execution state of processors and threads. It is used by the thread dispatcher to schedule the execution of threads on processors.
- Timer queue - This is a list of timers that are due to expire at some future point in time. The timer queue is actually implemented as a splay tree (nearly balanced binary tree maintained by splay transformations).
- Deferred Procedure Call (DPC) queue - This is a list of requests to call a specified procedure when the IRQL falls below DISPATCH_LEVEL.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Power restoration notify and status queues - These are lists of power notify and status objects that are to be acted upon if power fails and is later restored without the contents of volatile memory being lost.

1.4 Multiprocessor Synchronization

At various stages during its execution, the kernel must guarantee that one, and only one, processor at a time is active within a given critical region. This is necessary to prevent code executing on one processor from simultaneously accessing and modifying data that is being accessed and modified from another processor. The mechanism by which this is achieved is called a *spin lock*.

Spin locks are used when mutual exclusion must exist across all processors and context switching cannot take place. A spin lock takes its name from the fact that, while waiting on the spin lock, software continually tries to gain entry to a critical region and makes no progress until it succeeds.

Spin locks are implemented with a test and set operation on a lock variable. When software executes a test and set operation and finds the previous state of the lock variable free, entry to the associated critical region is granted. If, however, the previous state of the lock variable is busy, then the test and set operation on the lock variable is simply repeated until the previous state is found to be free.

> \ *The exact instructions that are used to implement spin locks are processor architecture specific. In most architectures the test and set operation is not repeated continuously, but rather once finding the lock busy, ordinary instructions are used to poll the lock until it is free. Another test and set operation is then performed to retest the lock. This guarantees a minimum of bus contention during spin lock sequences.* \

Spin locks can only be operated on from a safe interrupt request level. This means that any attempt to acquire a particular spin lock must be at the highest IRQL from which any other attempt to acquire the same spin lock could be made on the same processor. If this restriction were not followed, then deadlock could occur when code running at a lower IRQL acquired a spin lock and then was interrupted by a higher-level interrupt whose Interrupt Service Routine (ISR) also attempted to acquire the spin lock.

The kernel uses various spin locks to synchronize access to the objects and data structures it supports. These include:

- Dispatcher Database - The dispatcher database describes the scheduling state of the system. Whenever a change is made to the dispatching state of the system (e.g., the occurrence of an event), the dispatcher database spin lock must be acquired at IRQL DISPATCH_LEVEL.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Power Restoration Notify Queue - The power restoration notify queue enables a device driver to be asynchronously notified when power is restored after a failure. Whenever an insertion or removal is made to/from this queue, the power notify queue spin lock must be acquired at IRQL DISPATCH_LEVEL.

- Power Restoration Status Queue - The power restoration status queue provides the capability to have a specified boolean variable set to a value of TRUE when power is restored after a failure. Whenever an insertion or removal is made to/from this queue, the power status queue spin lock must be acquired at IRQL POWER_LEVEL.

- Device Queues - A device queue is used to pass an I/O Request Packet (IRP) between a thread and a device driver. Whenever an insertion or removal is made to/from a device queue, the associated device queue spin lock must be acquired at IRQL DISPATCH_LEVEL.

- Interrupts - Each connected interrupt object has a spin lock that prevents the associated Interrupt Service Routine (ISR) from executing at the same time as other device driver code that accesses the same device resources. Whenever an interrupt occurs and the ISR executes, the associated spin lock must be acquired at the IRQL of the interrupting source. Likewise, device driver code must acquire the associated spin lock at the IRQL of the interrupting source when synchronization with the ISR is required.

- Processor Request Queue - Each processor has a request queue that is used by other processors to signal an action to be performed. Whenever an entry is inserted into or removed from this queue, the associated processor request queue spin lock must be acquired at IRQL IPI_LEVEL.

- Kernel Debugger - The kernel debugger is used to debug the kernel which can be in execution on several processors simultaneously. Whenever the debugger is entered, the kernel debugger spin lock must be acquired at IRQL HIGH_LEVEL.

> */ The actual implementation of spin locks may be optimized in a uniprocessor system. This could be done by either generating the system specifically for a uniprocessor system with conditionalized code or by dynamically modifying the code at boot time such that only IRQL is used to synchronize kernel execution. /*

1.4.1 Executive Multiprocessor Synchronization

Executive software outside the kernel also has the requirement to synchronize access to resources in a multiprocessor environment. Unlike the kernel, however,

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

executive software can use kernel dispatcher objects (e.g., mutexes, semaphores, events, etc.), as well as spin locks, to implement mutually exclusive access.

Kernel dispatcher objects allow the processor to be redispatched (context switched) and should be used when the wait or access time to a resource is liable to be lengthy (e.g. greater than 25 microseconds on an i860). Spin locks should be used when the wait or access time to a resource is short and does not involve complicated interactions with other code.

Executive spin locks could cause serious maintenance problems if not used judiciously. In particular, no deadlock protection is performed and dispatching is disabled while the executive owns a spin lock. Therefore, certain rules must be followed by executive software when using spin locks:

1. The code within a critical region that is guarded by an executive spin lock must not be pageable and must not make any references to pageable data.

2. An executive spin lock can only be acquired from IRQL's 0, APC_LEVEL, and DISPATCH_LEVEL.

3. The code within a critical region that is guarded by an executive spin lock cannot call any external procedures, nor can it generate any software conditions or hardware exceptions.

Programming interfaces that support executive spin locks include:

KeAcquireSpinLock - Acquire an executive spin lock
KeReleaseSpinLock - Release an executive spin lock.

1.4.1.1 Acquire Executive Spin Lock

An executive spin lock can be acquired with the **KeAcquireSpinLock** function:

VOID
KeAcquireSpinLock (
    IN PKSPIN_LOCK *SpinLock*,
    OUT PKIRQL *OldIrql*
    );

Parameters:

SpinLock - A pointer to an executive spin lock.

OldIrql - A pointer to a variable that receives the previous IRQL.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The previous IRQL is saved in the *OldIrql* parameter, the current IRQL is raised to DISPATCH_LEVEL, and the specified spin lock is acquired. The previous IRQL value must be specified when the spin lock is released.

1.4.1.2 Release Executive Spin Lock

An executive spin lock can be released with the **KeReleaseSpinLock** function:

VOID
KeReleaseSpinLock (
    IN PKSPIN_LOCK *SpinLock*,
    IN KIRQL *OldIrql*
    );

<u>Parameters:</u>

SpinLock - A pointer to an executive spin lock.

OldIrql - The IRQL at which the executive spin lock was acquired.

The specified spin lock is released and the current IRQL is set to the specified value.

1.5 Dispatching

The kernel dispatches threads for execution according to their software priority level.

There are 32 levels of thread priority which are split into two classes:

- Realtime
- Variable

The priority of threads within the realtime priority class is not altered by the kernel. However, as quantum end events occur, the threads within a level are round robin scheduled.

The priority of threads within the variable priority class is altered by the kernel, dependent on the execution profile of the respective threads. At each quantum end event, the priority of the executing thread is decremented and a decision is made as to whether it should be preempted by another thread. If it should be preempted to execute a higher priority thread, then a context switch occurs. When a thread in the variable priority class transitions from a Waiting state to a Ready state, it is given a priority boost that is dependent on the type of event that caused the Wait to be

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

satisifed. If the event was keyboard input, for instance, the thread would get a large boost. However, if the event was file I/O it would be given a smaller boost.

When a thread is readied for execution, an attempt is made to dispatch the thread on an idle processor. If an idle processor can be selected, then preference is given to the last processor on which the thread executed.

If an idle processor is not available, then an attempt is made to find a processor that should be preempted. This determination is made using the active summary and the active matrix (these structures are described in the following two sections). If an appropriate processor is located, then preference is given to the last processor on which the thread executed.

If no processor can be preempted to execute the ready thread, the thread is inserted at the end of the ready queue selected by its priority and the ready summary is updated.

Giving preference to the last processor a thread executed on maximizes the chances there is still thread data in the respective processor's secondary cache.

1.5.1 Dispatcher Database

The kernel maintains several data structures to aid in choosing which threads should be active at any instance in time. These structures include:

- Ready queues - There is a ready queue for each software priority level. Each queue contains a list of threads that are ready to execute at that level.

- Ready summary - A set that contains a TRUE member for each ready queue that contains one or more threads.

- Active matrix - The active matrix is a two-dimensional array that represents a set of processors for each of the software priority levels. A member is TRUE in the matrix if a processor is executing a thread at the corresponding priority level.

- Active summary - A set that contains a TRUE member for each priority class that has one or more processors executing threads at that level.

- Idle summary - A set that contains a TRUE member for each processor that is currently idle.

- Idle thread - A thread that is run when no other thread is available to execute on a processor.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The ready summary is used to quickly locate a thread to execute when the currently executing thread is terminated or transitions to a Waiting state.

The active summary is used to quickly determine if preemption should occur when a thread transitions to a Ready state.

> */ Since this determination is simple in a uniprocessor system, the active summary and active matrix are only kept up to date and used on configurations with multiple processors. /*

1.5.2 Idle Thread

Each processor has an idle thread that can always execute. The idle thread has a stack that is capable of nesting all interrupts and a software priority that is below that of all other thread priority levels.

The idle thread is selected for execution when no other thread is available to execute. The idle thread executes at DISPATCH_LEVEL and continually loops looking for work that has been assigned to its processor. This work includes processing the Deferred Procedure Call (DPC) queue and initiating a context switch when another thread is selected for execution on the respective processor.

While an idle thread executes, the member in the idle summary selected by its processor number is TRUE. This enables the kernel to quickly determine which processors are executing idle threads.

2. Kernel Objects

The kernel exports a set of abstractions to the executive layer which are called *kernel objects*. Kernel objects are represented by a control block that describes the contents of each object. Kernel objects are used by the executive layer to construct more complex objects that are exported in user level API's.

There are two types of kernel objects:

1. *Dispatcher* objects

2. *Control* objects

Dispatcher objects have a signal state (*Signaled* or *Not-Signaled*) and control the dispatching and synchronization of system operations. These objects include the *event*, *mutant, mutex*, *semaphore*, *thread*, and *timer* objects. Dispatcher objects can be specified as arguments to the kernel Wait functions.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Control objects are used to control the operation of the kernel but do not affect dispatching or synchronization. These objects include the *Asynchronous Procedure Call* (APC), *Deferred Procedure Call* (DPC), *device queue*, *interrupt*, *power notify*, *power status*, *process*, and *profile* objects.

The kernel neither allocates nor deallocates kernel object storage. It is the responsibility of the executive layer to allocate an appropriate data structure and call the kernel to initialize a specific kernel object type.

The kernel exports kernel object types to the executive layer so the executive can allocate appropriately sized data structures and can access various read only data items (e.g., linkage pointers).

The executive is not allowed to manipulate the writeable data portion of kernel objects directly. Various interfaces are provided by the kernel to perform this type of operation.

Kernel objects are referenced by pointers to the respective data structures. It is the responsibility of the executive layer to synchronize the deallocation of kernel object storage such that the kernel does not access an object after its storage has been deleted.

2.1 Dispatcher Objects

This section describes the various types of dispatcher objects and the interfaces that are provided to manipulate these objects.

2.1.1 Event Object

An *event object* is used to record the occurrence of an event and synchronize it with some action that is to be performed.

There are two types of event objects:

- *synchronization*
- *notification*

A synchronization event object is used when it is desirable for a single waiter to continue execution when the event is set to the Signaled state. The state of a synchronization event object is automatically reset to the Not-Signaled state when a Wait for the event object is satisfied.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Synchronization events provide an optimal way to implement mutual exclusion at user level. An identical capability can be implemented using binary semaphores, but requires calling the system each time mutual exclusion is desired.

Synchronization events can also be used to provide synchronization in producer/consumer relationships where it is otherwise undesirable to use a counting semaphore.

A notification event is used when it is desirable for all waiters to continue execution when the event is set to the Signaled state. The state of a notification event object is not altered when a Wait for the event object is satisfied and remains Signaled until it is explicitly reset to the Not-Signaled state.

Notification events can be used to implement resource allocators where there is not a one-to-one relationship between the release of a resource and the allocation of the resource (e.g. a memory allocator).

Waiting on an event object causes the execution of the subject thread to be suspended until the event object attains a state of Signaled.

Satisfying the Wait for a synchronization event object automatically causes the state of the event object to be reset to the Not-Signaled state.

Satisfying the Wait for a notification event object does not cause the state of the event object to change. Therefore, when a notification event object attains a state of Signaled, an attempt is made to satisfy as many Waits as possible.

The state of an event object is controlled by a count value that is incremented each time the event object is set to the Signaled state. Thus, the state of the event object is Signaled when the count value is nonzero and Not-Signaled when the count value is zero.

The count value indicates the number of times that the event object has been set to a Signaled state since the last time it was reset to the Not-Signaled state.

Programming interfaces that support the event object include:

KeInitializeEvent - Initialize an event object
KePulseEvent - Set/reset event object state atomically
KeReadStateEvent - Read state of event object
KeResetEvent - Set event object to Not-Signaled state
KeSetEvent - Set event object to Signaled state

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.1.1 Initialize Event

An event object can be initialized with the **KeInitializeEvent** function:

VOID
KeInitializeEvent (
    IN PKEVENT *Event*,
    IN EVENT_TYPE *EventType*,
    IN BOOLEAN *State*
    );

Parameters:

Event - A pointer to a dispatcher object of type event.

EventType - The event type (*NotificationEvent* or *SynchronizationEvent*).

State - The initial state of the event.

The event object data structure for the specified event type is initialized with the specified initial state.

2.1.1.2 Pulse Event

An event object can be atomically set to a Signaled state and then reset to a Not-Signaled state with the **KePulseEvent** function:

LONG
KePulseEvent (
    IN PKEVENT *Event*,
    IN KPRIORITY *Increment*,
    IN BOOLEAN *Wait*
    );

Parameters:

Event - A pointer to a dispatcher object of type event.

Increment - The priority increment that is to be applied if setting the event causes a Wait to be satisfied.

Wait - A boolean value that specifies whether the call to **KePulseEvent** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The function atomically sets the state of the event object to Signaled, attempts to satisfy as many Waits as possible for the event object, and then resets the state of the event object to Not-Signaled.

The previous state of the event object is returned as the function value. If the previous state of the event object was Signaled, then a nonzero count value is returned. Otherwise, a value of zero is returned.

If the *wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Thus the call to **KePulseEvent MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to set an event and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.1.3 Read State Event

The current state of an event object can be read with the **KeReadStateEvent** function:

LONG
KeReadStateEvent (
    IN PKEVENT *Event*
    );

Parameters:

Event - A pointer to a dispatcher object of type event.

The current state of the event object is returned as the function value. If the current state of the event object is Signaled, then a nonzero count value is returned. Otherwise, a value of zero is returned.

2.1.1.4 Reset Event

An event object can be reset to a Not-Signaled state with the **KeResetEvent** function:

LONG
KeResetEvent (
    IN PKEVENT *Event*
    );

Parameters:

Event - A pointer to a dispatcher object of type event.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The previous state of the event object is returned as the function value and the state of the event object is reset to Not-Signaled (i.e., the count value is set to zero). If the previous state of the event object was Signaled, then a nonzero count value is returned. Otherwise, a value of zero is returned.

2.1.1.5 Set Event

An event object can be set to a Signaled state with the **KeSetEvent** function:

LONG
KeSetEvent (
  IN PKEVENT *Event*,
  IN KPRIORITY *Increment*,
  IN BOOLEAN *Wait*
  );

Parameters:

Event - A pointer to a dispatcher object of type event.

Increment - The priority increment that is to be applied if setting the event causes a Wait to be satisfied.

Wait - A boolean value that specifies whether the call to **KeSetEvent** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

The previous state of the event object is returned as the function value and the state of the event is set to Signaled (i.e., the count value is incremented). If the previous state of the event object was Signaled, then a nonzero count value is returned. Otherwise, a value of zero is returned.

Setting an event object causes the event to attain a Signaled state, and therefore, an attempt is made to satisfy as many Waits as possible for the event object.

If the *Wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spinlock. Thus the call to **KeSetEvent MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to set an event and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.2 Mutual Exclusion Objects

The kernel provides two objects for controlling mutually exclusive access to a resource; the *mutant object* and the *mutex object*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The *mutant object* is intended for use in providing a user mode mutual exclusion mechanism that has ownership semantics, but it can also be used in kernel mode.

The *mutex object* can only be used in kernel mode and is intended to provide a deadlock-free mutual exclusion mechanism with ownership and other special system semantics.

The state of mutant and mutex objects is controlled by a count. When the count is one, the mutant or mutex object is in the Signaled state, the mutant or mutex object is not owned, and exclusive access to the corresponding resource can be obtained by specifying the mutant or mutex object in a kernel Wait function. When the count is not one, the mutant or mutex object is in the Not-Signaled state and any attempt to acquire the mutant or mutex object will cause the subject thread to wait until the mutant or mutex object count is one.

Mutant and mutex objects are similar in that they both provide mutual exclusion mechanisms with recursive ownership capability. They have significant differences, however, which dictate the support of two separate object types. These differences include the following:

- o Mutex objects have a level number which is used to prevent deadlock, whereas mutant objects have no level number.
- o Mutant objects have an abandoned status and can be released by a thread other than the owner, whereas mutex objects do not have an abandoned status and can only be released by the owner thread.
- o Owning a mutex object prevents the owning thread's process from leaving the balance set, whereas owning a mutant object does not affect the swapability of the parent process.
- o Owning a mutex object causes the priority of the owning thread to be raised to the greater of its current priority and the lowest realtime priority, whereas owning a mutant object does not affect the owner thread's priority in any way.
- o Owning a mutex object prevents the delivery of kernel mode APCs, whereas owning a mutant object does not affect the delivery of kernel mode APCs.

2.1.2.1 Mutant Object

Waiting on (acquiring) a mutant object causes the execution of the subject thread to be suspended until the mutant object attains a state of Signaled. Satisfying the Wait for a mutant object causes the state of the mutant object to become Not-Signaled.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Threads are allowed to recursively acquire mutant objects that they already own. A recursively acquired mutant object must be released the same number of times it was acquired before it will again attain the state of Signaled.

Mutant objects can be exported to user mode for providing a mutual exclusion mechanism with ownership semantics.

Programming interfaces that support the mutant object include:

KeInitializeMutant - Initialize a mutant object
KeReadStateMutant - Read the state of a mutant object
KeReleaseMutant - Release ownership of a mutant object

2.1.2.1.1 Initialize Mutant

A mutant object can be initialized with the **KeInitializeMutant** function:

VOID
KeInitializeMutant (
    IN PKMUTANT *Mutant*,
    IN BOOLEAN *InitialOwner*
    );

Parameters:

Mutant - A pointer to a dispatcher object of type mutant.

InitialOwner - A boolean variable that determines whether the current thread is to be the initial owner of the mutant object.

If the value of the *InitialOwner* parameter is TRUE, then the mutant object data structure is initialized with the current thread as the owner and an initial state of Not-Signaled. Otherwise, the mutant object data structure is initialized as unowned with an initial state of Signaled.

2.1.2.1.2 Read State Mutant

The current state of a mutant object can be read with the **KeReadStateMutant** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
LONG
KeReadStateMutant (
    IN PKMUTANT Mutant
    );
```

Parameters:

Mutant - A pointer to a dispatcher object of type mutant.

The current state of the mutant object is returned as the function value. If the return value is one, then the state of the mutant object is Signaled. Otherwise, the state of the mutant object is Not-Signaled.

2.1.2.1.3 Release Mutant

A mutant object can be released with the **KeReleaseMutant** function:

```
LONG
KeReleaseMutant (
    IN PKMUTANT Mutant,
    IN KPRIORITY Increment,
    IN BOOLEAN Abandoned,
    IN BOOLEAN Wait
    );
```

Parameters:

Mutant - A pointer to a dispatcher object of type mutant.

Increment - The priority increment that is to be applied if releasing the mutant object causes a Wait to be satisfied.

Abandoned - A boolean value that specifies whether the release of the mutant object is to be forced.

Wait - A boolean variable that specifies whether the call to **KeReleaseMutant** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

If the value of the *Abandoned* parameter is TRUE, then the release of the mutant object is unconditional and can be requested by a thread other than the owner of the mutant object. The fact that the mutant object is being abandoned is recorded in the mutant object and is returned by the kernel Wait services when ownership of the mutant object is granted to another thread. Once set, the abandoned status of a

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

mutant object cannot be cleared and is thereafter always returned by the kernel Wait services.

If the value of the *Abandoned* parameter is FALSE, then only the owning thread can release the mutant object. Any attempt to release the mutant object made by a thread other than the owner, will cause an exception to be raised. If the mutant object has previously been abandoned, then the exception STATUS_ABANDONED is raised. Otherwise, the exception STATUS_MUTANT_NOT_OWNED is raised.

The previous state of the mutant object is returned as the function value.

If the value of the *Abandoned* parameter is TRUE, then the state of the mutant object is set to Signaled and the return value is not meaningful.

If the value of the *Abandoned* parameter is FALSE, then the new state of the mutant object can be determined by the value returned. If the returned value is zero, then the mutant object was actually released and attained a state of Signaled. Otherwise, the mutant object was not released and still has a state of Not-Signaled (i.e., the mutant object has been recursively acquired and has not yet been released the proper number of times to cause it to attain a Signaled state).

If the mutant object attains a Signaled state, then an attempt is made to satisfy a Wait for the mutant object.

If the mutant object attains a Signaled state and was previously owned by a thread, then the mutant object is removed from the list of mutant objects owned by the subject thread.

If the value of the *Wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Thus the call to **KeReleaseMutant MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to release a mutant object and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.2.2 Mutex Object

Waiting on (acquiring) a mutex object causes the execution of the subject thread to be suspended until the mutex object attains a state of Signaled. Satisfying the Wait for a mutex object causes the state of the mutex object to become Not-Signaled and disables the delivery of normal kernel APCs to the subject thread.

If the subject thread did not previously own any mutexes, then the current execution priority of the thread is saved and then raised to the maximum of its current priority and the lowest realtime priority. This ensures that the thread will

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

have a very high execution priority while it executes in a critical section and prevents the effects of priority inversion. When the thread releases the last mutex it owns, its priority is restored to the saved value.

Mutex ownership also prevents the owning thread's process from being removed from the balance set. If the balance set manager selects the process for removal from the balance set, all threads within the process that own mutexes will be allowed to continue execution until they no longer own any mutexes. This ensures that access to critical resources is not blocked because a thread belonging to a process that has been removed from the balance set owns one or more mutexes.

Each mutex object has a level number. This level number is used to prevent possible deadlock. When an attempt is made to acquire a mutex object, the level number of the mutex object must be higher (numerically) than the highest level number of any mutex object owned by the subject thread. If this condition is not met, then a system bug check occurs.

Level number checking is included mainly for debugging the system while it is under development. It may or may not be conditionalized out in a production system.

Threads are allowed to recursively acquire mutex objects that they already own. For this case level number checking does not occur since deadlock is not possible. A recursively acquired mutex object must be released the same number of times it was acquired before it will again attain the state of Signaled.

Mutex objects are not exported by the executive to user mode and are only available for use by the executive layer itself. Furthermore, the executive is not allowed to acquire a mutex object and then return to user mode while retaining ownership of the mutex.

Programming interfaces that support the mutex object include:

KeInitializeMutex - Initialize a mutex object
KeReadStateMutex - Read state of mutex object
KeReleaseMutex - Release ownership of mutex object

2.1.2.2.1 Initialize Mutex

A mutex object can be initialized with the **KeInitializeMutex** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
VOID
KeInitializeMutex (
    IN PKMUTEX Mutex,
    IN ULONG Level
    );
```

Parameters:

Mutex - A pointer to a dispatcher object of type mutex.

Level - The level number that is to be assigned to the mutex.

The mutex object data structure is initialized with the specified level number and an initial state of Signaled.

2.1.2.2.2 Read State Mutex

The current state of a mutex object can be read with the **KeReadStateMutex** function:

```
LONG
KeReadStateMutex (
    IN PKMUTEX Mutex
    );
```

Parameters:

Mutex - A pointer to a dispatcher object of type mutex.

The current state of the mutex object is returned as the function value. If the return value is one, then the state of the mutex object is Signaled. Otherwise, the state of the mutex object is Not-Signaled.

2.1.2.2.3 Release Mutex

A mutex object can be released with the **KeReleaseMutex** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

LONG
KeReleaseMutex (
  IN PKMUTEX *Mutex*,
  IN BOOLEAN *Wait*
  );

<u>Parameters:</u>

Mutex - A pointer to a dispatcher object of type mutex.

Wait - A boolean variable that specifies whether the call to **KeReleaseMutex** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

The previous state of the mutex object is returned as the function value. The state is returned as an integer value. If the returned value is zero, then the mutex object was actually released and attained a state of Signaled. Otherwise, the mutex object was not released and still has a state of Not-Signaled (i.e the mutex object has been recursively acquired and has not yet been released the proper number of times to cause it to attain a Signaled state).

If the mutex object attains a Signaled state, then an attempt is made to satisfy a Wait for the mutex object.

A mutex object can only be released by the subject thread that owns the mutex. If an attempt is made to release a mutex that is not owned by the subject thread, then a bug check will occur.

A mutex object can only be released if it is currently owned. An attempt to release a mutex object whose current state is Signaled, will also cause a bug check to occur.

If the mutex object attains a Signaled state, then the mutex object is removed from the list of mutexes owned by the subject thread. If the thread's owned mutex list does not contain any more entries, then the thread's original priority is restored (the priority that was previously saved) and a kernel APC is requested if the thread's kernel APC queue contains one or more entries.

If the mutex object attains a Signaled state, the mutex was the last one owned by the subject thread, and the thread's process has been selected for removal from the balance set, then a new thread is selected for execution, the subject thread is inserted into its process's ready queue, and a context switch to the selected thread is performed. If no other threads in the process own mutexes, then the balance set event is set in the process object to notify the balance set manager that it can remove the process from the balance set.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the value of the *Wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Thus the call to **KeReleaseMutex MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to release a mutex and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.2.2.4 Mutex Contention Data

Two counters are maintained for each mutex object to determine the activity level of the mutex object and any contention that may occur. One of the counters records the number of times the mutex object has been acquired and the other the number of times an attempt to acquire the mutex object resulted in the execution of the subject thread being suspended.

2.1.3 Semaphore Object

A *semaphore object* is used to control access to a resource, but not necessarily in a mutually exclusive fashion. A semaphore object acts as a gate through which a variable number of threads may pass concurrently, up to a specified limit. The gate is open (Signaled state) as long as there are resources available. When the number of resources specified by the limit are concurrently in use, the gate is closed (Not-Signaled state).

The gating mechanism of a semaphore object is controlled by a count. When the count is greater than zero, the semaphore object is in the Signaled state, and one or more threads may pass through the gate by specifying the semaphore in a kernel Wait function. When the count is zero, the semaphore object is in the Not-Signaled state, the gate is closed, and any attempt to pass through the gate will cause the subject thread to Wait until the semaphore count is greater than zero.

Waiting on (acquiring) a semaphore object causes the execution of the subject thread to be suspended until the semaphore object attains a Signaled state. Satisfying the Wait for a semaphore object causes the semaphore count to be decremented.

A semaphore object with a limit of one can be used to provide mutual exclusion semantics since only one thread will be allowed to pass through the gate concurrently. This is not, however, the same functionality as provided by mutex objects since there is no ownership (i.e., any thread can release the semaphore), there is no level number checking (i.e., deadlock is not prevented), and the priority of the subject thread is not raised (i.e., priority inversion problems can arise).

A semaphore object with a limit of one can also be used as a "*synchronization*" event provided the semaphore is never "over Signaled" (i.e., no thread attempts to release

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

the semaphore while it is already in the Signaled state). For this case, the semaphore is normally in a Not-Signaled state and is used to record the occurrence of an event by releasing the semaphore. Waiting on the semaphore object suspends the subject thread until the semaphore attains a Signaled state and causes the semaphore to be immediately set to the Not-Signaled state.

Programming interfaces that support the semaphore object include:

KeInitializeSemaphore - Initialize a semaphore object
KeReadStateSemaphore - Read state of semaphore
KeReleaseSemaphore - Adjust semaphore object count

2.1.3.1 Initialize Semaphore

A semaphore object can be initialized with the **KeInitializeSemaphore** function:

VOID
KeInitializeSemaphore (
    IN PKSEMAPHORE *Semaphore*,
    IN LONG *Count*,
    IN LONG *Limit*
    );

Parameters:

Semaphore - A pointer to a dispatcher object of type semaphore.

Count - The initial count value to be assigned to the semaphore. This value must be positive.

Limit - The maximum count value that the semaphore can attain. This value must be positive.

The semaphore object data structure is initialized with the specified initial count and limit.

2.1.3.2 Read State Semaphore

The current state of a semaphore object can be read with the **KeReadStateSemaphore** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
LONG
KeReadStateSemaphore (
    IN PKSEMAPHORE Semaphore
    );
```

Parameters:

Semaphore - A pointer to a dispatcher object of type semaphore.

The current signal state of the semaphore object is returned as the function value. If the return value is zero, then the current state of the semaphore object is Not-Signaled. Otherwise, the current state of the semaphore object is Signaled.

2.1.3.3 Release Semaphore

A semaphore object can be released with the **KeReleaseSemaphore** function:

```
LONG
KeReleaseSemaphore (
    IN PKSEMAPHORE Semaphore,
    IN KPRIORITY Increment,
    IN LONG Adjustment,
    IN BOOLEAN Wait
    );
```

Parameters:

Semaphore - A pointer to a dispatcher object of type semaphore.

Increment - The priority increment that is to be applied if releasing the semaphore causes a Wait to be satisfied.

Adjustment - The value that is to be added to the current semaphore count. This value must be positive.

Wait - A boolean value that specifies whether the call to **KeReleaseSemaphore** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

Releasing a semaphore object causes the semaphore count to be augmented by the value of the *Adjustment* parameter. If the resultant value is greater than the limit of the semaphore object, then the count is not adjusted and the exception STATUS_SEMAPHORE_COUNT_EXCEEDED is raised.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Augmenting the semaphore object count causes the semaphore to attain a state of Signaled, and therefore, an attempt is made to satisfy as many Waits as possible for the semaphore object.

The previous state of the semaphore object is returned as an integer function value. If the return value is zero, then the previous state of the semaphore object was Not-Signaled. Otherwise, the previous state of the semaphore object was Signaled.

If the value of the *wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Thus the call to **KeReleaseSemaphore MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to release a semaphore and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.4 Thread Object

A *thread object* is the agent that executes program code and is dispatched for execution by the kernel.

Each thread is associated with a *process object* which specifies the virtual address space mapping for the thread and accumulates thread execution time. Several thread objects can be associated with a single process object which enables the concurrent execution of multiple threads in a single address space (possibly simultaneous execution in a multiprocessor system).

A thread executes in kernel and user mode, usually at IRQL 0, and is dispatched for execution according to its software priority.

Although there is no actual difference, threads are usually referred to as either user threads or system threads. A user thread executes mostly in user mode and within the user part of the virtual address space. It enters kernel mode only to execute system services. System threads execute only in kernel mode and usually within the system part of the virtual address space. There are some system threads, however, that also use the user part of the address space to store information and execute code from. An example of such a thread is a file system.

The context of a thread typically consists of the following:

- Integer registers
- Floating point registers
- Architecture-dependent special registers

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- A user stack pointer
- A kernel stack pointer
- A program counter
- A processor status
- A floating point status

\ *The exact context of a thread is host architecture dependent.* \

Each thread has a set of processors on which it can execute. This is referred to as the processor affinity. When a thread is initialized it is given the processor affinity of its parent process. Thereafter, the affinity of the thread can be set to any proper subset of the parent process's affinity.

A thread can be in one of six dispatcher, or scheduling, states:

- *Initialized*
- *Ready*
- *Standby*
- *Running*
- *Waiting*
- *Terminated*

A thread enters the *Initialized* state when its thread object is initialized. A thread in the Initialized state can transition only to the Ready state.

A thread is in a *Ready* state when it is eligible to be selected for execution on a processor. A thread in the Ready state is enqueued on the dispatcher ready queue selected by its priority and can transition from the Ready state to the Standby state.

A thread is in a *Standby* state when it has been selected to execute on a processor, but the actual context switch to the thread has not yet occurred. A thread in the Standby state can transition to the Ready and Running states.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

A thread is in a *Running* state when it is currently being executed by a processor. A thread in the Running state can transition to the Ready, Waiting, and Terminated states.

A thread is in a *Waiting* state when it is waiting for one or more dispatcher objects to attain a state of Signaled. A thread in the Waiting state can transition to the Ready state.

A thread in the *Terminated* state has completed its execution and the corresponding thread object will be deleted by the executive at the appropriate time.

> */ Note that is possible to reuse a thread object that has a state of Terminated by simply reinitializing the thread object which will cause it to enter the Initialized state. /*

A thread's parent process is either in the balance set (Included) or not in the balance set (Excluded). The balance set is that set of processes and threads that are currently eligible for being considered for execution. Processes and threads that are not in the balance set are not considered for execution until they reenter the balance set.

The balance set is managed by the balance set manager. In a single user system there will be no balance set manager. Server systems, however, present the problem of having to manage more processes than there is space for in main memory without incurring excessive paging. Therefore, the balance set manager is responsible for determining when excessive paging is occurring and then selecting the appropriate processes to remove from the balance set.

> */ There may not be a balance set manager in the first release of NT OS/2. We may rely instead on working set trimming to obtain necessary memory when excessive paging levels are observed. /*

A thread is dispatched for execution according to its software priority level. Higher priority threads are given preference and preempt the execution of lower priority threads.

There are two classes of priority: 1) realtime, and 2) variable. Each of these classes contains several levels of thread priority.

In the realtime priority class, a thread executes at a priority level selected by the user. The system makes no attempt to alter or boost the priority as the thread executes and enters and leaves wait states. The realtime priority levels are higher than all the levels in the variable priority class. The realtime priority class is intended for use by time-critical threads that require a response time that is guaranteed by application design.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The variable priority class is where most threads execute. As a thread executes and experiences quantum end events, its priority decays. When a thread enters a Waiting state and is subsequently awakened, it is given a priority boost that is commensurate with the importance of the event that caused the Wait to be satisifed (e.g., a large boost is given for the completion of keyboard input but a small one is given for completing disk I/O). A thread therefore, runs at a high priority as long as it is interactive. When it becomes compute bound, its priority rapidly decays, and it is considered only after other, higher priority threads. In addition, the kernel arbitrarily boosts the priority of threads that are compute bound and haven't received any processor time for a given period of time.

As a thread executes, performance and accounting data are collected for the thread and for the thread's parent process.

Programming interfaces that support the thread object include:

KeInitializeThread - Initialize a thread object
KeAlertThread - Set thread alert for specified mode
KeAlertResumeThread - Alert and resume thread object
KeConfineThread - Confine thread object execution
KeDelayExecutionThread - Delay execution of thread object
KeDisableApcQueuingThread - Disable queuing of APCs
KeEnableApcQueuingThread - Enable queuing of APCs
KeForceResumeThread - Force resumption of thread execution
KeFreezeThread - Freeze thread object execution
KeQueryAutoAlignmentThread - Query alignment mode of thread object
KeQueryBasePriorityThread - Query base priority of thread object
KeReadStateThread - Read state of thread object
KeReadyThread - Ready thread object for execution
KeResumeThread - Resume thread object execution
KeRundownThread - Run down thread object
KeSetAffinityThread - Set thread object processor set
KeSetAutoAlignmentThread - Set alignment mode of thread object
KeSetBasePriorityThread - Set base priority of thread object
KeSetPriorityThread - Set priority of thread object
keSuspendThread - Suspend thread object exection
KeTerminateThread - Terminate thread object execution
KeTestAlertThread - Test if thread alerted for mode
KeUnfreezeThread - Unfreeze thread object execution

2.1.4.1 Initialize Thread

A thread object can be initialized with the **KeInitializeThread** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
VOID
KeInitializeThread (
    IN PKTHREAD Thread,
    IN PVOID KernelStack,
    IN PKSYSTEM_ROUTINE SystemRoutine,
    IN PKSTART_ROUTINE StartRoutine OPTIONAL,
    IN PVOID StartContext OPTIONAL,
    IN PCONTEXT ContextFrame OPTIONAL,
    IN PVOID Teb OPTIONAL,
    IN PKPROCESS Process
    );
```

Parameters:

Thread - A pointer to a dispatcher object of type thread.

KernelStack - A pointer to the base (highest address) of a kernel stack on which the initial context for the thread is to be constructed.

SystemRoutine - A pointer to a function that is to called when the thread is scheduled for execution. This routine performs executive initialization.

StartRoutine - An optional pointer to a function that is to be called after the executive has finished initializing the thread.

StartContext - A optional pointer to an arbitrary data structure which will be passed to the *StartRoutine* function as a parameter.

ContextFrame - An optional pointer to a context frame which contains the initial user mode state of the thread. This parameter is specified if the thread will execute in user mode. If this parameter is not specified, then the *Teb* parameter is ignored.

Teb - An optional pointer to the user mode thread environment block. This parameter is ignored if the *ContextFrame* parameter is not specified.

Process - A pointer to a control object of type process.

The function specified by the *SystemRoutine* parameter has the following type definition:

```
typedef
VOID
(*PKSYSTEM_ROUTINE) (
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
IN PKSTART_ROUTINE StartRoutine OPTIONAL,
IN PVOID StartContext OPTIONAL
);
```

Parameters:

StartRoutine - An optional pointer to a function that is to be called after the executive has finished initializing the thread.

StartContext - A optional pointer to an arbitrary data structure which will be passed to the *StartRoutine* function as a parameter.

The function specified by the *StartRoutine* parameter has the following type definition:

```
typedef
VOID
(*PKSTART_ROUTINE) (
    IN PVOID StartContext
    );
```

Parameters:

StartContext - A pointer to an arbitrary data structure that was specified when the thread object was initialized.

The thread object data structure is initialized and the thread's dispatcher state is set to Initialized. The thread's quantum, affinity, data alignment handling mode, current priority, and base priority are taken from the parent process object.

A kernel context frame is built on the specified kernel stack which will cause the thread to begin execution in the kernel thread startup routine. The kernel thread startup routine will call the specified start routine which is responsible for initializing the executive state of the thread as necessary. If the thread is a system thread, then the executive startup routine will call the thread's entry point directly. If, however, the thread is a user thread, then the executive startup routine returns control to the kernel thread startup routine which restores the user mode state and continues execution of the thread in user mode.

A thread begins execution in kernel mode at IRQL APC_LEVEL with the queuing of APCs enabled. It is the responsibility of the executive to lower IRQL to 0 as soon as thread initialization is complete.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Once a thread object has been initialized, it can be readied for execution with the **KeReadyThread** function.

2.1.4.2 Alert Thread

A thread object can be alerted with the **KeAlertThread** function:

BOOLEAN
KeAlertThread (
    IN PKTHREAD *Thread*,
    IN KPROCESSOR_MODE *AlertMode*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

AlertMode - The processor mode (*UserMode* or *KernelMode*) for which the thread is to be alerted.

Alerting a thread object causes the alert variable associated with the specified processor mode to be set to a value of TRUE.

If the thread object is currently in a Wait state, the Wait is alertable, and the specified processor mode is less than or equal to the Wait mode, then the thread is Unwaited with a completion status of *STATUS_ALERTED* and the specified alert variable is set to a value of FALSE.

Alerts provide a way in which to break into a thread's execution at well-defined points. These points occur when the thread Waits in an alertable state and when the thread polls the alerted flag using the **KeTestAlertThread** function.

The previous value of the alert variable for the specified processor mode is returned as the function value. If the return value is TRUE, then the subject thread was already alerted for the specified processor mode. If the return value is FALSE, then the subject thread was not previously alerted.

2.1.4.3 Alert and Resume Thread

A thread object can be kernel mode alerted and its execution resumed with the **KeAlertResumeThread** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG
KeAlertResumeThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

This function executes the equivalent of a **KeAlertThread** function for kernel mode followed by a **KeResumeThread** function on the specified thread object.

Resuming a thread object checks the suspend count of the subject thread. If the suspend count is zero, then the thread is not currently suspended and no operation is performed. Otherwise, the subject thread's suspend count is decremented. If the resultant value is zero, then the execution of the subject thread is resumed by releasing its builtin suspend semaphore.

The previous suspend count is returned as the function value. If the return value is zero, then the subject thread was not previously suspended. If the return value is one, then the subject thread's execution was resumed. If the returned value is not zero or one, then the subject thread is still suspended and must be resumed the number of times specified by the return value minus one before it will actually resume execution.

2.1.4.4 Confine Thread

The execution of the current thread can be confined to the current processor with the **KeConfineThread** function:

KAFFINITY
KeConfineThread (
    );

Confining the execution of the current thread to the current processor causes the thread's affinity to be set such that it can only execute on the current processor. The previous affinity is returned as the function value and can be used to later restore the thread's affinity with the **KeSetAffinityThread** function.

This function is useful when it is desirable to avoid translation buffer flushes across the entire multiprocessor complex while certain page manipulations are taking place. For example, the zero page writer selects a page to zero, confines its execution to the current processor, flushes the current processor's translation buffer, maps the page into the system part of the virtual address space reserved for zeroing

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

pages, and then proceeds to zero the page. If the execution of the zero page writer was not confined during the page zeroing operation, then the translation buffers of all processors in the multiprocessor complex would have to be flushed before mapping and zeroing the page could commence.

2.1.4.5 Delay Execution

The execution of the current thread can be delayed for a specified interval of time with the **KeDelayExecutionThread** function:

NTSTATUS
KeDelayExecutionThread (
    IN KPROCESSOR_MODE *WaitMode*,
    IN BOOLEAN *Alertable*,
    IN PTIME *Interval*
    );

Parameters:

WaitMode - The processor mode on whose behalf the Wait is occurring.

Alertable - A boolean value that specifies whether the Wait is alertable.

Interval - The absolute or relative time over which the Wait is to occur.

The expiration time is computed and the current thread is put in a Waiting state. When the specified interval of time has passed, the thread will exit the Waiting state and continue execution.

The reason for the Wait is set to DelayExecution.

The *WaitMode* parameter specifies on whose behalf the Wait is occurring (i.e., kernel or user).

The *Alertable* parameter specifies whether the thread can be alerted while it is in the Waiting state. If the value of this parameter is TRUE and the thread is alerted for a mode that is equal to or more privileged than the Wait mode, then the thread's Wait will be satisfied with a completion status of STATUS_ALERTED.

If the *WaitMode* parameter is *UserMode* and the *Alertable* parameter TRUE, then the thread can also be awakened to deliver a user mode APC. Kernel mode APCs always cause the subject thread to be awakened if the Wait IRQL is zero and no kernel APC is in progress.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The expiration time of the delay is expressed as either an absolute time at which the delay is to expire, or time that is relative to the current system time. If the value of the *Interval* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

The value returned by **KeDelayExecutionThread** function determines how the delay was completed.

A value of STATUS_SUCCESS is returned if the delay was completed because the specified interval of time elapsed.

A value of STATUS_ALERTED is returned if the delay was completed because the thread was alerted.

A value of STATUS_USER_APC is returned if a user mode APC is to be delivered.

2.1.4.6 Disable Queuing of APCs

The queuing of APCs to a thread object can be disabled with the **KeDisableApcQueuingThread** function:

BOOLEAN
KeDisableApcQueuingThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Disabling the queuing of APCs to a thread object causes any attempt to direct an APC to the thread to be ignored.

During the termination of a thread, the executive must run down and clean up all thread data structures. When the APC queue itself is processed, the executive first disables APCs and then flushes the APC queue.

The previous value of the APC queuable state is returned as the function value. If the return value is TRUE, then APC queuing was previously enabled. Otherwise, a value of FALSE is returned and APC queuing was disabled.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.4.7 Enable Queuing of APCs

The queuing of APCs to a thread object can be enabled with the **KeEnableApcQueuingThread** function:

BOOLEAN
KeEnableApcQueuingThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Enabling the queuing of APCs to a thread object allows APC objects to be inserted in the subject thread's APC queue for subsequent delivery when conditions permit.

The previous value of the APC queuable state is returned as the function value. If the return value is TRUE, then APC queuing was previously enabled. Otherwise, a value of FALSE is returned and APC queuing was disabled.

2.1.4.8 Force Resumption of Thread

A thread object's execution can be forced to resume with the **KeForceResumeThread** function:

ULONG
KeForceResumeThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Forcing the resumption of a thread object's execution checks the suspend and freeze count of the subject thread. If both counts are zero, then the thread is not currently suspended and no operation is performed. Otherwise, the subject thread's suspend and freeze counts are both set to zero, and the execution of the subject thread is resumed by releasing its builtin suspend semaphore.

The sum of the previous suspend and freeze counts is returned as the function value. If the return value is zero, then the subject thread was not previously

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

suspended. Otherwise, the subject thread was suspended and its execution was resumed.

This function is intended for use by the executive when it wants to terminate the execution of a thread that may be in a suspended state.

2.1.4.9 Freeze Thread

The execution of a thread object can be frozen with the **KeFreezeThread** function:

ULONG
KeFreezeThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Freezing a thread object causes the thread's freeze count to be incremented. If incrementing the freeze count would cause it to overflow, then the count is not incremented and the exception STATUS_SUSPEND_COUNT_EXCEEDED is raised. Otherwise, the freeze count is incremented. If the resultant value is one and the suspend count is zero, then the thread's builtin suspend APC object is queued.

When the suspend APC is delivered to the subject thread, a nonalertable Wait on the thread's builtin semaphore object is executed which freezes thread execution. The subject thread can be subsequently unfrozen with the **KeUnfreezeThread** function.

The previous freeze count is returned as the function value. If the return value is zero, then the subject thread was not previously frozen. Otherwise, the thread was previously frozen and must be unfrozen the number of times specified by the return value plus one before it will actually resume execution.

The freeze and unfreeze functions are similar to the suspend and resume functions, but are intended for use by system software as opposed to being exported to users. The freeze and unfreeze functions are used to suspend and resume thread execution during debugging operations.

2.1.4.10 Query Data Alignment Mode

The data alignment handling mode for the current thread can be queried with the **KeQueryAutoAlignmentThread** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
KeQueryAutoAlignmentThread (
)

The data alignment handling mode for the current thread is returned as the function value. A value of TRUE is returned if user mode data alignment exceptions are automatically handled by the kernel and are not raised as exceptions. Otherwise, user mode data alignment exceptions are not handled by the kernel and may, or may not, be raised as exceptions depending on host hardware capabilities. Automatic handling of user mode data alignment exceptions means that the kernel emulates misaligned data references and completes the offending instructions as if no misalignment exception had occurred. Misaligned references in kernel mode are never automatically handled and are always raised as exceptions.

IMPLEMENTATION NOTES:

Certain processors (e.g., the i386) always handle misaligned data in hardware. On these processors, enabling or disabling the automatic handling of data alignment exceptions has no effect. On other processors (e.g., i486, MIPS r3000, r4000SP, and r4000MP) the handling of misaligned data is handled according to the mode established for the respective thread.

2.1.4.11 Query Base Priority

The base priority of a thread object can be queried with the **KeQueryBasePriorityThread** function:

LONG
KeQueryBasePriorityThread (
 IN PKTHREAD *Thread*
 );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

The base priority increment of the specified thread is returned as the function value. The base priority increment is defined as the difference between the specified thread's base priority and the base priority of the thread's process.

2.1.4.12 Read State Thread

The current state of a thread object can be read with the **KeReadStateThread** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
KeReadStateThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

The current state of the thread object is returned as the function value. If the current state of the thread is Signaled, then a value of TRUE is returned. Otherwise, a value of FALSE is returned.

2.1.4.13 Ready Thread

A thread object can be readied for execution with the **KeReadyThread** function:

VOID
KeReadyThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Readying a thread object causes the thread to be considered for immediate execution on a processor.

If the thread's process is not in the balance set, then the thread's dispatching state is set to Ready and the thread object is inserted in the process' ready queue. Otherwise, an attempt is made to dispatch the thread on one of the processors in the multiprocessor complex. If the priority of the subject thread is greater than the priority of one or more threads running on processors that the subject thread can also run on, then the thread with the lowest priority is selected for preemption, the subject thread's dispatching state is set to Standby and an interprocessor interrupt is sent to the target processor to cause it to redispatch. Otherwise, the subject thread's dispatching state is set to Ready and the thread is inserted at the tail of the dispatcher ready queue selected by its priority.

2.1.4.14 Resume Thread

The execution of a thread object can be resumed with the **KeResumeThread** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG
KeResumeThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Resuming a thread object checks the suspend count of the subject thread. If the suspend count is zero, then the thread is not currently suspended and no operation is performed. Otherwise, the subject thread's suspend count is decremented. If the resultant value is zero and the freeze count is also zero, then the execution of the subject thread is resumed by releasing its builtin suspend semaphore.

The previous suspend count is returned as the function value. If the return value is zero, then the subject thread was not previously suspended. If the return value is one, then the subject thread's execution was resumed. If the returned value is not zero or one, then the subject thread is still suspended and must be resumed the number of times specified by the return value minus one before it will actually resume execution.

The suspend and resume functions are similar to the freeze and unfreeze functions, but are usable from both system and user software. The freeze and unfreeze functions are used to suspend and resume thread execution during debugging operations.

2.1.4.15 Rundown Thread

The current thread object can be run down with the **KeRundownThread** function:

VOID
KeRundownThread (
    );

This function runs down thread structures that are guarded by the dispatcher database lock and which must be processed before actually terminating the thread. An example of such a data structure is the mutant ownership list that is anchored in the thread object.

2.1.4.16 Set Affinity Thread

The affinity of a thread object can be set with the **KeSetAffinity** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
KAFFINITY
KeSetAffinityThread (
    IN PKTHREAD Thread,
    IN KAFFINITY Affinity
    );
```

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Affinity - The new set of processors on which the thread can run.

Setting the affinity of a thread object establishes a new set of processors on which the thread can execute. The new affinity must be a subset of the parent process's affinity. If the new affinity is zero, or is not a subset of the parent process's affinity, then an error condition is raised. Otherwise, the new affinity of the thread is set, and the previous affinity is returned as the function value.

If the dispatching state of the thread object is Running or Standby and the new affinity is such that the thread cannot execute on the target processor, then a new thread is selected for execution and an interprocessor interrupt is sent to the target processor.

2.1.4.17 Set Data Alignment Mode

The data alignment handling mode for the current thread can be set with the **KeSetAutoAlignmentThread** function:

```
BOOLEAN
KeSetAutoAlignmentThread (
    IN BOOLEAN Enable
    )
```

Parameters:

Enable - A boolean variable that specifies the handling mode for data alignment exceptions in the current thread.

The *Enable* parameter specifies the handling mode for data alignment exceptions in the current thread. If this parameter is TRUE, then user mode data alignment exceptions are automatically handled by the kernel and are not raised as exceptions. Otherwise, user mode data alignment exceptions are not handled by the kernel and may, or may not, be raised as exceptions depending on host hardware capabilities. Automatic handling of user mode data alignment exceptions means

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

that the kernel emulates misaligned data references and completes the offending instructions as if no misalignment exception had occurred. Misaligned references in kernel mode are never automatically handled and are always raised as exceptions.

The previous data alignment handling mode is returned as the function value.

IMPLEMENTATION NOTES:

Certain processors (e.g., the i386) always handle misaligned data in hardware. On these processors, enabling or disabling the automatic handling of data alignment exceptions has no effect. On other processors (e.g., i486, MIPS r3000, r4000SP, and r4000MP) the handling of misaligned data is handled according to the mode established for the respective thread.

2.1.4.18 Set Base Priority

The base priority of a thread object can be set with the **KeSetBasePriorityThread** function:

LONG
KeSetBasePriorityThread (
    IN PKTHREAD *Thread*,
    IN LONG *Increment*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Increment - The base priority increment that is to be applied to the specified thread.

The new base priority is computed by adding the specified priority increment to the base priority of the specified thread's process. The resultant value is stored as the base priority of the specified thread.

The new base priority is restricted to the priority class of the specified thread's process. This means that the base priority is not allowed to cross over from a realtime priority class to a variable priority class or vice versa.

The previous base priority increment of the specified thread is returned as the function value. The previous base priority increment is defined as the difference between the specified thread's old base priority and the base priority of the thread's process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.4.19 Set Priority Thread

The priority of a thread object can be set with the **KeSetPriority** function:

KPRIORITY
KeSetPriorityThread (
    IN PKTHREAD *Thread*,
    IN KPRIORITY *Priority*
    );

<u>Parameters:</u>

Thread - A pointer to a dispatcher object of type thread.

Priority - The new priority of the thread.

Setting the priority of a thread object causes its eligibility for execution to be reexamined. The exact action that is taken depends on the current dispatching state of the thread. If the new priority is greater than the maximum thread priority, then an error condition is raised.

If the dispatching state of the thread object is Ready and the thread is currently inserted in the parent process's ready queue, then the priority of the thread is changed and no further action is taken.

If the dispatching state of the thread object is Ready and the thread is currently inserted in one of the dispatcher ready queues, then the thread is removed from its current ready queue, its priority is set to the specified value, and the thread is readied for execution as if it had just entered the ready state.

If the dispatching state of the thread object is Waiting or Terminated, then the priority of the thread is changed and no further action is taken.

If the dispatching state of the thread object is Standby or Running and the priority of the thread is being raised, then the priority of the thread is changed and no further action is taken.

If the dispatching state of the thread object is Standby or Running and the priority of the thread is being lowered, then a check is performed to determine if the thread should be preempted to run a higher priority thread. If a higher priority thread can execute on the target processor, then it is selected for execution and an interprocessor interrupt is sent to the target processor. Otherwise, no action is taken.

The previous priority of the subject thread is returned as the function value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.4.20 Suspend Thread

The execution of a thread object can be suspended with the **KeSuspendThread** function:

```
ULONG
KeSuspendThread (
    IN PKTHREAD Thread
    );
```

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Suspending a thread object causes the thread's suspend count to be incremented. If incrementing the suspend count would cause it to overflow, then the count is not incremented and the exception STATUS_SUSPEND_COUNT_EXCEEDED is raised. Otherwise, the suspend count is incremented, and if the resultant value is one and the freeze count is zero, then the thread's builtin suspend APC object is queued.

When the suspend APC is delivered to the subject thread, a nonalertable Wait on the thread's builtin semaphore object is executed which suspends thread execution. The subject thread can be subsequently resumed with either the **KeResumeThread** or **KeAlertResumeThread** function.

The previous suspend count is returned as the function value. If the return value is zero, then the subject thread was not previously suspended. Otherwise, the thread was previously suspended and must be resumed the number of times specified by the return value plus one before it will actually resume execution.

The suspend and resume functions are similar to the freeze and unfreeze functions, but are usable from both system and user software. The freeze and unfreeze functions are used to suspend and resume thread execution during debugging operations.

2.1.4.21 Terminate Thread

The execution of the current thread can be terminated with the **KeTerminateThread** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
KeTerminateThread (
    IN KPRIORITY *Increment*
    );

Parameters:

Increment - The priority increment that is to be applied is terminating the current thread causes a Wait to be satisfied.

Terminating the current thread causes the dispatching state of the thread to be set to Terminated and the state of the thread object to be set to Signaled.

An attempt is made to satisfy as many Waits as possible for the current thread object.

A new thread object is selected for execution on the current processor and a context switch to the new thread is performed. There is no return from this function.

2.1.4.22 Test Alert Thread

An alert condition for the current thread can be tested for with the **KeTestAlertThread** function:

BOOLEAN
KeTestAlertThread (
    IN KPROCESSOR_MODE *AlertMode*
    );

Parameters:

AlertMode - The processor mode (*UserMode* or *KernelMode*) which is to be tested for an alert condition.

This function tests to determine if the current thread's alert variable for the specified processor mode has a value of TRUE or whether a user mode APC should be delivered to the current thread.

If the alert variable associated with the specified processor mode is TRUE, then it is set to a value of FALSE.

If the alert variable associated with the specified processor mode is FALSE and the specified processor mode is user, then the subject thread's APC queue is examined to determine whether a user mode APC should be delivered. If the user mode APC

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

queue contains one or more entries, then the user APC pending variable is set to a value of TRUE in the current thread object.

The previous value of the alert variable for the specified processor mode is returned as the function value. If the return value is TRUE, then the current thread was alerted. Otherwise, a value of FALSE is returned and the current thread was not alerted.

2.1.4.23 Unfreeze Thread

The execution of a thread object can be unfrozen with the **KeUnfreezeThread** function:

ULONG
KeUnfreezeThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Unfreezing a thread object checks the freeze count of the subject thread. If the freeze count is zero, then the thread is not currently frozen and no operation is performed. Otherwise, the subject thread's freeze count is decremented. If the resultant value is zero and the suspend count is also zero, then the execution of the subject thread is unfrozen by releasing its builtin suspend semaphore.

The previous freeze count is returned as the function value. If the return value is zero, then the subject thread was not previously frozen. If the return value is one, then the subject thread's execution was unfrozen. If the returned value is not zero or one, then the subject thread is still frozen and must be unfrozen the number of times specified by the return value minus one before it will actually resume execution.

The freeze and unfreeze functions are similar to the suspend and resume functions, but are intended for use by system software as opposed to being exported to users. The freeze and unfreeze functions are used to suspend and resume thread execution during debugging operations.

2.1.4.24 Thread Performance Data

Several counters are maintained for each thread object to determine the various execution characteristics of the thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Two of the counters maintain the number of clock ticks that occurred while the thread was in user mode and while the thread was in kernel mode. These counters provide a quantitative measure of the distribution of computation time between the system and the user.

2.1.5 Timer Object

A *timer object* is used to record the passage of time. A timer object is set to a specified time and then expires when the time becomes due. When a timer object is set, its state is set to Not-Signaled and it is inserted in the system timer queue according to its expiration time. When the timer object expires, it is removed from the system timer queue and its state is set to Signaled.

A Deferred Procedure Call (DPC) can optionally be executed when a timer expires. This procedure executes at IRQL DISPATCH_LEVEL in the context of whatever thread happens to be executing when the timer expires.

Waiting on a timer object causes the execution of the subject thread to be suspended until the timer object attains a Signaled state. Satisfying the Wait for a timer object does not cause the state of the timer object to change. Therefore, when a timer object attains a Signaled state, an attempt is made to Satisfy as many Waits as possible.

Timer objects can be used to synchronize the execution of specific actions with time. This execution can occur at fixed points in time or at various intervals.

Programming interfaces that support the thread object include:

KeInitializeTimer - Initialize a timer object
KeCancelTimer - Cancel timer object expiration
KeReadStateTimer - Read state of timer object
KeSetTimer - Set timer object expiration time

2.1.5.1 Initialize Timer

A timer object can be initialized with the **KeInitializeTime** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
KeInitializeTimer (
    IN PKTIMER *Timer*
    );

Parameters:

Timer - A pointer to a dispatcher object of type timer.

The timer object data structure is initialized with a state of Not-Signaled.

2.1.5.2 Cancel Timer

A timer object can be canceled with the **KeCancelTimer** function:

BOOLEAN
KeCancelTimer (
    IN PKTIMER *Timer*
    );

Parameters:

Timer - A pointer to a dispatcher object of type timer.

If the timer object is currently in the system timer queue, then it is removed from the queue and a value of TRUE is returned as the function value (a boolean state variable records whether the timer object is in the system timer queue). Otherwise, no operation is performed and a value of FALSE is returned as the function value.

2.1.5.3 Read State Timer

The current state of a timer object can be read with the **KeReadStateTimer** function:

BOOLEAN
KeReadStateTimer (
    IN PKTIMER *Timer*
    );

Parameters:

Timer - A pointer to a dispatcher object of type timer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The current state of the timer object is returned as the function value. If the current state of the timer object is Signaled, then a value of TRUE is returned. Otherwise, a value of FALSE is returned.

2.1.5.4 Set Timer

A timer object can be set to expire at a specified time with the **KeSetTimer** function:

BOOLEAN
KeSetTimer (
    IN PKTIMER *Timer*,
    IN TIME *DueTime*,
    IN PKDPC *Dpc* **OPTIONAL**
    );

Parameters:

Timer - A pointer to a dispatcher object of type timer.

DueTime - The absolute or relative time at which the timer is to expire.

Dpc - An optional pointer to a control object of type deferred procedure call.

Setting a timer object causes the absolute expiration time to be computed, the state of the timer to be set to Not-Signaled, and the timer object to be inserted in the system timer queue. If the timer object is already in the timer queue, then it is implicitly canceled before it is set to the new expiration time (a boolean state variable records whether the timer object is in the system timer queue).

The expiration time of the timer object is expressed as either the absolute time that the timer is to expire, or a time that is relative to the current system time. If the value of the *DueTime* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

The expiration time is expressed in system time units which are 100ns intervals.

If the *Dpc* parameter is specified, then a DPC object is associated with the timer object.

If the timer object was previously in the system timer queue, then a value of TRUE is returned as the function value. Otherwise, a value of FALSE is returned.

When the timer object expires, it is removed from the system timer queue and its state is set to Signaled. If a DPC object was associated with the timer object when it was set, then it is inserted in the system DPC queue and will execute as soon as

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

conditions permit (a boolean state variable records whether the DPC object is in the system DPC queue).

2.2 Control Objects

This section describes the various types of control objects and the interfaces that are provided to manipulate these objects.

2.2.1 Asynchronous Procedure Call (APC) Object

An *Asynchronous Procedure Call* (APC) object provides the capability to break into the execution of a specified thread and cause a procedure to be called in a specified processor mode. Software running in kernel mode can only be interrupted to execute asynchronous procedures in kernel mode, whereas software running in user mode can be interrupted to execute asynchronous procedures in both user and kernel mode.

An asynchronous procedure call occurs in the context of a specified thread and is triggered by a software interrupt at APC_LEVEL.

There are two types of APC objects:

1. *Special*
2. *Normal*

Special APC objects cause the execution of a thread to be interrupted to execute a procedure in kernel mode at IRQL APC_LEVEL. Special APC objects can break into the execution of a thread at any time the thread is executing at IRQL 0.

Ordinarily, special APC procedures perform a minimal amount of work and return immediately to the APC dispatcher without calling any external procedures. However, if code running as part of a special APC procedure acquires a mutex that is also acquired by code outside the special APC procedure, then the code outside the special APC procedure must explicitly raise IRQL to APC_LEVEL when it wants to acquire the mutex.

This convention is required to prevent the special APC procedure from acquiring the mutex at an inappropriate time, which can happen if the thread that receives the special APC already owns the mutex when the special APC procedure is executed.

During the execution of a special APC procedure, page faults can be taken and all the kernel services are available. However, system services are not available, and care must be taken to ensure that any executive services that are used can be safely called.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Normal APC objects cause the execution of a thread to be interrupted to execute a procedure in kernel mode at IRQL APC_LEVEL and, in addition, a procedure in either kernel or user mode at IRQL 0. The first procedure (executed in kernel mode) is called just prior to calling the second procedure in the specified mode, and must adhere to the conventions for special APC procedures.

Normal APC objects can only break into the execution of a thread when the thread is executing at IRQL 0 and a normal APC for the specified mode is not already active.

While a normal APC is active for kernel mode, further normal APCs are software disabled until the active APC completes. The delivery of normal APCs for kernel mode is also implicitly disabled while a thread owns one or more mutexes (i.e, the thread does not have to explicitly raise IRQL to APC_LEVEL in order to synchronize with normal APC procedures).

Normal user mode APCs are only delivered when the subject thread is alertable. This occurs when a thread waits user-mode alertable and when the thread calls **KeTestAlertThread**.

While a normal APC is active for user mode, further normal APCs for user mode are software disabled by the alert mechanism. Upon completion of a normal APC in user mode, **KeTestAlertThread** is automatically called, which enables the delivery of another user mode APC. Thus, once a thread is user-mode alertable and an APC is delivered, further APCs are delivered one after the other until there are no APCs remaining in the user-mode APC queue.

During the execution of a normal APC procedure, all system operations are available and page faults can be taken.

Programming interfaces that support the APC object include:

KeInitializeApc - Initialize an APC object
KeFlushQueueApc - Flush all APC objects from APC queue
KeInsertQueueApc - Insert APC object into APC queue
KeRemoveQueueApc - Remove APC object from APC queue

2.2.1.1 Initialize APC

An APC object can be initialized with the **KeInitializeApc** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
VOID
KeInitializeApc (
    IN PKAPC Apc,
    IN PKTHREAD Thread,
    IN KAPC_ENVIRONMENT Environment,
    IN PKKERNEL_ROUTINE KernelRoutine,
    IN PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL,
    IN PKNORMAL_ROUTINE NormalRoutine OPTIONAL,
    IN KPROCESSOR_MODE ApcMode OPTIONAL,
    IN PVOID NormalContext OPTIONAL
    );
```

Parameters:

Apc - A pointer to a control object of type APC.

Thread - A pointer to a dispatcher object of type thread.

Environment - The environment in which the APC will execute (*OriginalApcEnvironment*, *AttachedApcEnvironment*, or *CurrentApcEnvironment*).

KernelRoutine - A pointer to a function that is to be executed at IRQL APC_LEVEL in kernel mode.

RundownRoutine - An optional pointer to a function that is to be executed if the APC object is contained in a thread's APC queue when the thread terminates.

NormalRoutine - An optional pointer to a function that is to be executed at IRQL 0 in the specified processor mode. If this parameter is not specified, then the *ApcMode* and *NormalContext* parameters are ignored.

ApcMode - The processor mode (*UserMode* or *KernelMode)* in which the function specified by the *NormalRoutine* parameter is to be executed. This parameter is ignored if the *NormalRoutine* parameter is not specified.

NormalContext - A pointer to an arbitrary data structure which is to be passed to the function specified by the *NormalRoutine* parameter. This parameter is ignored if the *NormalRoutine* parameter is not specified.

The function specified by the *KernelRoutine* parameter has the following type definition:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

typedef
VOID
(***PKKERNEL_ROUTINE**) (
    IN PKAPC *Apc*,
    IN OUT PKNORMAL_ROUTINE **NormalRoutine*,
    IN OUT PVOID **NormalContext*,
    IN OUT PVOID **SystemArgument1*,
    IN OUT PVOID **SystemArgument2*
    );

<u>Parameters:</u>

Apc - A pointer to a control object of type APC.

NormalRoutine - A pointer to a pointer to the normal routine function that was specified when the APC was initialized.

NormalContext - A pointer to a pointer to an arbitrary data structure that was specified when the APC was initialized.

SystemArgument1, *SystemArgument2* - A set of two pointers to two arguments that contain untyped data.

The function specified by the *RundownRoutine* parameter has the following type definition:

typedef
VOID
(***PKRUNDOWN_ROUTINE**) (
    IN PKAPC *Apc*
    );

<u>Parameters:</u>

Apc - A pointer to a control object of type APC.

The function specified by the *NormalRoutine* parameter has the following type definition:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef
VOID
(*PKNORMAL_ROUTINE) (
    IN PVOID NormalContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
    );
```

Parameters:

NormalContext - A pointer to an arbitrary data structure that was specified when the corresponding APC object was initialized.

SystemArgument1, *SystemArgument2* - A set of two arguments that contain untyped data.

The type of APC object to be initialized is determined by the presence or absence of the optional *NormalRoutine* parameter. If the *NormalRoutine* parameter is present, then a normal APC object is initialized and the values of the *ApcMode* and *NormalContext* parameters are stored in the APC object. Otherwise, a special APC object is initialized for execution in kernel mode.

The *Environment* parameter specifies the execution environment of the specified APC object. An APC object can be executed in the context of a thread's parent process or a process to which the thread has attached.

The *KernelRoutine* parameter specifies the procedure that is to be called in kernel mode at IRQL APC_LEVEL. This procedure is called with a copy of the parameters that are specified for the normal routine when the APC is initialized and can modify these parameters as necessary to alter the execution of the normal routine. If the normal routine is not specified when the APC is initialized, then any assignment to these parameters is ignored.

If specified, the *RundownRoutine* parameter specifies a procedure that is to be called when a thread terminates with the APC object in its APC queue. The purpose of this routine is to allow for special disposition of the APC object during thread rundown.

If specified, the *NormalRoutine* parameter specifies the procedure that is to be executed in the processor mode specified by the *ApcMode* parameter. This procedure will be called with the *NormalContext* parameter and two additional arguments provided by the system when the APC queued.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

In order to actually interrupt the execution of a thread, an APC object must be inserted into one of the thread's APC queues (there is a separate APC queue for user and kernel mode).

2.2.1.2 Flush Queue APC

All the APC objects in a specified thread's APC queue can be flushed with the **KeFlushQueueApc** function:

PLIST_ENTRY
KeFlushQueueApc (
    IN PKTHREAD *Thread*,
    IN KPROCESSOR_MODE *ApcMode*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

ApcMode - The processor mode (*UserMode* or *KernelMode*) of the APC queue that is to be flushed.

An APC queue is flushed by removing the APC listhead from the list of APC entries, reinitializing the APC listhead, and returning the list of APC objects as the function value. If the APC queue is empty, then a NULL pointer is returned. Otherwise, the address of the list entry for the first APC object is returned as the function value. It is the responsibility of the caller to scan the list of APC objects and dispense with each object as appropriate.

The APC objects are linked together by a list entry in each object. Scanning this list can be accomplished using the CONTAINING_RECORD function to locate the address of the respective APC object given the address of its list entry.

This function is used by the executive during thread termination to remove remaining entries from the thread's APC lists.

2.2.1.3 Insert Queue APC

An APC object can be inserted into a thread's APC queue with the **KeInsertQueueApc** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
BOOLEAN
KeInsertQueueApc (
    IN PKAPC Apc,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2,
    IN KPRIORITY Increment
    );
```

Parameters:

Apc - A pointer to a control object of type APC.

SystemArgument1, *SystemArgument2* - A set of two arguments that contain untyped data.

Increment - The priority increment that is to be applied if queuing the APC causes the target thread's wait to be satisfied.

If the specified APC object is already in an APC queue (a boolean state variable records whether the APC object is in an APC queue) or APC queuing is disabled for the subject thread, then no operation is performed and a function value of FALSE is returned. Otherwise, the APC object is inserted into the APC queue specified by the *ApcMode* and *Thread* parameters that were supplied when the APC object was initialized and a function value of TRUE is returned.

When proper enabling conditions are present, the APC will be delivered to the subject thread and the specified procedure(s) will be executed in the specified processor mode.

A special APC is deliverable whenever the IRQL of the subject thread is zero.

A normal kernel APC is deliverable whenever the IRQL of the subject thread is zero, a normal kernel APC is not already in progress, and the subject thread does not own any mutexes.

A normal user APC is deliverable when the subject thread waits user-mode alertable and when the subject thread calls **KeTestAlertThread**.

2.2.1.4 Remove Queue APC

An APC object can be removed from an APC queue with the **KeRemoveQueueApc** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
KeRemoveQueueApc (
    IN PKAPC *Apc*
    );

Parameters:

Apc - A pointer to a control object of type APC.

If the specified APC object is not currently in an APC queue (a boolean state variable records whether the APC object is in an APC queue), then a value of FALSE is returned and no operation is performed. Otherwise, the specified APC object is removed from its APC queue and a function value of TRUE is returned.

2.2.2 Deferred Procedure Call (DPC) Object

A *Deferred Procedure Call* (DPC) object provides the capability to break into the execution of the current thread and cause a procedure to be executed in kernel mode at IRQL DISPATCH_LEVEL.

There is one DPC queue for the entire system. When a DPC object is inserted in the DPC queue, a software interrupt is requested at DISPATCH_LEVEL on the current processor. As soon as the IRQL falls below DISPATCH_LEVEL, a software interrupt will be taken which will cause the DPC dispatcher to execute.

The DPC dispatcher removes entries from the DPC queue, calls the specified procedure, and upon return, removes another entry from the queue. This is continued until there are no longer any entries in the DPC queue, at which time the DPC dispatcher checks to determine if another thread has been selected for execution on the current processor. If a thread has been selected, then a context switch to that thread is performed. Otherwise, the interrupt is dismissed and execution of the current thread is continued.

A deferred procedure call occurs at IRQL DISPATCH_LEVEL in the context of whatever thread was interrupted when the DISPATCH_LEVEL interrupt occurred. A very limited set of operations can be performed by the DPC procedure.

No system services can be executed by the DPC procedure nor can any page faults be taken. The kernel services are generally available. However, the Wait functions can only be called if it is known that they will not actually cause a wait to occur. (Using an explicit time-out value of zero implements a conditional Wait operation). If page faults or waits were allowed, then it would be possible to randomly cause an arbitrary thread to wait in the kernel at IRQL DISPATCH_LEVEL causing possible deadlocks and data corruption.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Deferred procedure execution is intended mainly for use by device drivers that need to lower their IRQL to complete an I/O operation. The kernel itself, however, uses DPC objects to implement timers, quantum end, and power failure recovery.

Programming interfaces that support the DPC object include:

KeInitializeDpc - Initialize a DPC object
KeInsertQueueDpc - Insert DPC object into the DPC queue
KeRemoveQueueDpc - Remove DPC object from the DPC queue

2.2.2.1 Initialize DPC

A DPC object can be initialized with the **KeInitializeDpc** function:

VOID
KeInitializeDpc (
    IN PKDPC *Dpc*,
    IN PKDEFERRED_ROUTINE *DeferredRoutine*,
    IN PVOID *DeferredContext*
    );

Parameters:

Dpc - A pointer to a control object of type DPC.

DeferredRoutine - A pointer to a function that is to be called when the DPC object is removed from the DPC queue.

DeferredContext - A pointer to an arbitrary data structure that is to be passed to the function specified by the *DeferredRoutine* parameter.

The function specified by the *DeferredRoutine* parameter has the following type definition:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

typedef
VOID
(*PKDEFERRED_ROUTINE) (
    IN PKDPC *Dpc,*
    IN PVOID *DeferredContext,*
    IN PVOID *SystemArgument1*,
    IN PVOID *SystemArgument2*
    );

Parameters:

Dpc - A pointer to a control object of type DPC.

DeferredContext - A pointer to an arbitrary data structure that was specified when the DPC was initialized.

SystemArgument1, *SystemArgument2* - A set of two arguments that contain untyped data.

2.2.2.2 Insert Queue DPC

A DPC object can be inserted in the system DPC queue with the **KeInsertQueueDpc** function:

BOOLEAN
KeInsertQueueDpc (
    IN PKDPC *Dpc*,
    IN PVOID *SystemArgument1*,
    IN PVOID *SystemArgument2*
    );

Parameters:

Dpc - A pointer to a control object of type DPC.

SystemArgument1, *SystemArgument2* - A set of two arguments that contain untyped data.

If the specified DPC object is already in the DPC queue (a boolean state variable records whether the DPC object is in the DPC queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the DPC object is inserted in the DPC queue, a software interrupt is request at IRQL DISPATCH_LEVEL on the current processor, and a function value of TRUE is returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The deferred procedure will be executed as soon as the IRQL of the current processor drops below DISPATCH_LEVEL.

2.2.2.3 Remove Queue DPC

A DPC object can be removed from the DPC queue with the **KeRemoveQueueDpc** function:

BOOLEAN
KeRemoveQueueDpc (
    IN PKDPC *Dpc*
    );

Parameters:

Dpc - A pointer to a control object of type DPC.

If the specified DPC object is not currently in the DPC queue (a boolean state variable records whether the DPC object is in the DPC queue), then a value of FALSE is returned and no operation is performed. Otherwise, the specified DPC object is removed from the DPC queue and a function value of TRUE is returned.

2.2.3 Device Queue Object

A *device queue object* is used to record the state of a device driver and to provide a queue into which I/O requests can be placed for subsequent processing.

A device queue object has a state which is either *Busy* or *Not-Busy*.

When the state of a device queue object is Not-Busy, the associated device driver is idle and therefore not performing any work.

A device queue object transitions to the Busy state when an attempt is made to insert a device queue entry into a device queue that is empty. For this case, the device queue entry is not actually placed in the device queue, but rather, the device queue object is marked Busy and a boolean value of FALSE is returned to signify that the associated device driver should process the device queue entry immediately.

Once a device queue object is Busy, further I/O requests are placed in the device queue in either a FIFO or key-sorted order.

A device queue object transitions to a Not-Busy state when an attempt is made to remove a device queue entry from a device queue object and the corresponding device queue is empty.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

A device queue entry has the following type definition:

typedef struct _KDEVICE_QUEUE_ENTRY {
    LIST_ENTRY *DeviceListEntry*;
    ULONG *SortKey*;
    BOOLEAN *Inserted*;
} **KDEVICE_QUEUE_ENTRY**;

Programming interfaces that support the device queue object include:

KeInitializeDeviceQueue - Initialize a device queue
KeInsertDeviceQueue - Insert entry at tail of device queue
KeInsertByKeyDeviceQueue - Insert entry by key into device queue
KeRemoveDeviceQueue - Remove entry from head of device queue
KeRemoveEntryDeviceQueue - Remove entry from device queue

2.2.3.1 Initialize Device Queue

A device queue object can be initialized with the **KeInitializeDeviceQueue** function:

VOID
KeInitializeDeviceQueue (
    IN PKDEVICE_QUEUE *DeviceQueue*,
    IN PKSPIN_LOCK *SpinLock*
    );

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

SpinLock - A pointer to an executive spin lock.

The device queue object data structure is initialized and the state of the device queue is set to Not-Busy.

2.2.3.2 Insert Device Queue

An entry can be inserted at the tail of a device queue with the **KeInsertDeviceQueue** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
BOOLEAN
KeInsertDeviceQueue (
    IN PKDEVICE_QUEUE DeviceQueue,
    IN PKDEVICE_QUEUE_ENTRY DeviceQueueEntry
    );
```

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

DeviceQueueEntry - A pointer to the device queue entry that is to be inserted at the tail of the device queue.

The specified device queue spin lock is acquired, and the state of the device queue is checked.

If the state of the device queue is Not-Busy, then the state of the device queue is set to Busy, the device queue spin lock is released, and a value of FALSE is returned as the function value (i.e., the device queue entry is not inserted in the device queue).

If the state of the device queue is Busy, then the specified device queue entry is inserted at the tail of device queue, the device queue spin lock is released, and a value of TRUE is returned as the function value.

This function is intended for use by code that queues an I/O request to a device driver. It must be called from an IRQL of DISPATCH_LEVEL.

2.2.3.3 Insert By Key Device Queue

An entry can be inserted into a device queue according to a key value with the **KeInsertByKeyDeviceQueue** function:

```
BOOLEAN
KeInsertByKeyDeviceQueue (
    IN PKDEVICE_QUEUE DeviceQueue,
    IN PKDEVICE_QUEUE_ENTRY DeviceQueueEntry,
    IN ULONG SortKey
    );
```

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

DeviceQueueEntry - A pointer to the device queue entry that is to be inserted into the device queue by key.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SortKey - The sort key value that is to be used to determine the position at which the device queue entry is to be inserted in the specified device queue.

The specified device queue spin lock is acquired, and the state of the device queue is checked.

If the state of the device queue is Not-Busy, then the state of the device queue is set to Busy, the device queue spin lock is released, and a value of FALSE is returned as the function value (i.e., the device queue entry is not inserted in the device queue).

If the state of the device queue is Busy, then the specified device queue entry is inserted into the device queue according to its sort key value, the device queue spin lock is released, and a value of TRUE is returned as the function value.

Insertion in the device queue is such that the preceding entry in the queue has a sort key that is less than or equal to the new entry's sort key and the succeeding entry has a sort key that is greater than the new entry's sort key.

This function is intended for use by code that queues an I/O request to a device driver. It must be called from an IRQL of DISPATCH_LEVEL.

2.2.3.4 Remove Device Queue

An entry can be removed from the head of a device queue with the **KeRemoveDeviceQueue** function:

PKDEVICE_QUEUE_ENTRY
KeRemoveDeviceQueue (
    IN PKDEVICE_QUEUE *DeviceQueue*
    );

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

This function can only be called from an IRQL of DISPATCH_LEVEL and is intended for use by device driver code that completes one I/O request and starts the next one.

The specified device queue spin lock is acquired and the state of the device queue is checked.

If the state of the device queue is Not-Busy, then a bug check will occur (i.e., **KeRemoveDeviceQueue** cannot be called when the device queue is Not-Busy).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the state of the device queue is Busy, then an attempt is made to remove an entry from the head of the device queue. If the device queue is empty, then the state of the device queue is set to Not-Busy and a NULL pointer is returned as the function value. Otherwise, the next entry is removed from the head of the device queue, the inserted status of the entry is set to FALSE, and the address of the entry is returned as the function value.

The specified device queue spin lock is released.

2.2.3.5 Remove Entry Device Queue

A specific entry can be removed from a device queue with the **KeRemoveEntryDeviceQueue** function:

BOOLEAN
KeRemoveEntryDeviceQueue (
  IN PKDEVICE_QUEUE *DeviceQueue*,
  IN PKDEVICE_QUEUE_ENTRY *DeviceQueueEntry*
  );

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

DeviceQueueEntry - A pointer to the device queue entry that is to be removed from the specified device queue.

The IRQL is raised to DISPATCH_LEVEL and the specified device queue spin lock is acquired.

If the specified device queue entry is currently in a device queue (a boolean state variable records whether a device queue entry is in a device queue), then the device queue entry is removed from the device queue, the inserted status of the device queue entry is set to FALSE, and a value of TRUE is returned as the function value. Otherwise, the specified device queue entry is not in a device queue and a value of FALSE is returned.

The specified device queue spin lock is released and IRQL is restored to its previous value.

This function is intended for use in canceling I/O operations and is callable from any IRQL that is less than or equal to DISPATCH_LEVEL.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.4 Interrupt Object

An *interrupt object* provides the capability to connect an interrupt source to an interrupt service routine via an entry in an Interrupt Dispatch Table (IDT). Each processor has an IDT that is used to dispatch interrupts which occur on that processor.

The IDT is a software-defined table that contains an entry for each of the Interrupt Request Levels (IRQLs). When an interrupt occurs at one of these levels, the interrupt dispatcher reads the IRQL of the interrupting source from the interrupt controller. This value is then used to locate the corresponding entry in the IDT that is used to dispatch the execution of the associated service routine.

Several of the IDT entries are reserved for use by the kernel and cannot be connected to interrupt objects. These entries include the following:

- PASSIVE_LEVEL - Passive release
- APC_LEVEL - Asynchronous Procedure Call
- DISPATCH_LEVEL - Dispatch and Deferred Procedure Call
- WAKE_LEVEL - Wake system debugger
- CLOCK2_LEVEL - Interval timer
- IPI_LEVEL - Interprocessor request
- POWER_LEVEL - Power failure
- HIGH_LEVEL - Machine check

The remaining levels can be used for device interrupts or bus adapters.

In addition to the 16 entries that are directly associated with the hardware interrupt request levels, there are 48 more entries in the IDT that are provided to allow secondary level dispatching of interrupts. These entries can be used by a first-level service routine (i.e., one connected to IRQLs 0 - 15) to dispatch secondary level interrupts such as those that might be received from a bus adapter. For example, a bus adapter might have several devices that can cause interrupts and a mechanism for identifying which device is requesting service. When a bus adapter interrupt is received, the service routine connected to the appropriate first level IDT entry is executed. This service routine reads the adapter register that identifies the interrupting device and uses the information to locate the appropriate second-level

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IDT entry. The second-level IDT entry must be connected to an interrupt object and contains the address of the interrupt transfer routine which is called.

An interrupt transfer routine has the following type definition:

```
typedef
BOOLEAN
(*PKTRANSFER_ROUTINE) (
    );
```

Interrupt sources are classified as either *LevelSensitive* or *Latched*. Level sensitive interrupts request an interrupt whenever the corresponding interrupt request signal is asserted. The service routine associated with the interrupt source must remove the cause of the interrupt before the interrupt request is dropped. Latched interrupts are requested whenever the corresponding interrupt request signal transitions from the deasserted to the asserted state.

An interrupt object can only be connected to a single IDT entry. If a particular service routine must be connected to the same interrupt on multiple processors, then multiple interrupt objects must be used. Multiple interrupt objects can be connected to a single IDT entry. They must, however, all have the same interrupt type (i.e., level sensitive or latched).

Interrupt objects are intended for use by device drivers.

> \ *Kernel code that utilizes interrupts directly does not connect interrupts using this object. These interrupts include the interval timer, power failure, machine check, and the two software interrupt levels. The code for these interrupts is written in assembler since it is small and system dependent.* \

Programming interfaces that support the interrupt object include:

> **KeInitializeInterrupt** - Initialize an interrupt object
> **KeConnectInterrupt** - Connect interrupt object to an IDT entry
> **KeDisconnectInterrupt** - Disconnect interrupt object from an IDT entry
> **KeSynchronizeExecution** - Synchronize execution with an interrupt

2.2.4.1 Initialize Interrupt

An interrupt object can be initialized with the **KeInitializeInterrupt** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
KeInitializeInterrupt (
  IN PKINTERRUPT *Interrupt*,
  IN PKSERVICE_ROUTINE *ServiceRoutine*,
  IN PVOID *ServiceContext*,
  IN PKSPIN_LOCK *SpinLock*,
  IN CCHAR *Vector*,
  IN KIRQL *InterruptIrql*,
  IN KIRQL *SynchronizeIrql*,
  IN KINTERRUPT_MODE *InterruptMode*,
  IN BOOLEAN *ShareVector*,
  IN CCHAR *ProcessorNumber*,
  IN BOOLEAN *FloatingSave*
  );

Parameters:

Interrupt - A pointer to a control object of type interrupt.

ServiceRoutine - A pointer to a function that is to be called when an interrupt occurs on the specified processor through the specified IDT entry.

ServiceContext - A pointer to an arbitrary data structure which will be passed to the *ServiceRoutine* function as a parameter.

SpinLock - A pointer to an spin lock that is to be used to synchronize the execution of the *ServiceRoutine* function with the corresponding device driver.

Vector - The index of the entry in the specified IDT that is to be associated with *ServiceRoutine* function.

InterruptIrql - The request priority of the interrupting source.

SynchronizeIrql - The request priority that the interrupt should be synchronzied with.

InterruptMode - The mode of the interrupt (*LevelSensitive* or *Latched*).

ShareVector - A boolean that specifies whether the interrupt vector to which the object is connected may be shared. If FALSE then the vector may not be shared, if TRUE it may be.

ProcessorNumber - The number of the processor whose IDT is to used when connecting the interrupt.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FloatingSave - A boolean variable that specifies whether the floating point context needs to be saved when a interrupt is received from the interrupt source.

The function specified by the *ServiceRoutine* parameter has the following type definition:

typedef
BOOLEAN
(***PKSERVICE_ROUTINE**) (
    IN PKINTERRUPT *Interrupt*,
    IN PVOID *ServiceContext*
    );

Parameters:

Interrupt - A pointer to a control object of type interrupt which is connected to the associated interrupt source.

ServiceContext - A pointer to an arbitrary data structure that was specified when the corresponding interrupt object was initialized.

The interrupt object is initialized with the specified parameters. In order for the function specified by the *ServiceRoutine* parameter to actually get called when an interrupt is received from the interrupt source, the interrupt object must be connected to the specified IDT entry using the **KeConnectInterrupt** function.

The spin lock specified by the *SpinLock* parameter is used to synchronize execution.

If *SynchronizeIrql* is not equal to *InterruptIrql*, then the system will raise its priority level to *SynchronizeIrql* level before acquiring the lock specified by *SpinLock*. This allows support for devices with multiple interrupt sources, since all can be synchronized with a single spin lock at a single priority level.

It is an error for *SynchronizeIrql* to be less than *InterruptIrql*, the system will refuse to connect such an interrupt object.

An interrupt object can only be connected to a single IDT entry on a single processor. The *Vector* parameter specifies the IDT entry and the *ProcessorNumber* parameter specifies which IDT is to be used. If a particular service routine must be connected to the same IDT entry on multiple processors, then multiple interrupt objects must be used. More than one interrupt object, however, can be connected to the same IDT entry on the same processor, but all such interrupt objects must have been initialized with *ShareVector* set to TRUE. When this happens, appropriate data

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

structures are automatically set up to call each connected interrupt service routine one after the other.

The mode of the interrupt specifies whether the interrupt is a *LevelSensitive* or *Latched* interrupt. Level sensitive interrupts are continually requested as long as the interrupt signal stays asserted. Therefore, the interrupt service routine must remove the reason for the interrupt before returning control. Latched interrupts are requested only on the transition of the interrupt signal from deasserted to asserted. The function specified by the *ServiceRoutine* parameter must return a boolean value that signifies whether the interrupt was handled or not.

The *ShareVector* parameter declares whether the interrupt object may be connected to its interrupt vector at the same time as other interrupt objects. All interrupt objects sharing an interrupt vector must have *ShareVector* set to TRUE. The system may disallow sharing of an interrupt vector, even if all interrupt objects for which connections are attempted have *ShareVector* set to TRUE. This could happen because the underlying hardware does not support sharing.

The *FloatingSave* parameter specifies whether the *ServiceRoutine* function uses the floating point registers. If this parameter is TRUE, then the floating context is saved before calling the specified service routine. Otherwise, it is not saved and a fair amount of overhead is saved.

> \ *This parameter is being provided with the hope that a compiler option will be implemented that allows a module to be compiled such that it will not use the floating point registers. This option does not currently exist and this parameter should always be specified as TRUE.* \

Initializing an interrupt causes code to be generated that will synchronize execution with the appropriate interrupt object, call the specified interrupt service routine, and dismiss the interrupt.

2.2.4.2 Connect Interrupt

An interrupt object can be connected to an IDT entry with the **KeConnectInterrupt** function:

BOOLEAN
KeConnectInterrupt (
    IN PKINTERRUPT *Interrupt*
    );

Parameters:

> *Interrupt* - A pointer to a control object of type interrupt.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the specified interrupt object is already connected (a boolean state variable records whether the interrupt object is connected), the specified vector number is greater than the maximum vector, the specified IRQL is greater than HIGH_LEVEL, the specified level cannot be connected to (e.g., a reserved level, sharing conflicts), or the specified processor number is greater than the number of processors in the configuration, then no operation is performed and a function value of FALSE is returned. Otherwise, the interrupt object is connected to the IDT entry that was specified when the interrupt object was initialized and a function value of TRUE is returned.

Once an interrupt object is connected to an IDT entry, the corresponding service routine will be called each time an interrupt is received from that interrupt source. If multiple interrupt objects are connected to a single IDT entry, then the service routines are called in the order in which they were connected.

2.2.4.3 Disconnect Interrupt

An interrupt object can be disconnected from an IDT entry with the **KeDisconnectInterrupt** function:

BOOLEAN
KeDisconnectInterrupt (
    IN PKINTERRUPT *interrupt*
    );

Parameters:

Interrupt - A pointer to a control object of type interrupt.

If the specified interrupt object is not connected (a boolean state variable records whether the interrupt object is connected), then no operation is performed and a function value of FALSE is returned. Otherwise, the interrupt object is disconnected from the IDT entry that was specified when the interrupt object was initialized and a function value of TRUE is returned.

Further interrupts received from the interrupting source will be logged, but otherwise ignored.

2.2.4.4 Synchronize Execution

The execution of a device driver function can be synchronized with the execution of the service routine associated with an interrupt object with the **KeSynchronizeExecution** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
KeSynchronizeExecution (
    IN PKINTERRUPT *Interrupt*,
    IN PKSYNCHRONIZE_ROUTINE *SynchronizeRoutine*,
    IN PVOID *SynchronizeContext*
    );

Parameters:

Interrupt - A pointer to a control object of type interrupt.

SynchronizeRoutine - A pointer to a device driver function whose execution is to be synchronized with the execution of the service routine associated with the specified interrupt object.

SynchronizeContext - A pointer to an arbitrary data structure which is to be passed to the function specified by the *SynchronizeRoutine* parameter.

The function specified by the *SynchronizeRoutine* parameter has the following type definition:

typedef
BOOLEAN
(***PKSYNCHRONIZE_ROUTINE**) (
    IN PVOID *SynchronizeContext*
    );

Parameters:

ServiceContext - A pointer to an arbitrary data structure that was specified when the call to **KeSynchronizeExecution** was executed.

This function is used by a device driver to synchronize execution with a service routine which may be executing on another processor in a multiprocessor configuration. Such synchronization is only necessary in those cases where both the service routine and device driver access the same resources in a way that requires mutually exclusive access.

When this function is executed, the IRQL is raised to the level specified by the interrupt source's interrupt object (the higher of *InterruptIrql* and *SynchronizeIrql*), access is synchronized with the corresponding service routine by acquiring the associated spin lock, and then the specified routine is called. The routine should access resources as necessary and return a boolean value. Upon return, the IRQL is restored and the boolean value is returned as the function value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Routines executed with this function execute at an elevated IRQL and must be very short in duration. It is intended that these routines be used for such purposes as loading device registers and should be only a few microseconds in length.

The boolean return value is intended to be attached to the occurrence of a power failure. A device driver can use a power status object to record the occurrence of a power failure. The synchronize routine should raise IRQL to POWER_LEVEL and test the corresponding status variable before loading any device registers. If the value is TRUE, then power has failed and the device may not be in an appropriate state. If the value is FALSE, then power has not failed and a sequence of device register loads can be performed without a power failure since power failure interrupts are disabled.

2.2.5 Power Notify Object

A *power notify object* provides the capability to automatically have a specified function called when power is restored after a power failure.

This object is intended for use by device drivers and other code that needs to be asynchronously notified via a function call when power is restored after a failure. The function call can be used to reinitialize device state, restart I/O operations, etc.

A power notify object, when inserted in the power notify queue, is a repeatable operation. That is, the specified function will be called each time the power is restored. The kernel guarantees that once called, the specified function will not be recursively recalled until it has completed its execution and returned to the kernel.

When power is restored after a power failure, the kernel scans the power notify queue and calls the specified functions in the order in which they were inserted. Thus layered device drivers can ensure that they are called in the correct order.

If the power fails and is restored during the scan of the power notify queue, then the scan is immediately restarted at the beginning of the queue.

A power notify object cannot be inserted in, or removed from, the power notify queue from a function that is called as the result of power restoration (i.e., a power notify routine).

Programming interfaces that support the power notify object include:

KeInitializePowerNotify - Initialize notify object
KeInsertQueuePowerNotify - Insert power notify object
KeRemoveQueuePowerNotify - Remove power notify object

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.5.1 Initialize Power Notify

A power notify object can be initialized with the **KeInitializePowerNotify** function:

VOID
KeInitializePowerNotify (
    IN PKPOWER_NOTIFY *PowerNotify*,
    IN PKNOTIFY_ROUTINE *NotifyRoutine*,
    IN PVOID *NotifyContext*
    );

Parameters:

PowerNotify - A pointer to a control object of type power notify.

NotifyRoutine - A pointer to a function that is to be called when power is restored after a power failure.

NotifyContext - A pointer to an arbitrary data structure which will be passed to the *NotifyRoutine* as a parameter.

The function specified by the *NotifyRoutine* parameter has the following type definition:

typedef
VOID
(*PKNOTIFY_ROUTINE) (
    IN PVOID *NotifyContext*
    );

Parameters:

NotifyContext - A pointer to an arbitrary data structure that was specified when the power notify object was initialized.

The power notify object data structure is initialized.

In order to actually have the specified function called when power is restored after a failure, the power notify object must be inserted in the power notify queue.

2.2.5.2 Insert Power Notify

A power notify object can be inserted in the power notify queue with the **KeInsertQueuePowerNotify** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
KeInsertQueuePowerNotify (
    IN PKPOWER_NOTIFY *PowerNotify*
    );

<u>Parameters:</u>

PowerNotify - A pointer to a control object of type power notify.

If the specified power notify object is already in the power notify queue (a boolean state variable records whether the power notify object is in the power notify queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the power notify object is inserted in the power notify queue and a function value of TRUE is returned.

When the power is restored after a failure, the kernel scans the power notify queue and calls the specified function.

2.2.5.3 Remove Power Notify

A power notify object can be removed from the power notify queue with the **KeRemoveQueuePowerNotify** function:

BOOLEAN
KeRemoveQueuePowerNotify (
    IN PKPOWER_NOTIFY *PowerNotify*
    );

<u>Parameters:</u>

PowerNotify - A pointer to a control object of type power notify.

If the power notify object is not in the power notify queue (a boolean state variable records whether the power notify object is in the power notify queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the power notify object is removed from the power notify queue and a function value of TRUE is returned.

2.2.6 Power Status Object

A *power status object* provides the capability to automatically have a boolean state variable set to a value of TRUE when power is restored after a power failure.

This object is intended for use by device drivers and other code which needs to synchronize access to volatile register and device state such that a power failure

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

does not leave the registers or device in an indeterminate state. The boolean value can be interrogated at critical points during driver execution to determine whether a given operation should be continued or restarted.

A power status object, when inserted in the power status queue, is a one-shot operation. That is, the boolean variable will be set to a value of TRUE exactly once after the power is restored. If it is desirable to have the boolean variable set to a value of TRUE the next time that power fails, then the power status object must be reinserted in the power status queue.

Programming interfaces that support the power status object include:

> **KeInitializePowerStatus** - Initialize status object
> **KeInsertQueuePowerStatus** - Insert power status object
> **KeRemoveQueuePowerStatus** - Remove power status object

2.2.6.1 Initialize Power Status

A power status object can be initialized with the **KeInitializePowerStatus** function:

VOID
KeInitializePowerStatus (
    IN PKPOWER_STATUS *PowerStatus*
    );

Parameters:

> *PowerStatus* - A pointer to a control object of type power status.

The power status object data structure is initialized.

In order to actually have a boolean variable set to a value of TRUE when power is restored after a failure, the power status object must be inserted in the power status queue.

2.2.6.2 Insert Power Status

A power status object can be inserted in the power status queue with the **KeInsertQueuePowerStatus** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
KeInsertQueuePowerStatus (
    IN PKPOWER_STATUS *PowerStatus*,
    IN PBOOLEAN *Status*
    );

Parameters:

PowerStatus - A pointer to a control object of type power status.

Status - A pointer to a boolean variable that is to be set to a value of TRUE when power is restored after a failure.

If the specified power status object is already in the power status queue (a boolean state variable records whether the power status object is in the power status queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the power status object is inserted in the power status queue, the specified boolean variable is set to a value of FALSE, and a function value of TRUE is returned.

When the power is restored after a failure, the kernel removes each entry from the power status queue, sets the specified boolean variable to a value of TRUE, and sets the inserted state of the power status object to FALSE.

2.2.6.3 Remove Power Status

A power status object can be removed from the power status queue with the **KeRemoveQueuePowerStatus** function:

BOOLEAN
KeRemoveQueuePowerStatus (
    IN PKPOWER_STATUS *PowerStatus*
    );

Parameters:

PowerStatus - A pointer to a control object of type power status.

If the power status object is not in the power status queue (a boolean state variable records whether the power status object is in the power status queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the power status object is removed from the power status queue and a function value of TRUE is returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.7 Process Object

A *process object* represents the virtual address space and control information necessary for the execution of a set of thread objects.

A process object contains a pointer to an address map, a thread ready list to hold thread objects while the process is not in the balance set, a list of threads that are children of the process, the total accumulated time for all threads executing within the process, a base priority, and a default thread affinity.

A process object must be initialized before any thread objects can be initialized that specify the process as their parent.

A process is either in the balance set (Included) or not in the balance set (Excluded). When a process is in the balance set, then all threads that are children of the process are eligible to be considered for execution on a processor. When a process is not in the balance set, then necessary pages are not locked in memory (e.g. thread kernel stacks) and threads that are children of the process are not eligible for execution on a processor.

The balance set is managed by the balance set manager; see also the discussion under Thread Object.

A process cannot leave the balance set while any of its children threads own mutexes. Therefore, when a process is selected for removal from the balance set, any children threads that own mutexes are allowed to continue execution until they release their last mutex. When this occurs, execution of the thread is suspended and it is placed in the process ready queue rather than returning to one of the dispatcher ready queues. When no threads in the process own mutexes, then the process can actually be removed from the balance set.

Programming interfaces that support the process object include:

KeInitializeProcess - Initialize a process object
KeAttachProcess - Attach process address space
KeDetachProcess - Detach process address space
KeExcludeProcess - Exclude process from balance set
KeIncludeProcess - Include process in balance set
KeSetPriorityProcess - Set priority of process object

2.2.7.1 Initialize Process

A process object can be initialized with the **KeInitializeProcess** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
KeInitializeProcess (
    IN PKPROCESS *Process,*
    IN KPRIORITY *BasePriority*,
    IN KAFFINITY *Affinity*,
    IN ULONG *DirectoryTableBase*,
    IN BOOLEAN *Enable*
    );

Parameters:

Process - A pointer to a control object of type process.

BasePriority - The base priority of the process.

Affinity - The set of processors on which children threads of the process can execute.

DirectoryTableBase - The value that is to be loaded into the Directory Table Base Register when a child thread of the process is dispatched for execution.

Enable - A boolean variable that specifies the default handling mode for data alignment exceptions in children threads.

The process object data structure is initialized with the specified base priority, affinity, directory table base, and default alignment exception handling mode. The process and thread quantum values are initialized with system default values and the process is not considered to be in the balance set.

The *Enable* parameter specifies the default handling mode for data alignment exceptions in children threads. If this parameter is TRUE, then user mode data alignment exceptions are automatically handled by the kernel and are not raised as exceptions. Otherwise, user mode data alignment exceptions are not handled by the kernel and may, or may not, be raised as exceptions depending on host hardware capabilities. Automatic handling of user data alignment exceptions means that the kernel emulates misaligned data references and completes the offending instructions as if no misalignment exception had occurred. Misaligned references in kernel mode are never automatically handled and are always raised as exceptions.

IMPLEMENTATION NOTES:

Certain processors (e.g., the i386) always handle misaligned data in hardware. On these processors, enabling or disabling the automatic handling of data alignment exceptions has no effect. On other processors (e.g., i486, MIPS r3000, r4000SP, and

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

r4000MP) the handling of misaligned data is handled according to the mode established for the respective thread.

2.2.7.2 Attach Process

A thread can attach to another process's address space with the **KeAttachProcess** function:

VOID
KeAttachProcess (
    IN PKPROCESS *Process*
    );

Parameters:

Process - A pointer to a control object of type process.

Attaching to another process's address space causes the subject thread to leave the parent process's address space and enter the address space of the target process. This provides the capability for one thread to alter the address space and resources of another process without having complicated data structures or locking protocols.

All of the resources of the target process can be accessed and manipulated by the subject thread while the thread is executing in the target process's address space. This includes the process object table, process private mapping information, working set, etc.

A thread can only attach to one address space at a time. If an attempt is made to attach to a second process's address space while the thread is already attached to another process's address space, then a bug check will occur. In addition, a thread cannot own any mutexes when it attaches to another process's address space. An attempt to do will also cause a bug check to occur.

Attaching to another process's address causes the current APC state of the subject thread to be saved and a new state initialized. While the thread is executing in the attached process's address space, it can receive APCs that were initiated in that address space. APCs that were initiated in the parent process's address space are queued, but not delivered until the thread returns to the parent process's address space.

Attaching to another process's address space does not cause the kernel stack of the subject thread to be locked in memory while the target process is in the balance set. Therefore, both the source and target processes are not allowed to leave the balance set while a thread has the target process's address space attached. This is accomplished by incrementing the process mutex count of both the source and

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

target processes. Artificially incrementing this count prevents each of the processes from being removed from the balance set until their respective counts go to zero. In addition, a thread that has another process's address space attached is allowed to continue execution as if it owned a mutex, if one of the processes is selected for removal from the balance set by the balance set manager.

The execution time of a thread that has another process's address space attached is charged to the target process.

This service will be used by the executive to alter the address map of another process.

2.2.7.3 Detach Process

A thread can detach from another process's address space with the **KeDetachProcess** function:

```
VOID
KeDetachProcess (
    );
```

Detaching from another process's address space causes the subject thread to return to the parent process's address space.

If an attempt is made to detach from another process's address space when the subject thread does not have another process's address space attached, then a bug check will occur. In addition, if a kernel APC is in progress, the kernel APC queue contains an entry, the user APC queue contains an entry, or the thread owns one or more mutexes, then a bug check will also occur.

Detaching from another process's address space causes the saved APC state to be restored and the process mutex counts to be adjusted. If the kernel APC queue is not empty, then an APC_LEVEL software interrupt is requested which will cause the kernel mode APCs to get delivered as appropriate.

2.2.7.4 Exclude Process

A process object can be excluded from the balance set with the **KeExcludeProcess** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BOOLEAN
KeExcludeProcess (
    IN PKPROCESS *Process*
    );

Parameters:

Process - A pointer to a control object of type process.

The specified process is excluded from the balance set and its children threads will be removed from further consideration by the thread dispatcher when they no longer own any mutexes and do not have another process's address space attached.

A value of TRUE is returned as the function value, if the process can be immediately removed from the balance set (i.e., none of its children threads own any mutexes or have any address spaces attached). Otherwise, a value of FALSE is returned and the caller must wait on the process's balance set event to determine the exact point when the process can be removed from the balance set.

Processors on which children threads are running (Running state) or about to run (Standby state) are forced to redispatch if their respective threads do not own any mutexes and are not attached to another process's address space.

As ready threads are considered for execution, a test is made to determine if the thread's process is in the balance set. If the thread does not own any mutexes, is not attached to another process' address space, and its parent process is excluded from the balance set, then the thread is removed from its dispatcher ready queue and inserted in the process ready queue. The process ready queue is scanned when the process reenters the balance set and any threads in the process's ready queue are rereadied for execution.

This function is intended for use by the balance set manager.

2.2.7.5 Include Process

A process object can be included in the balance set with the **KeIncludeProcess** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
KeIncludeProcess (
    IN PKPROCESS *Process*
    );

Parameters:

Process - A pointer to a control object of type process.

The specified process is included in the balance set and the process's ready queue is scanned. The process ready queue is a list of threads that are ready to run, but which were moved to the process ready queue when they were encountered in one of the dispatcher ready queues. Each thread in the list is removed and readied for execution.

This function is intended for use by the balance set manager.

2.2.7.6 Set Priority Process

The base priority of a process object can be set with the **KeSetPriorityProcess** function:

KPRIORITY
KeSetPriorityProcess (
    IN PKPROCESS *Process*,
    IN KPRIORITY *BasePriority*
    );

Parameters:

Process - A pointer to a control object of type process.

BasePriority - The new base priority of the process object.

The base priority of the specified process is set to the specified value and the priority of all the process's children threads are adjusted as appropriate.

If the new priority is in the realtime class, then the priority of each child thread is set to the new base priority.

If the new priority is in the variable class, then the priority of each thread is computed by taking the current priority of the thread, subtracting out the old base priority, and then adding the new base priority. The computed value is not allowed to cross into the realtime class or go below a priority of 1.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.7.7 Process Accounting Data

As children threads within a process execute, their runtime is accumulated in the parent process object. The units of this summation are clock ticks.

2.2.8 Profile Object

A *profile object* provides the capability to measure the distribution of execution time within a block of code. Both user and system code may be profiled.

Each profile object has three key attributes. First, a profile object applies to an address range or a set of address ranges. The address range is specified by the *RangeBase*, *RangeSize*, and *Process* parameters. *RangeBase* and *RangeSize* select a range of bytes (that is, the area of code) on which to collect profile data. This range is within the address space described by the *Process* parameter. A given profile object either profiles a single address range within a single address space (i.e., applies to one process) or profiles the same single address range across all processes in the system.

Second, a profile object divides the address range being profiled into buckets. Each time a program counter (PC) sample shows the PC to be in one of these buckets, the corresponding counter is incremented. The *BucketSize* parameter controls the size of these buckets.

Third, a profile object reports the number of sampling hits for any given bucket in the corresponding counter. Counters reside in the buffer associated with the profile object when it is created.

Profiling works by sampling the processors PC using a periodic interrupt. The handler for the profiling interrupt searches the list of active profile objects for those with address ranges that match the PC. (I.e., the sampled PC falls within the address range associated with the profile object, and the current process matches the process associated with the profile object.) For each matching profile object, the bucket is computed, and the counter corresponding to the bucket is updated.

When profiling is off, it consumes no processor cycles, and thus may be present in any system. When turned on, the burden it places on the system is inversely proportional to the profiling interval set with **KeSetIntervalProfile** and proportional to the number of active (started) profile objects. A small number of profile objects may be active at any one time.

IMPLEMENTATION NOTE:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

On symmetric MP machines, profiling interrupts occur on all processors (at the same rate). On asymmetric machines (i.e., the SystemPro) slave processors do NOT do profiling.

Programming interfaces that support the profile object include:

KeInitializeProfile - Initialize a profile object
KeStartProfile - Start data collection for a profile object
KeStopProfile - Stop data collection for a profile object
KeSetIntervalProfile - Set length of profile interval (globally)
KeQueryIntervalProfile - Query length of profile interval

2.2.8.1 Initialize Profile

A profile object is initialized with **KeInitializeProfile.**

VOID
KeInitializeProfile (
    IN PKPROFILE *Profile*,
    IN PKPROCESS *Process* **OPTIONAL**,
    IN PVOID *RangeBase*,
    IN ULONG *RangeSize*,
    IN ULONG *BucketSize*
    );

Parameters:

Profile - A pointer to a control object of type profile.

Process - If specified, a pointer to a kernel process object that describes the address space to profile. If not specified, then all address spaces are included in the profile.

RangeBase - Address of the first byte of the address range for which profiling information is to be collected.

RangeSize - Size of the address range for which profiling information is to be collected. The *RangeBase* and *RangeSize* parameters are interpreted such that *RangeBase* <= address < *RangeBase*+*RangeSize* generates a profile hit.

BucketSize - Log base 2 of the size of a profiling bucket. Thus, *BucketSize* = 2 yields 4-byte buckets, *BucketSize* = 7 yields 128-byte buckets. All profile hits in a given bucket increment the corresponding counter in *Buffer*. Buckets cannot be smaller than a ULONG.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The profile object is initialized with the specified parameter values, and its state is set to stopped. **KeStartProfile** must be called to actually start profiling.

2.2.8.2 Start Profile

KeStartProfile must be called to start gathering data for a profile object.

BOOLEAN
KeStartProfile (
    IN PKPROFILE *Profile*,
    IN PULONG *Buffer*
    );

Parameters:

Profile - A pointer to a control object of type profile.

Buffer - Array of ULONGs. Each ULONG is a hit counter, which records the number of hits in the corresponding bucket. The *Buffer* must be accessible at DPC_LEVEL and above.

The value TRUE is returned if the profile object is successfully started. FALSE is returned if the object is already in the started state. An exception (STATUS_INSUFFICIENT_RESOURCES) is raised if there are insufficient resources available to make the profile active.

2.2.8.3 Stop Profile

KeStopProfile is called to stop gathering data for a profile object.

BOOLEAN
KeStopProfile (
    IN PKPROFILE *Profile*
    );

Parameters:

Profile - A pointer to a control object of type profile.

TRUE is returned if the profile is successfully stopped, FALSE if it is not already in the started state. Once a profile is stopped, no more updates are written into its buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.8.4 Set System Profile Interval

The time interval between profile interrupts (and thus the profiling rate) is set by calling **KeSetIntervalProfile.**

VOID
KeSetIntervalProfile (
    IN ULONG *Interval*
    );

Parameters:

Interval - The sampling interval in 100ns units.

The actual interval set by the system is the closest available, but may differ significantly. **KeQueryIntervalProfile** returns the actual value in use by the system.

The value is set globally; it affects all profiles on all processors.

IMPLEMENTATION NOTE:

PC-based i386 and i486 machines offer sampling intervals from about 10,000 units (1 millisecond) to 300 units (30 microseconds).

2.2.8.5 Query System Profile Interval

KeQueryIntervalProfile returns the current profile sampling interval.

ULONG
KeQueryIntervalProfile (
    );

The current profile sampling interval is returned in units of 100ns. This is the value the system is actually using, and thus may be different from the value set with **KeSetIntervalProfile**.

3. Wait Operations

Threads synchronize their access to dispatcher objects with object-specific functions and the generic kernel Wait functions. When a thread desires to wait until a dispatcher object attains a Signaled state, it executes one of the kernel Wait functions specifying the dispatcher object as a parameter. If the dispatcher object is not currently in a Signaled state, then the kernel puts the thread in a Waiting state and selects another thread to run on the current processor.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

At some future point, a cooperating thread or system operation will cause the specified dispatcher object to attain a state of Signaled. When this occurs, the thread will be given a priority boost and enter the Ready state. The thread will be dispatched for execution according to its priority.

The kernel Wait functions also allow a thread to wait on more than one dispatcher object at a time. The conditions under which the Wait will be satisfied can be specified as *WaitAny* or *WaitAll*.

If *WaitAny* is specified, then the Wait will be satisfied when any of the objects attain a state of Signaled. If *WaitAll* is specified, then the Wait will not be satisfied until all of the objects *concurrently* attain a state of Signaled.

Each Wait operation can optionally specify a timeout value. If a timeout value is specified, then the Wait will be automatically satisfied if the timeout period is exceeded without the Wait being satisfied in the normal manner.

If a timeout value of zero is specified, then no wait will actually occur, but an attempt will be made to satisfy the Wait immediately. If the Wait can be satisfied, then all side effects are performed (e.g. acquiring a mutex). Otherwise, no side effects are performed.

Wait operations can be alertable or nonalertable. If a wait is alertable and the subject thread is alerted while it is waiting, then the wait will be satisifed with a completion status of STATUS_ALERTED.

Wait operations also take a processor mode as a parameter which specifies on whose behalf the Wait is actually occurring. This is required since executive code itself performs the Wait operation and the previous mode of the processor is not necessarily the correct mode. This mode determines what happens when the subject thread is alerted or an APC is queued while the thread is in a Waiting state.

Each Wait operation also takes a Wait reason as a parameter. The Wait reason is an enumerated type supplied by the kernel and is used for debugging system code and for system management functions (i.e., it will be possible to display the reason a thread is in a Waiting state).

Programming interfaces that support wait operations include:

> **KeWaitForMultipleObjects** - Wait for dispatcher objects
> **KeWaitForSingleObject** - Wait for one dispatcher object

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.1 Wait For Multiple Objects

A thread can wait for a set of dispatcher objects with the **KeWaitForMultipleObjects** function:

NTSTATUS
KeWaitForMultipleObjects (
  IN CCHAR *Count*,
  IN PVOID *Objects[]*,
  IN WAIT_TYPE *WaitType*,
  IN KWAIT_REASON *WaitReason*,
  IN KPROCESSOR_MODE *WaitMode*,
  IN BOOLEAN *Alertable*,
  IN PTIME *Timeout* **OPTIONAL**,
  IN PKWAIT_BLOCK *WaitBlockArray* **OPTIONAL**
  );

Parameters:

Count - A count of the number of objects that are to be waited on.

Objects - An array of pointers to dispatcher objects.

WaitType - The type of wait operation that is to be performed (*WaitAny* or *WaitAll*).

WaitReason - The reason for the Wait.

WaitMode - The processor mode on whose behalf the Wait is occurring.

Alertable - A boolean value that specifies whether the Wait is alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which the Wait is to be completed.

WaitBlockArray - An optional pointer to an array of wait blocks that are to be used to describe the wait operation.

Each thread object has a builtin array of wait blocks that can be used to wait on multiple objects concurrently. Whenever possible, the builtin array of wait blocks should be used in a wait multiple operation since no additional wait block storage need be allocated and later deallocated. However, if the number of objects to be waited on concurrently is greater than the number of builtin wait blocks, then the *WaitBlockArray* parameter can be used to specify an alternate set of wait blocks to be used in the wait operation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the *WaitBlockArray* parameter is not specified, then the *Count* parameter must be less than or equal to THREAD_WAIT_OBJECTS which defines the number of builtin wait objects. If the *WaitBlockArray* parameter is not specified and the *Count* parameter is greater than THREAD_WAIT_BLOCKS, then a bug check will occur.

If the *WaitBlockArray* parameter is specified, then the *Count* parameter must be less than or equal to MAXIMUM_WAIT_OBJECTS which is the maximum number of objects that can be waited on concurrently. If the *WaitBlockParameter* is specified and the *Count* parameter is greater than MAXIMUM_WAIT_OBJECTS, then a bug check will occur.

The current state for each of the specified objects is examined to determine if the Wait can be satisfied immediately. If the Wait can be satisfied, then necessary side effects are performed on the objects and an appropriate value is returned as the function value. If the Wait cannot be satisfied immediately, and either no timeout value or a nonzero timeout value is specified, then the current thread is put in a Waiting state and a new thread is selected for execution on the current processor.

The *WaitType* parameter specifies the type of wait operation that is to be performed. If the *WaitType* is *WaitAll*, then all of the specified objects must attain a state of Signaled before the Wait will be satisfied. If the *WaitType* is *WaitAny*, then any of the objects must attain a state of Signaled before the Wait will be satisifed.

The reason for the Wait is set to the value specified by the *WaitReason* parameter.

The *WaitMode* parameter specifies on whose behalf the Wait is occurring.

The *Alertable* parameter specifies whether the thread can be alerted while it is in the Waiting state. If the value of this parameter is TRUE and the thread is alerted for a mode that is equal to or more privileged than the Wait mode, then the thread's Wait will be satisfied with a completion status of STATUS_ALERTED.

If the *WaitMode* parameter is *UserMode* and the *Alertable* parameter TRUE, then the thread can also be awakened to deliver a user mode APC. Kernel mode APCs always cause the subject thread to be awakened if the Wait IRQL is zero and there is not a kernel APC in progress.

The *Timeout* parameter is optional. If a timeout value is specified, then the Wait will be automatically satisfied if the timeout occurs before the specified Wait conditions are met.

If a zero timeout value is specified, then the Wait will not actually Wait regardless of whether it can be satisfied or not. An explicit timeout value of zero allows for the testing of a set of Wait conditions, and conditionally performing any side effects if the Wait can be immediately satisifed (e.g. the acquisition of a mutex).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The expiration time of the timeout is expressed as either an absolute time at which the Wait is to be automatically satisifed, or a time that is relative to the current system time. If the value of the *Timeout* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

The values returned by the **KeWaitForMultipleObjects** function determine how the Wait was satisfied.

A value in the range of zero to *Count* minus one is returned if the Wait is satisfied by one or more of the dispatcher objects specified by the *Objects* parameter and none of the dispatcher objects satisfying the Wait is an abandoned mutant object. The actual value returned is the index of the object (zero based) in the *Objects* array that satisfied the Wait.

A value in the range of STATUS_ABANDONED to STATUS_ABANDONED plus *Count* minus one is returned if the Wait is satisfied by one or more of the dispatcher objects specified by the *Objects* parameter and one or more of the dispatcher objects satisfying the Wait is an abandoned mutant object. The actual value returned is the index of the object (zero based) in the *Objects* array that satisfied the Wait plus the value of STATUS_ABANDONED.

A value of STATUS_ALERTED is returned if the Wait was completed because the thread was alerted.

If a value of STATUS_TIMEROUT is returned, then timeout occurred before the specified set of wait conditions were met. Note that this value can be returned when an explicit timeout value of zero is specified and the specified set of wait conditions cannot be immediately met.

A value of STATUS_USER_APC is returned if a user mode APC is to be delivered.

3.2 Wait For Single Object

A thread can wait for a single dispatcher object with the **KeWaitForSingleObject** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
KeWaitForSingleObject (
    IN PVOID *Object*,
    IN KWAIT_REASON *WaitReason*,
    IN KPROCESSOR_MODE *WaitMode*,
    IN BOOLEAN *Alertable*,
    IN PTIME *Timeout* **OPTIONAL**
    );

Parameters:

Object - A pointer to a dispatcher object.

WaitReason - The reason for the Wait.

WaitMode - The processor mode on whose behalf the Wait is occurring.

Alertable - A boolean value that specifies whether the Wait is alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which the Wait is to be completed.

The current state of the specified object is examined to determine if the Wait can be satisfied immediately. If the Wait can be satisfied, then necessary side effects are performed on the object and an appropriate value is returned as the function value. If the Wait cannot be satisfied immediately, and either no timeout value or a nonzero timeout value is specified, then the current thread is put in a Waiting state and a new thread is selected for execution on the current processor.

The reason for the Wait is set to the value specified by the *WaitReason* parameter.

The *WaitMode* parameter specifies on whose behalf the Wait is occurring.

The *Alertable* parameter specifies whether the thread can be alerted while it is in the Waiting state. If the value of this parameter is TRUE and the thread is alerted for a mode that is equal to or more privileged than the Wait mode, then the thread's Wait will be satisfied with a completion status of STATUS_ALERTED.

If the *WaitMode* parameter is *UserMode* and the *Alertable* parameter TRUE, then the thread can also be awakened to deliver a user mode APC. Kernel mode APCs always cause the subject thread to be awakened if the Wait IRQL is zero and there is not a kernel APC in progress.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The *Timeout* parameter is optional. If a timeout value is specified, then the Wait will be automatically satisfied if the timeout occurs before the specified Wait conditions are met.

If a zero timeout value is specified, then the Wait will not actually Wait regardless of whether it can be satisfied or not. An explicit timeout value of zero allows for the testing of a set of Wait conditions, and conditionally performing any side effects if the Wait can be immediately satisifed (e.g. the acquisition of a mutex).

The expiration time of the timeout is expressed as either an absolute time at which the Wait is to be automatically satisifed, or a time that is relative to the current system time. If the value of the *Timeout* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

The values returned by the **KeWaitForSingleObject** function determine how the Wait was satisfied.

A value of STATUS_SUCCESS is returned if the dispatcher object specified by the *Object* parameter satisfied the Wait.

A value of STATUS_ABANDONED is returned if the dispatcher object specified by the *Object* parameter satisfied the Wait and is a mutant object that was previously abandoned.

A value of STATUS_ALERTED is returned if the Wait was completed because the thread was alerted.

If a value of STATUS_TIMEOUT is returned, then timeout occurred before the specified wait condition was met. Note that this value can be returned when an explicit timeout value of zero is specified and the specified set of wait conditions cannot be immediately met.

A value of STATUS_USER_APC is returned if a user mode APC is to be delivered.

4. Miscellaneous Operations

Several miscellaneous functions are provided to perform hardware-related operations and to provide the operations necessary to debug a multiprocessor operating system.

The exact implementation for some of these operations varies, depending on the particular host architecture. Implementation notes have been provided for these functions.

Programming interfaces that support miscellaneous operations include:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

KeBugCheck - Generate bug check halt
KeContextFromKframes - Move machine state to context frames
KeContextToKframes - Move machine state from context frames
KeFillEntryTb - Fill translation buffer entry
KeFlushDcache - Flush data cache
KeFlushEntireTb - Flush entire translation buffer
KeFlushIcache - Flush instruction cache
KeFlushIoBuffers - Flush I/O buffers from the data cache
KeFlushSingleTb - Flush single translation buffer entry
KeFreezeExecution - Freeze processor execution
KeGetCurrentApcEnvironment - Get the current APC environment
KeGetCurrentIrql - Get the current IRQL
KeGetPreviousMode - Get previous processor mode
KeLowerIrql - Lower the current IRQL to the specified value
KeQuerySystemTime - Query the current system time
KeRaiseIrql - Raise the current IRQL to the specified value
KeRundownThread - Run down thread before termination
KeSetSystemTime - Set the current system time
KeStallExecutionProcessor - Stall processor execution
KeUnFreezeExecution - Unfreeze processor execution

4.1 Bug Check

A bug check halt can be generated with the **KeBugCheck** function:

VOID
KeBugCheck (
    IN ULONG *BugCheckCode*
    );

Parameters:

BugCheckCode - A value that specifies the reason for the bug check.

A bug check is a system-detected error that causes a controlled shutdown of the system. The various kernel mode components of the system perform online consistency checking. When an inconsistency is discovered, a bug check is generated.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2 Context Frame Manipulation

The kernel trap handler is responsible for saving and restoring machine state when an interrupt, exception, system call, or other trapping condition is detected by system hardware.

Depending on the type of trapping condition, the trap handler may save only the volatile register state, or may save both the volatile and the nonvolatile register state. In addition, the previous processor state and floating point status are also saved.

This state information is saved on the kernel stack in the form of a call frame. Separate call frames are used to store the volatile and the nonvolatile register state. These frames are called the trap frame and exception frame respectively.

A third structure, called a context frame, is constructed from the information contained in the trap and exception frames. This structure contains the complete machine state for a thread of execution.

A context frame is used to specify the initial machine state of a thread, to save the previous machine state when an exception handler is invoked, and to continue the execution of a thread after an exception has been handled.

The kernel supplies two routines to marshal information to/from a context frame.

4.2.1 Move Machine State To Context Frame

Saved machine state can be moved from a trap frame and/or an exception frame to a context frame with the **KeContextFromKframes** function:

VOID
KeContextFromKframes (
    IN PKTRAP_FRAME *TrapFrame*,
    IN PKEXCEPTION_FRAME *ExceptionFrame*,
    IN OUT PCONTEXT *ContextFrame*
    );

Parameters:

TrapFrame - A pointer to a trap frame.

ExceptionFrame - A pointer to an exception frame.

ContextFrame - A pointer to a context frame.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Saved machine state is moved from the specified trap frame and/or the specified exception frame to the specified context frame. The *ContextFlags* field of the context frame controls the information that is moved.

ContextFlags Field

CONTEXT_CONTROL - Specifies that the processor state information from the trap frame is to be moved to the context frame.

CONTEXT_FLOATING_POINT - Specifies that the floating point register state from the trap and exception frames is to be moved to the context frame.

CONTEXT_INTEGER - Specifies that the integer register state from the trap and exception frames is to be moved to the context frame.

CONTEXT_PIPELINE - Specifies that the floating point pipe state is to be moved from the trap frame to the context frame.

CONTEXT_FULL - Specifies that all of the state information from the trap and exception frames is to be moved to the context frame.

4.2.2 Move Machine State From Context Frame

Saved machine state can be moved from a context frame to a trap frame and/or an exception frame with the **KeContextToKframes** function:

VOID
KeContextToKframes (
    IN OUT PKTRAP_FRAME *TrapFrame*,
    IN OUT PKEXCEPTION_FRAME *ExceptionFrame*,
    IN PCONTEXT *ContextFrame*,
    IN ULONG *ContextFlags*,
    IN KPROCESSOR_MODE *PreviousMode*
    );

Parameters:

TrapFrame - A pointer to a trap frame.

ExceptionFrame - A pointer to an exception frame.

ContextFrame - A pointer to a context frame.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ContextFlags - A set of flags that specifies the state information that is to be moved from the specified context frame to the specified trap frame and/or the specified exception frame.

ContextFlags Flags

CONTEXT_CONTROL - Specifies that the processor state information from the context frame is to be moved to the trap frame.

CONTEXT_FLOATING_POINT - Specifies that the floating point register state from the context frame is to be moved to the trap and exception frames.

CONTEXT_INTEGER - Specifies that the integer register state from the context frame is to be moved to the trap and exception frames.

CONTEXT_PIPELINE - Specifies that the floating point pipe state is to be moved from the context frame to the trap frame.

CONTEXT_FULL - Specifies that all of the state information from the context frame is to be moved to the trap and exception frames.

PreviousMode - The processor mode for which the context frame is specified.

Saved machine state is moved from the specified context frame to the specified trap frame and/or the specified exception frame. The *ContextFlags* parameter specifies the information that is to be moved. The *PreviousMode* parameter determines which bits the caller may specify if the processor state information is being moved to the trap frame.

4.3 Fill Entry Translation Buffer

A page table entry can be inserted into the translation buffer of the current processor with the **KeFillEntryTb** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
KeFillEntryTb (
    IN HARDWARE_PTE *Pte[1]*,
    IN PVOID *Virtual*,
    IN BOOLEAN *Invalid*
    );

Parameters:

Pte - A pointer to a page table entry, or a pair of page table entries, that are to be inserted into the translation buffer of the current processor.

Virtual - The virtual address that corresponds to the first page table entry.

Invalid - A boolean value that determines whether a translation buffer entry should be invalidated if the host architecture does not provide a software-managed translation buffer.

This function is intended for use by memory management software for the following cases:

1. A page table entry transitions from the invalid to the valid state.

2. A page table entry transitions from the unmodified (clean) to the modified (dirty) state.

3. A page table entry transitions from the unaccessed to the accessed state.

None of these transitions affects other processors in the configuration; however, they provide the opportunity to optimize the filling of the translation buffer on systems that have a software-managed translation buffer.

If the page table entry is transitioning from the invalid to the valid state, then the *Invalid* parameter should be FALSE. Otherwise, the *Invalid* parameter should be TRUE.

If the specified virtual address is already mapped by the translation buffer, then the contents of the specified page table entry(s) replace the page table entry in the translation buffer. Otherwise, a new translation buffer entry is created that maps the specified virtual address.

IMPLEMENTATION NOTES:

The Intel i860 does not have a software-managed translation buffer. It also cannot invalidate a single translation buffer entry. Therefore, if the *Invalid* parameter is

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

TRUE, then the entire translation buffer is invalidated. Otherwise, no operation is performed.

The Intel i386 and i486 do not have a software-managed translation buffer. Also neither of these processors can invalidate a single translation buffer entry. Therefore, if the *Invalid* parameter is TRUE, the entire translation buffer is invalidated. Otherwise, no operation is performed.

> \ *The i486 can invalidate a single translation buffer entry, but it is not yet supported.* \

The MIPS r3000, r4000SP, and r4000MP have software-managed translation buffers. Therefore, the specified page table entry either replaces the current translation buffer entry or a new translation buffer entry is created to map the specified virtual address.

4.4 Flush Data Cache

The data cache can be flushed on all processors, or only that set of processors that are currently executing threads that belong to the current thread's process with the **KeFlushDcache** function:

VOID
KeFlushDcache (
    IN BOOLEAN *AllProcessors*
    );

Parameters:

AllProcessors - A boolean value that determines which data caches are to be flushed.

This function is intended for use by memory management and device driver software to keep the data cache coherent with DMA I/O operations.

If the *AllProcessors* parameter is TRUE, then the data cache is flushed on all processors in the system. Otherwise, only the data caches on processors running threads that belong to the current thread's process are flushed.

IMPLEMENTATION NOTES:

The Intel i860 employs a writeback data cache with virtual tags that does not maintain coherency with DMA I/O operations. Therefore, this function must flush the data cache.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The Intel i386 and i486 employ data caches that maintain coherency with I/O operations. Therefore, this function performs no operation.

The MIPS r3000 and r4000SP employ data caches that do not maintain coherency with I/O operations. Therefore, the data cache must be flushed for this function.

The MIPS r4000MP employs a data cache that maintains coherency with I/O operations. Therefore, this function performs no operation.

4.5 Flush Entire Translation Buffer

The entire translation buffer can be flushed on all processors, or only that set of processors that are currently executing threads that belong to the current thread's process with the **KeFlushEntireTb** function:

VOID
KeFlushEntireTb (
    IN BOOLEAN *Invalid*,
    IN BOOLEAN *AllProcessors*
    );

Parameters:

Invalid - A boolean value that specifies why the translation buffer is being flushed.

AllProcessors - A boolean value that determines which translation buffers are to be flushed.

This function is intended for use by memory management software when virtual pages are deleted, removed from the process working set, or their protection is changed. Normally, the entire translation buffer is not flushed when virtual pages are removed from the process working set. However, when a number of pages are removed all at once, it is more efficient to simply flush the entire translation buffer rather than flush individual entries.

If the value of the *Invalid* parameter is TRUE, then the translation buffer is being flushed because one or more pages have become invalid and not present in memory. If the value of the *Invalid* parameter is FALSE, then the translation buffer is being flushed because the protection on one or more pages has been changed.

If the *AllProcessors* parameter is TRUE, then the entire translation buffer is flushed on all processors in the system. Otherwise, only the translation buffers on processors running threads that belong to the current thread's process are flushed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IMPLEMENTATION NOTE:

The Intel i860 employs a data cache with virtual tags. It also cannot flush the translation buffer without also flushing the instruction cache. If the *Invalid* parameter is TRUE, then the data cache is flushed in addition to flushing the instruction cache and invalidating the translation buffer. Otherwise, the instruction cache is flushed and the translation buffer is invalidated.

The Intel i386 and i486 flush the translation buffer for this function.

The MIPS r3000, r4000SP, and r4000MP flush the random part of the software-managed translation buffer for this function. The fixed part of the translation buffer is not affected.

4.6 Flush Instruction Cache

The instruction cache can be flushed on all processors, or only that set of processors that are currently executing threads that belong to the current thread's process with the **KeFlushIcache** function:

VOID
KeFlushIcache (
  IN BOOLEAN *AllProcessors*
  );

Parameters:

AllProcessors - A boolean value that determines which instruction caches are to be flushed.

This function is intended for use by system debuggers. When a breakpoint is inserted in system code, the instruction caches of all processors in the system must be flushed. If a breakpoint is placed in process code, then only the instruction caches of processors executing threads that belong to current thread's process need to be flushed.

The executive also exports this function for use by code that modifies the instruction stream. After each such modification, and before attempting to execute the modified instructions, the instruction cache must be flushed.

If the *AllProcessors* parameter is TRUE, then the instruction cache is flushed on all processors in the system. Otherwise, only the instruction caches on processors running threads that belong to the current thread's process are flushed.

IMPLEMENTATION NOTES:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The Intel i860 does not maintain coherency between the data and instruction caches. It also cannot flush the instruction cache without invalidating the translation buffer. Therefore, the instruction cache is flushed and the translation buffer is invalidated for this function.

The Intel i386 and i486 maintain coherency between the data and instruction caches. Therefore, no operation is performed for this function.

The MIPS r3000, r4000SP, and r4000MP do not maintain coherency between the instruction and data caches. Therefore, the instruction cache is flushed for this function.

4.7 Flush I/O Buffers

The memory region occupied by an I/O buffer can be flushed from both the instruction and data caches of all processors in the system with the **KeFlushIoBuffers** function:

VOID
KeFlushIoBuffers (
    IN PMDL *Mdl*,
    IN BOOLEAN *ReadOperation*
    );

Parameters:

Mdl - A pointer to a memory descriptor list that describes the areas of memory occupied by the I/O buffer.

ReadOperation - A boolean value that determines whether the flush is being performed for a read operation.

This function is intended for use by device drivers and affects all processors in the system.

If the *ReadOperation* parameter is TRUE, then the I/O operation is reading information into memory that may be valid in the instruction and data caches. If the *ReadOperation* parameter is FALSE, then the I/O operation is writing data from memory to a device and information may be present in the data cache and not in memory.

IMPLEMENTATION NOTES:

The Intel i860 employs a writeback data cache and an instruction cache that do not maintain coherency with I/O operations. Therefore, the data cache must be flushed

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

for both read and write operations. The Intel i860 also cannot flush the instruction cache without invalidating the translation buffer. Therefore, if the *ReadOperation* parameter is TRUE, then the instruction cache is flushed and the translation buffer is also invalidated for this function.

The Intel i386 and i486 maintain data and instruction cache coherency with I/O operations. Therefore, no operation is performed for this function.

> \ *The i486 has a write buffer which may have to be flushed before all I/O operations.* \

The MIPS r3000 employs a write-through data cache and does not maintain coherency with I/O operations for either of the instruction or data caches. Therefore, if the *ReadOperation* parameter is TRUE, then both the instruction and data caches must be flushed. Otherwise, no operation is performed for this function.

> \ *The r3000 has a write buffer which must be flushed before all I/O operations.* \

The MIPS r4000SP employs a writeback data cache and an instruction cache that do not maintain coherency with I/O operations. Therefore, the data cache must be flushed for both read and write operations. In addition, if the *ReadOperation* parameter is TRUE, then the instruction cache is also flushed for this function.

The MIPS r4000MP employs a writeback data cache that maintains coherency with I/O operations. However, cache coherency is not maintained for the instruction cache with I/O operations. Therefore, if the *ReadOperation* parameter is TRUE, then the instruction cache is flushed. Otherwise, no operation is performed for this function.

4.8 Flush Single Translation Buffer Entry

A single entry can be flushed from the translation buffer of all processors, or only that set of processors that are currently executing threads that belong to the current thread's process with the **KeFlushSingleTb** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
HARDWARE_PTE
KeFlushSingleTb (
    IN PVOID Virtual,
    IN BOOLEAN Invalid,
    IN BOOLEAN AllProcessors,
    IN PHARDWARE_PTE PtePointer,
  IN HARDWARE_PTE PteValue
  );
```

Parameters:

Virtual - A virtual address that is within the page whose translation buffer entry is to be flushed.

Invalid - A boolean value that specifies why the translation buffer is being flushed.

AllProcessors - A boolean value that determines which translation buffers are to be flushed.

PtePointer - A Pointer to a page table entry which is to be updated with the new PteValue.

PteValue - The new Pte value.

Return Value:

The contents of the page table entry *PtePointer* refers to before the entry is set to *PteValue*.

This function is intended for use by virtual memory management software when a virtual page is deleted, removed from the process working set, or its protection is changed. If several virtual pages are removed from a process's address space at once or their protection is changed, then it may be more efficient to use the **KeFlushEntireTb** function.

If the value of the *Invalid* parameter is TRUE, then the translation buffer is being flushed because a page has become invalid and is not present in memory. If the value of the *Invalid* parameter is FALSE, then the translation buffer is being flushed because the protection on a page has been changed.

If the *AllProcessors* parameter is TRUE, then the specified translation buffer entry is flushed on all processors in the system. Otherwise, only the specified translation

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

buffer entry on process running threads that belong to the current thread's processor are flushed.

IMPLEMENTATION NOTE:

The Intel i860 employs a data cache with virtual tags. It also cannot invalidate a single entry from the translation buffer nor can it invalidate the translation buffer without also flushing the instruction cache. If the *Invalid* parameter is TRUE, then the data cache is flushed in addition to flushing the instruction cache and invalidating the translation buffer. Otherwise, the instruction cache is flushed and the translation buffer is invalidated.

The Intel i386 and i486 cannot flush a single entry from the translation buffer. Therefore, the entire translation buffer is invalidated for this function.

> \ *The i486 can invalidate a single translation buffer entry, but it is not yet supported.* \

The MIPS r3000, r4000SP, and r4000MP provide the capability to flush a single entry from the random part of the software-managed translation buffer. Therefore, a single translation buffer entry is invalidated for this function.

4.9 Freeze Execution

The execution of all other processors in the system, excluding the current processor, can be frozen with the **KeFreezeExecution** function:

KIRQL
KeFreezeExecution (
    );

The IRQL is raised to the highest level, the execution of all other processors in the host configuration is frozen, and the previous IRQL is returned as the function value.

This function does not return control to the caller until the execution of all other processors has been frozen. It is intended for use by system debuggers and should be called whenever the debugger is entered so that a consistent picture of the multiprocessor system can be examined and modified.

4.10 Get Current APC Environment

The APC execution environment for the current thread can be obtained with the **KeGetCurrentApcEnvironment** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

KAPC_ENVIRONMENT
KeGetCurrentApcEnvironment (
    );

The APC execution environment is obtained from the current thread and returned as the function value.

Possible values that can be returned by this function include:

- o OriginalApcEnvironment - The current APC environment is the thread's parent process.
- o AttachedApcEnvironment - The current APC environment is a process that has been attached by the current thread.

4.11 Get Current IRQL

The current IRQL can be obtained with the **KeGetCurrentIrql** function:

KIRQL
KeGetCurrentIrql (
    );

The current IRQL is returned as the function value.

4.12 Get Previous Mode

The previous processor mode can be obtained with the **KeGetPreviousMode** function:

KPROCESSOR_MODE
KeGetPreviousMode (
    );

The previous processor mode is obtained from the processor status. This function can be used to determine the previous processor mode during a system service.

4.13 Lower IRQL

The current IRQL can be lowered with the **KeLowerIrql** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
KeLowerIrql (
    IN KIRQL *NewIrql*
    );

Parameters:

NewIrql - The new IRQL value.

If the new IRQL is greater than the current IRQL, then a bug check will occur. Otherwise, the current IRQL is set to the specified value.

4.14 Query System Time

The current system time can be queried with the **KeQuerySystemTime** function:

VOID
KeQuerySystemTime (
    OUT PTIME *CurrentTime*
    );

Parameters:

CurrentTime - A pointer to a variable that receives the current system time.

This function returns the current system time in 100ns units. It is the responsibility of the executive to maintain the correspondence between system time and external time as seen by a user of the system.

4.15 Raise IRQL

The current **IRQL** can be raised with the **KeRaiseIrql** function:

KIRQL
KeRaiseIrql (
    IN KIRQL *NewIrql*
    );

Parameters:

NewIrql - The new IRQL value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the new IRQL is less than the current IRQL, then a bug check will occur. Otherwise, the current IRQL is set to the specified value.

4.16 Run Down Thread

Data structures for the current thread that must be guarded by the dispatcher database lock can be run down with the **KeRundownThread** function:

VOID
KeRundownThread (
    );

This function is intended for use just prior to terminating a thread. It run downs appropriate data structures and performs operations necessary to terminate the thread.

Operations performed include:

1. Processing of the mutant ownership list which causes each mutant object owned by the current thread to be released with an abandoned status.

4.17 Set System Time

The current system time can be set with the **KeSetSystemTime** function:

VOID
KeSetSystemTime (
    IN PTIME *NewTime*,
    OUT PTIME *OldTime*,
    );

Parameters:

NewTime - A pointer to a variable that specifies the new system time.

OldTime - A pointer to a variable that receives the previous system time.

This function returns the previous system time in 100ns units and sets the system time to the specified value. It is the responsibility of the executive to maintain the correspondence between system time and external time as seen by a user of the system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.18 Stall Execution

The execution of the current processor can be stalled with the **KeStallExecutionProcessor** function:

VOID
KeStallExecutionProcessor (
    IN ULONG *MicroSeconds*
    );

Parameters:

MicroSeconds - The number of microseconds for which execution is to be stalled.

This function stalls the execution of the current processor by executing a processor-dependent routine that busy waits at least the specified number of microseconds, but not significantly longer.

This routine is intended for use by device drivers and other software that must wait a short interval which is less than a clock tick, but larger than a few instructions.

IMPLEMENTATION NOTES:

This function is guaranteed to busy wait for at least the number of specified microseconds and is calibrated at system initialization. Long intervals tend to be very accurate, whereas, short intervals may busy wait for a period that is slightly longer than the specified number of microseconds.

4.19 Unfreeze Execution

The execution of all other processors in a host configuration, excluding the current processor, can be resumed with the **KeUnfreezeExecution** function:

VOID
KeUnfreezeExecution (
    IN KIRQL *Irql*
    );

Parameters:

Irql - The previous IRQL value that is to be restored.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The execution of all processors in the system, excluding the current processor, is unfrozen, the previous IRQL is restored, and the instruction cache of each processor in the configuration is flushed.

This function is intended for use by system debuggers and should be called when execution is to be continued after entering the debugger and calling **KeFreezeExecution** function. Before the execution of an unfrozen processor is continued, its instruction cache and translation buffer are flushed.

5. Intel x86 Specific Functions.

There is a small set of special functions peculiar to the Intel x86 family of processors, which are necessary to fully exploit those processors. These functions are used primarily to manipulate x86 specific control structures, such as the Ldt.

Programming interfaces:

Ke386SetLdtProcess - Set Ldt for a process
Ke386SetDescriptorProcess - Set entry in Ldt a process

5.1 Load an Ldt for a process.

An Ldt (Local Descriptor Table) can be made the active Ldt for a process with Ke386SetLdtProcess:

```
VOID
Ke386SetLdtProcess (
    PKPROCESS Process,
    PLDT_ENTRY Ldt[],
    ULONG       Limit
    );
```

Parameters:

Process - Pointer to KPROCESS object describing the process for which the Ldt is to be set.

Ldt - Pointer to an array of LDT_ENTRYs (that is, a pointer to an Ldt.

Limit - Ldt limit (must be 0 mod 8)

The specified LDT (which may be null) will be made the active Ldt of the specified process, for all threads thereof, on whichever processors they are running. The change will take effect before the call returns.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

An Ldt address of NULL or a Limit of 0 will cause the process to receive the NULL Ldt.

This function only exists on i386 and i386 compatible processors.

No checking is done on the validity of Ldt entries.

IMPLEMENATION NOTES:

While a single Ldt structure can be shared among processes, any edits to the Ldt of one of those processes will only be synchronized for that process. Thus, processes other than the one the change is applied to may not see the change correctly.

5.2 Set and Entry in a Process's Ldt.

An individual entry in the Ldt of a process may be edited with Ke386SetDescriptorProcess:

```
VOID
Ke386SetDescriptorProcess (
      PKPROCESS Process,
      ULONG        Offset,
      LDT_ENTRY  LdtEntry
      );
```

Parameters:

Process - Pointer to KPROCESS object describing the process for which the descriptor edit is to be performed.

Offset - Byte offset into the Ldt of the descriptor to edit. Must be 0 mod 8.

LdtEntry - Value to edit into the descriptor in hardware format. No checking is done on the validity of this item.

The specified LdtEntry (which could be 0, not present, etc) will be edited into the specified Offset in the Ldt of the specified Process. This will be synchronized across all the processors executing the process. The edit will take affect on all processors before the call returns.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Get an Entry from a Thread's Gdt.

Get an Entry from a Thread's Gdt.

5.3 Get an Entry from a Thread's Gdt.

An individual entry in the Gdt of a thread may be obtained using Ke386GetGdtEntryThread:

VOID
Ke386GetGdtEntryThread (
  IN PKTHREAD *Thread*,
  IN ULONG *Offset*,
  IN PGDT_ENTRY *Descriptor*
  );

Parameters:

Thread — Supplies a pointer to the thread from whose Gdt the entry is to come.

Offset — Supplies the descriptor number of the descriptor to return. This value must be 0 mod 8.

Descriptor — Returns the descriptor contents

The Gdt entry specified by *Selector* will be copied from the specified thread's Gdt, into *Descriptor*.

For descriptors that don't exist when the thread is not running (KGDT_R3_TEB, and KGDT_LDT), the descriptor values will be "materialized".

For descriptors that are processor specific, rather than thread specific, the current processor's value will be returned.

For all other Gdt descriptors, the descriptor will be copied from the Gdt.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, March 8, 1989

Revision 1.1, March 16, 1989

1. Add text to describe the muxwait object.
2. Add text to describe the interrupt object.
3. Add text to describe the power notify object.
4. Add text to describe the power status object.
5. Add text to describe the generic wait functions.
6. Addition of text to describe the miscellaneous functions.
7. Add text to overview of document.

Revision 1.2, March 29, 1989

1. Change **KeDelayExecution** to return a wait completion value.
2. Complete section on multiprocessor synchronization.
3. Complete section on device queue object.
4. Delete muxwait object and replace with a wait multiple function that takes an array of pointer to dispatcher objects as a parameter.

Revision 1.3, April 18,1989

1. Alphabetically order section on miscellaneous functions.
2. Add **KeBugCheck**, **KeLowerIrql**, and **KeRaiseIrql** miscellaneous functions.
3. A thread will start execution at IRQL APC_LEVEL rather than with APCs disabled.
4. Returning from the executive thread start up routine will cause a thread to enter user mode provide that a user mode context was supplied when the thread was initialized.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Add three parameters to thread initialization to optionally describe user mode context.

6. Delete builtin user mode alert APC. Alerting a thread that is waiting alertable causes a wait completion status of ALERTED to be returned.

7. Replace all reference to DPC_LEVEL with DISPATCH_LEVEL.

8. Put interrupt level names in hardware interrupt table.

9. Add pointers to the PRCB which point to the time expiration and power notify DPC's that are system wide.

10. Change thread context to include ten pipeline state registers rather than six.

11. Change **KeResumeThread** and **KeSuspendThread** to return a CHAR rather than a UCHAR.

12. Delete voluntary and preemption wait counters from thread object.

13. Reduce number of APC and DPC system parameters to two.

14. If a device is Not-Busy, then release device queue spin lock but do not lower IRQL before returning.

15. Allow interrupt service routine to user the floating point registers for block moves and graphics functions.

16. If a thread is awakened to deliver a user mode APC, then return a status of USER_APC.

Revision 1.4, May 4, 1989

1. Delete increment parameter from **KeReleaseMutex**.

2. Change *Count* and *Limit* parameters of **KeInitializeSemaphore** from ULONG to LONG.

3. Change **KeReadStateSemaphore** to return a LONG rather than a ULONG.

4. Change **KeReleaseSemaphore** to return a LONG rather than a ULONG and change the type of the *Adjustment* parameters from ULONG to LONG.

5. Set the value of a semaphore to the maximum value if an attempt is made to adjust the count of a semaphore above the limit.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6. Add system startup routine to **KeInitializeThread** for executive level initialization.

7. Change the wait functions to return NTSTATUS rather than ULONG.

8. Change the type of the *WaitType* parameter from KWAIT_TYPE to WAIT_TYPE.

Revision 1.5, May 8, 1989

1. Change data type of the **KeBugCheck** parameter to ULONG.

2. Add *Invalid* parameter to **KeFlushEntireTb** and **KeFlushSingleTb** to allow specification of why the flush is being performed.

Revision 1.6, August 14, 1989

1. General correction of typos and grammatical errors as suggested by Helenc review. Clarification and rewrite of several sections.

2. Reorganization of section 1.0 with the deletion of redundant information.

3. Added miscellaneous functions to get the previous processor mode, get the current IRQL, and to set the current IRQL.

4. The interrupt object section was extensively rewritten to match the actual implementation.

5. The time parameter in the **KeDelayExecution** function was changed to a pointer to a time value.

Revision 1.7, November 15, 1989

1. Delete **KeSetCurrentIrql** function which was an optimization of the **KeRaiseIrql** function that didn't require saving the old **IRQL**.

2. Add functions **KeContextToKframes** and **KeContextFromKframes** to move information in the trap frame and the exception frame to/from the context frame.

3. Add priority increment parameter to the **KeInsertQueueApc** function.

4. Change **KeInitializeThread** function to replace the optional trap and exception frame parameters with an optional context frame argument. If the context frame parameter is specified, then it is assumed that the thread will execute in user mode.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Change **KeAcquireSpinlock** function to return the old IRQL as an output parameter and delete the Wait parameter from the **KeReleaseSpinlock** function.

6. Change **KeInsertDeviceQueue** and **KeInsertByKeyDevice** functions to neither raise nor lower IRQL.

7. Split the event object into two types of event objects: synchronization and notification.

8. Change the definition of the state of an event object to be a count.

9. Add a parameter to the **KeInitializeApc** function to specify the APC execution environment. Add a function (**KeGetApcEnvironment**) that returns the current APC environment.

Revision 1.8, November 16, 1989

1. Add optional parameter to **KeWaitForMultipleObjects** that allows more than the builtin number of objects to be waited on concurrently.

2. Add abandoned return status from **KeWaitForMultipleObjects** when one or more of the dispatcher objects satisfying the Wait is an abandoned mutant object.

3. Add new mutant object which provides for user level mutexes. Add the functions **KeInitializeMutant**, **KeReadStateMutant**, and **KeReleaseMutant** to manipulate the mutant object.

Revision 1.9, November 17, 1989

1. Add **KeRundownThread** function to provide kernel thread rundown when a thread is deleted.

2. Minor edits and corrections.

Revision 1.10, January 6, 1990

1. Change name of **KeReadSystemTime** to **KeQuerySystemTime**.

2. Add miscellaneous kernel function (**KeSetSystemTime**) to set the system time.

Revision 1.11, June 6, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Change text that explains how a binary semaphore can be used like a synchronization event to include a statement that the semaphore cannot be over Signaled.

2. Change the definition of **KeInitializeSemaphore** to omit any checks on the limit and initial value of the semaphore.

3. Change the semantics of release semaphore such that an attempt to over Signal the semaphore does not cause the current count to be set to the maximum value. The current count remains unchanged and the exception STATUS_SEMAPHORE_COUNT_EXCEEDED is raised.

4. Once set, the abandoned status of a mutant object cannot be cleared and will continually be returned by the kernel wait services. The mutant object, however, will continue to function and ownership can be requested and released.

5. If an attempt is made to release a mutant object by a nonowner with the Abandoned parameter FALSE, then either the exception STATUS_ABANDONED or STATUS_MUTANT_NOT_OWNED will be raised.

6. Remove the hard assignment of Interrupt Request Levels (IRQL's) to kernel functions and replace with symbolic assignments. Explain that IRQL's are hierarchically ordered by priority.

7. Change the return type of **KeResumeThread** and **KeSuspendThread** to ULONG.

8. An attempt to suspend a thread more than MAXIMUM_SUSPEND_COUNT times causes the exception STATUS_SUSPEND_COUNT_EXCEEDED to be raised.

9. Add **KeFreezeThread** and **KeUnfreezeThread** functions to suspend and resume a thread on behalf of the system. These functions are identical to suspend and resume, but are not exported to user mode.

10. Add **KeRundownThread** function to run down appropriate kernel data structures before thread termination.

11. Add **KeStallExecution** function to enable executive software to stall execution for short periods of time.

12. Add **KeFlushIoBuffers** function to flush the memory region occupied by an I/O buffer from the data and instruction caches.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

13. Remove text that described the **KeFlushIcache** and **KeFlushDcache** functions as intended for use in systems in which DMA I/O operations do not invalidate caches.

14. Add **KeFillEntryTb** function to update TB entries in systems with software-managed translation buffers.

15. Make explanation of thread context more general and not specific to the Intel i860.

Revision 1.12, September 19, 1990 (Bryan Willman)

1. Add Profile object section.

Revision 1.13, March 11, 1991

1. Change the operation of the power notify object so that a DPC object is not required and make the operation repeatable.

2. Add parameter to **KeInitializeProcess** to specific the default data alignment handling mode for children threads.

3. Add **KeSetAutoAlignmentThread** and **KeQueryAutoAlignmentThread** functions to set and query the data alignment handling mode of the current thread.

4. Change the name of the **KeDelayExecution** function to **KeDelayExecutionThread** and move the explanatory text to the section on thread objects.

5. Change the name of the **KeStallExecution** function to **KeStallExecutionProcessor**.

6. Change REQUEST_LEVEL with IPI_LEVEL.

7. Add **KeQueryBasePriorityThread** and **KeSetBasePriorityThread** functions to set and query the base priority of a thread object.

Revision 1.14, May 2, 1991

1. Add x86 specific section, Ke386SetLdtProcess and Ke386SetDescriptorProcess.

Revision 1.15, May 28, 1991 (daveh)

1. Added Ke386GetThreadGdt

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision 1.16, June 18, 1991 (bryanwi)

1. Added ShareVector parameter to KeInitializeInterrupt.
2. Applied spelling checker to document.

Revision 1.17, August 7, 1991 (shielint)

1. Made KeFlushSingleTb spec match reality

Revision 1.18, August 8, 1991 (bryanwi)

1. Removed coordinator, added SynchronizeIrql to KeInitializeInterrupt.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Mailslot Specification

Author: *Manny Weiser*

Original Draft December 28, 1990
Revision 1.1, January 10, 1991
Revision 1.2, March 11, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction....1

2. Goals....1

3. Overview of OS/2 Mailslots....1

4. Overview of NT OS/2 Mailslots....3
4.1 Implementation Alternatives....3
4.2 Read/Write Buffering Strategy....4
4.2.1 OS/2 Read/Write Buffering Strategy....4
4.2.2 NT OS/2 Read/Write Buffering Strategy....4

5. NT OS/2 Mailslot I/O Operations....7
5.1 Create Mailslot....7
5.2 Create File....9
5.3 Open File....9
5.4 Read File....9
5.5 Write File....10
5.6 Read Terminal File....10
5.7 Query Directory Information....10
5.8 Notify Change Directory....10
5.9 Query File Information....10
5.9.1 Basic Information....11
5.9.2 Standard Information....11
5.9.3 Internal Information....11
5.9.4 Extended Attribute Information....11
5.9.5 Access Information....11
5.9.6 Name Information....11
5.9.7 Position Information....11
5.9.8 Mode Information....11
5.9.9 Alignment Information....11
5.9.10 All Information....12
5.9.11 Mailslot Information....12
5.10 Set File Information....12
5.10.1 Basic Information....12
5.10.2 Disposition Information....13
5.10.3 Link Information....13
5.10.4 Position Information....13
5.10.5 Mode Information....13
5.10.6 Mailslot Information....13
5.11 Query Extended Attributes....13
5.12 Set Extended Attributes....13
5.13 Lock Byte Range....13
5.14 Unlock Byte Range....14
5.15 Query Volume Information....14
5.16 Set Volume Information....14
5.17 File Control Operations....14
5.17.1 Peek....14
5.18 Flush Buffers....15
5.19 Set New File Size....15

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.20 Cancel I/O Operation 15
5.21 Device Control Operations 15
5.22 Close Handle 15

6. Win32 API Emulation...15
6.1 CreateMailslot 15
6.2 GetMailslotInfo...16
6.3 SetMailslotInfo 16

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification discusses the mailslot facilities of **NT OS/2**. Mailslots are a form of interprocess communication (IPC). They provide a facility for unidirectional message passing. The creator of a mailslot is the only process that can read from the mailslot. Other processes can only write messages to the mailslot.

The real value of mailslots is their usefulness in a network context. Mailslots can be used to send messages to either a single machine, or to all machines in a LAN Manager domain. More importantly, the network server needn't be running to support remote mailslots. The **NT OS/2** LAN Manager redirector can receive second class mailslot messages. The LAN Manager server must be running in order to receive first class mailslot messages. The second class mailslot allows simple peer-to-peer communication without the memory burden of running the server. Remote mailslots are described in greater detail in the **NT OS/2** LAN Manager Redirector Specification. Mailslot send classes are described later in this document.

In addition to describing the **NT OS/2** mailslot facilities, this specification also discusses the way in which the Win32 mailslot APIs are emulated.

2. Goals

The major goals for the mailslot capabilities of **NT OS/2** are the following:

1. Provide the basic primitives necessary to compatibly emulate the OS/2 LAN Manager mailslot capabilities.

2. Provide protection and security attributes for mailslots that are comparable to the capabilities provided for files and other **NT OS/2** objects.

3. Provide for LAN Manager server and client redirection of mailslots without having to enter the OS/2 subsystem.

4. Provide a fully qualified name space for mailslots that fits into the **NT OS/2** name structure in a straightforward manner.

3. Overview of OS/2 Mailslots

OS/2 mailslots provide a unidirectional IPC facility. Used locally (unnetworked) mailslots are analogous to a unidirectional-message mode-blocking named pipe.

Mailslot messages consist of a message buffer and a priority. The priority is an integer number in the range 0 to 9. Zero is the lowest priority. The OS/2 implementation treats mailslot messages as having only 2 priorities: zero and non-zero. Priority zero messages are written onto the end of the message buffer. Higher priority messages are copied to the front of the circular message buffer if their priority is higher than the message at the head of the buffer.

A mailslot is created by calling the **DosMakeMailslot** API. The creator of the mailslot receives a handle to the server side of the mailslot. Only the owner of a server side handle may read messages from the mailslot.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Under OS/2, mailslots are not part of the file system in any sense. A process cannot obtain a handle to a mailslot using **DosOpen**, nor can it use the handle obtained from **DosMakeMailslot** and pass it to a file system API such as **DosRead** or **DosWrite**.

The only way to read a mailslot is to use the **DosReadMailslot** or the **DosPeekMailslot** API. The **DosReadMailslot** API supplies a buffer for the data, but does not supply a buffer size. The buffer supplied must be at least as large as the buffer size defined by **DosMakeMailslot**. **DosReadMailslot** also returns size and priority information about the next message in the mailslot, but there is no guarantee that this message will be the next message that is read. The API takes a timeout parameter, which is the maximum time to wait for a message to become available to read if there are no messages waiting to be read when the call is issued.

DosPeekMailslot reads the next message from the mailslot but does not remove it from the mailslot buffer. It does not wait for a message to become available.

DosMailslotInfo returns configuration and status information about the mailslot and the current first message in the mailslot.

DosDeleteMailslot closes and deletes the mailslot and discards all unread messages.

The only action that can be performed on a mailslot by a process that does not have a server side handle is to write to the mailslot using the **DosWriteMailslot** API. This API, takes the name of the mailslot, rather than a handle, as a parameter. The mailslot name has one of the following forms:

- o *Mailslot**Name*. The target is a local mailslot.
- o *Server**Mailslot**Name*. The target is a remote mailslot.
- o *Domain**Mailslot**Name*. The target is the set of remote mailslots with this name in the domain *Domain*.
- o **Mailslot**Name*. The target is the set of remote mailslots with this name in the workstation's primary domain.

In addition to multiple mailslot name formats **DosWriteMailslot** also supports two classes of mailslot: first class and second class. A first class mailslot write guarantees delivery, while a second class mailslot write does not.

A second class write to a local mailslot is identical to a first class write. A first class write can be used to write a message to a remote machine running as a server. However, in order to write to a remote workstation only machine, or to do a broadcast write to a domain of machines, only a second class write can be used.

DosWriteMailslot allows specification of a timeout. This is the maximum time to wait for enough space to become available in the mailslot buffer to complete the write.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Overview of NT OS/2 Mailslots

4.1 Implementation Alternatives

Mailslots, as defined by OS/2 LAN Manager, are not a perfect fit for an NT file system. Special semantics exist for creating, reading and writing to mailslots.

There are two ways to implement **NT OS/2** mailslots:

1. Implement the mailslot capabilities as a separate object with its own complete set of APIs.

2. Implement the mailslot capabilities as a file system.

The first alternative is attractive because mailslot APIs can be designed to directly support OS/2 LAN Manager semantics. However, using this approach would complicate the security and networking implementations.

The second alternative has the advantage that it allows mailslots to use the built-in file system features of the NT I/O system. However, some allowance must be made to adapt to mailslot semantics.

There are several ways to implement mailslots as a file system.

1. Extend or bend the currently existing I/O system APIs, by adding new parameters, or by redefining old parameters that are not needed by the mailslot file system.

2. Use extended attributes as the means of defining the mailslot attributes required by OS/2.

3. Create a new API for creating a mailslot, and use a combination of existing APIs and the NtFsControlFile sub-APIs to implement the desired features.

The first alternative does not add any new APIs to the system but requires either adding special case code to the I/O system or adding parameters to **NtReadFile** and **NtWriteFile**. This is clearly unacceptable.

The second alternative also requires special case code for dealing with extended attributes on a mailslot.

The third alternative requires the addition of a new mailslot only API.

Mailslots will be implemented as a file system with a new API to create the mailslot and existing APIs for all other mailslot functions. This will be the easiest to implement and will yield an efficient implementation.

NT OS/2 will not support priorities or mailslot send classes. Priorities as implemented in OS/2 are essentially useless. The absence of a first class mailslot write may be noticed, but this is unlikely, as there are no known applications that use first class mailslots.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2 Read/Write Buffering Strategy

4.2.1 OS/2 Read/Write Buffering Strategy

OS/2 implements mailslots using a synchronous I/O model with a single circular buffer. The buffer must fit in a single 64KB segment. For each write/read operation there are always 2 data copies. Write data is copied from the user buffer to the mailslot buffer. Read data is copied from the mailslot buffer to the user buffer.

The read and write operations are controlled by four semaphores. Two signaling semaphores are used to signal waiting readers or writers that data is available to be read or that space is now available to write into the mailslot buffer. The other two semaphores are used to restrict reading or writing to a single thread at one time.

When a write begins the writer obtains the write lock. If there is no space available in the mailslot buffer the writer waits on the write semaphore for the timeout period specified by the caller. If there is space in the mailslot buffer, or space becomes available, the writer copies its data onto the end of the mailslot buffer.

If the write message is a high priority message (priority greater than zero) and its priority is greater than the priority of the message at the head of the mailslot message queue, then the writer must also obtain the read lock, copy the message ahead of the first message in the buffer, update the next-message-to-read pointer and release the read lock.

When the write has completed, it signals the read semaphore, indicating that write data is available and releases the write lock.

When a read operation begins the reader obtains the read lock. If there is no message available to read, the reader waits on the read semaphore for the time period specified by the caller. If a message is available to be read, the reader copies the message and updates the next message available pointer.

When the read is complete, the reader signals the write semaphore, indicating that there is space in the mailslot buffer, and releases the read lock.

4.2.2 NT OS/2 Read/Write Buffering Strategy

NT OS/2 supports an asynchronous I/O model and uses the concept of quotas to control the allocation of system buffers.

The mailslot buffer is not actually allocated to real memory in **NT OS/2**. Instead, the creator of a mailslot is simply charged memory quota for the buffer. Writers can use up to the quota charged to the creator without having any quota charged against themselves. If the quota charged to the creator is exhausted then the writer is charged for any additional memory that is required. Likewise, a reader is charged quota, equal to the size of the user's read buffer, if no data is available, the quota charged to the creator is exhausted, and the read request is queued rather than completed immediately.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following is a somewhat simplified discussion of the buffering scheme used for mailslots in **NT OS/2**. The exact behavior of the **NT OS/2** mailslot buffering depends on whether a read occurs before a write or vice versa.

When a write operation occurs the writer's output buffer is probed for read accessibility in the requesting mode. A system buffer is allocated that is the required size to hold the write data and memory quota is charged to the writer if and only if the quota charged to the creator of the mailslot instance has been exhausted (e.g., because of a previous read or write request). A buffer header is initialized at the front of the system buffer, the write data is copied into the system buffer, and the buffer header is inserted into the prioritized list of writers. The writer's I/O request is always completed immediately. The system buffer will be deallocated and the creator's quota returned when a matching read arrives.

At this point the write has been completed and control is returned to the caller. If another write request is received before the first write message is read, then the same operations are performed and the new request is placed at the appropriate place in the pending queue.

At some subsequent point in time, a read request arrives and it is determined that write data is available. The caller's input buffer is probed for write accessibility in the requesting mode. The read then proceeds to "pull" (copy) data directly from the system buffer that was previously allocated for the write data into the user's input buffer. At the completion of the copy, the read I/O request is completed.

Completion of the read request involves writing the I/O status block and setting the completion event. The system buffer for the original write request is deallocated and memory quota is returned for mailslot write buffering.

If an access violation occurs during a copy from the output buffer to a system buffer, then the write operation is immediately terminated. This has no effect on the integrity of the system. A malicious writer could easily accomplish the same effect by simply writing a shortened message. The write I/O status is set to access violation, the write I/O request is completed, and successful completion is returned as the service status.

If a read operation occurs before a write, then the reader's input buffer is probed for write accessibility in the requesting mode. A system buffer is allocated that is the required size to hold the input data and memory quota is charged to the reader if and only if the quota charged to the creator of the mailslot instance has been exhausted (e.g., because of a previous read request). A buffer header is initialized at the front of the system buffer and the header is inserted in a first-in-first-out list of readers. The read request type is converted to a buffered request so that upon completion, the I/O system will copy the received data from the system buffer into the reader's input buffer, deallocate the system buffer, and return memory quota as appropriate.

At this point, the I/O operation is pending and control is returned to the caller. If another read request is received before the first read is completed, then the same operations are performed and the new request is placed at the end of the pending queue.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

At some subsequent point in time, a write request arrives and it is determined that a read is pending. The caller's output buffer is probed for read accessibility in the requesting mode. The write then proceeds to "push" (copy) data directly from the output buffer into the system buffer that was previously allocated for the read operation. At the completion of the copy, the read and write I/O requests are both completed.

When a read operation is queued, a timer is started and set for the timeout specified by the read operation. A pointer to the timer object is saved in the header of the read buffer. If the read operation is dequeued for completion before the timer expires then the timer is cancelled, and the operation completes normally.

If the timeout DPC is called, the queue of pending reads is searched for the read corresponding to the expired timer. If it is not found the read is assumed to have completed and there is no need to take any action. If the read is found it is dequeued from the pending queue and completed with an error status.

If the buffer supplied by the reader is not large enough to read the entire mailslot message the read operation fails.

Completion of the write request involves writing the I/O status block and setting the completion event, whereas completion of the read request requires copying the read data from the system buffer to the reader's input buffer, deallocating the system buffer and returning the memory quota as appropriate, writing the I/O status block, and setting the completion event.

If an access violation occurs during a copy from a system buffer to the input buffer, then the read operation is immediately terminated. Previously completed write I/O requests are not backed out. This has no effect on the integrity of the system. A malicious reader could easily accomplish the same effect by simply reading and discarding information. The read I/O status is set to access violation and the read I/O request is completed with an error status.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. NT OS/2 Mailslot I/O Operations

The following subsections describe the **NT OS/2** I/O operations with respect to mailslots. Additional information can be found in the **NT OS/2** I/O System Specification.

5.1 Create Mailslot

A server end handle to a mailslot is obtained by calling the **NtCreateMailslotFile** function:

NTSTATUS
NtCreateMailslotFile (
    OUT PHANDLE *FileHandle,*
    IN ULONG *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    IN ULONG *CreateOptions,*
    IN ULONG *MailslotQuota,*
    IN ULONG *MaximumMessageSize,*
    IN PTIME *ReadTimeout*
    );

Parameters:

FileHandle - A pointer to a variable that receives the file handle value.

DesiredAccess - Specifies the type of access that the caller requires to the mailslot.

DesiredAccess Flags:

SYNCHRONIZE - The file handle may be waited on to synchronize with the completion of I/O operations.

READ_CONTROL - The ACL and ownership information associated with the mailslot may be read.

WRITE_DAC - The discretionary ACL associated with the mailslot may be written.

WRITE_OWNER - Ownership information associated with the mailslot may be written.

FILE_READ_DATA - Data may be read from the mailslot.

FILE_WRITE_DATA - Data may be written to the mailslot.

FILE_READ_ATTRIBUTES - Mailslot attributes flags may be read.

FILE_WRITE_ATTRIBUTES - Mailslot attribute flags may be written.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The three following values are the generic access types that the caller may request along with their mapping to specific access rights:

GENERIC_READ - Maps to *FILE_READ_DATA* and *FILE_READ_ATTRIBUTES*.

GENERIC_WRITE - Maps to *FILE_WRITE_DATA* and *FILE_WRITE_ATTRIBUTES*.

GENERIC_EXECUTE - Maps to *SYNCHRONIZE*.

ObjectAttributes - A pointer to a structure that specifies the object attributes; refer to the I/O System Specification for details.

IoStatusBlock - A pointer to a structure that receives the final completion status. The actual action taken by the system is written into the *Information* field of this structure.

CreateOptions - Specifies the options that should be used when creating the mailslot.

CreateOptions Flags:

FILE_SYNCHRONOUS_IO_ALERT - Indicates that all operations on the mailslot are to be performed synchronously. Any wait that is performed on behalf of the caller is subject to premature termination by alerts.

FILE_SYNCHRONOUS_IO_NONALERT - Indicates that all operations on the mailslot are to be performed synchronously. Any wait that is performed on behalf of the caller is not subject to premature termination by alerts.

MailslotQuota - Specifies the pool quota that is reserved for the mailslot. If set to *MAILSLOT_SIZE_AUTO* to file system will set the pool quota to zero. This means that all mailslot quota will come from readers or writers.

MaximumMessageSize - Specifies the maximum size message that can be written to the mailslot.

ReadTimeout - If specified, specifies the maximum amount of time that a read operation can block waiting for a mailslot message to become available. The default setting is *MAILSLOT_WAIT_FOREVER*.

This service creates a mailslot. The Access Control List (ACL) from the object attributes parameter defines the discretionary access control for the mailslot.

The create options, and share access are set to their specified values.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The actual pool quota that is reserved for the mailslot is either the system default, the system minimum, the system maximum, or the specified quota rounded up to the next allocation boundary.

The name of the mailslot is taken from the object attributes parameter, which must be specified.

The mailslot is deleted, along with any unread message, when the last reference to the creation handle is closed.

If *STATUS_SUCCESS* is returned as the service status, then the mailslot was successfully created.

If *STATUS_INVALID_PARAMETER* is returned as the service status, then an invalid value was specified for one or more of the input parameters.

5.2 Create File

The **NtCreateFile** function can be used to open a client end handle to an instance of a specified mailslot.

In order to use this function to open a mailslot, the mailslot must already exist and the *CreateDisposition* value must be specified as either *FILE_OPEN* or *FILE_OPEN_IF*.

ShareAccess should be set to *FILE_SHARE_WRITE* | *FILE_SHARE_READ*.

If a mailslot of the specified name cannot be found, then *STATUS_OBJECT_PATH_NOT_FOUND* is returned as the service status.

5.3 Open File

The **NtOpenFile** function can be used to open a client end handle to an instance of a specified mailslot.

ShareAccess should be set to *FILE_SHARE_WRITE* | *FILE_SHARE_READ*.

If a mailslot of specified name cannot be found, then *STATUS_OBJECT_PATH_NOT_FOUND* is returned as the service status.

5.4 Read File

The **NtReadFile** function can be used to read data from the server end of a mailslot. Priority information for the message is discarded. If the mailslot is empty, the operation will be queued. The operation will complete when either of the following is true:

1. A write operation occurs and data becomes available to be read.
2. The mailslot file handle is closed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The byte offset and key parameters of the **NtReadFile** function are ignored by the mailslot file system.

If *STATUS_PENDING* is returned as the service status, then the read I/O operation is pending and its completion must be synchronized using the standard **NT OS/2** mechanisms. Any other service status indicates that the read I/O operation has already been completed or will never complete. If a success status is returned, then the I/O status block contains the I/O completion information. Otherwise, the service status determines any error that may have occurred.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the read buffer became inaccessible after it was probed for write access.

If the I/O status *STATUS_BUFFER_OVERFLOW* is returned, then the read I/O operation was completed successfully, but the size of the input buffer was not large enough to hold the entire input message. A full buffer of data is returned; additional data can be read from the message using the **NtReadFile** function. The I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_SUCCESS* is returned, then the read I/O operation was completed successfully and the I/O status block contains the number of bytes that were read.

5.5 Write File

The **NtWriteFile** function can be used to write data to a mailslot.

The byte offset and key parameters of the **NtWriteFile** function are ignored by the mailslot file system.

If a success status is returned, then the I/O status block contains the I/O completion information. Otherwise, the service status determines any error that may have occurred.

If the size of the message buffer is larger then the maximum message size specified by the caller of **NtCreateMailslotFile**, then the operation will complete with the I/O status *STATUS_BUFFER_TOO_SMALL* and no data is written to the mailslot.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the write buffer became inaccessible after it was probed for read access.

If the I/O status *STATUS_SUCCESS* is returned, then the write I/O operation was completed successfully and the I/O status block contains the number of bytes that were written.

5.6 Read Terminal File

This function is not supported by the mailslot file system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.7 Query Directory Information

The **NtQueryDirectoryFile** function can be used to enumerate files within the root mailslot file system directory (i.e., "\Device\Mailslot\"). All the standard **NT OS/2** information classes are supported. **NtOpenFile** is used to open the root mailslot directory.

5.8 Notify Change Directory

The **NtNotifyChangeDirectoryFile** function can be used to monitor modifications to the root mailslot file system directory. The standard **NT OS/2** capabilities are supported.

5.9 Query File Information

Information about a mailslot can be obtained with the **NtQueryInformationFile** function. Most information classes, not including extended attribute information, are supported for mailslots with special interpretation of the returned data as appropriate. An additional information class is also provided to return information that is specific to mailslots.

Information is returned by the mailslot file system for mailslots and for the mailslot root directory. The following subsections describe the information that is returned for mailslot entries. The information returned for the root directory is identical to the information that is returned by other file systems and is described in the **NT OS/2** I/O System Specification.

5.9.1 Basic Information

Basic information about a mailslot includes the creation time, the time of the last access, the time of the last write, the time of the last change, and the attributes of the mailslot. The file attribute value for a mailslot is **FILE_ATTRIBUTE_NORMAL**.

5.9.2 Standard Information

Standard information about a mailslot includes the allocation size, the end of file offset, the device type, the number of hard links, whether a delete is pending, and the directory indicator.

The allocation size is the amount of pool quota charged to the creator. The end of file offset is the number of bytes that are available in the buffer. The device type is **FILE_DEVICE_MAILSLOT**, the number of hard links is one, delete pending is TRUE, and the directory indicator is FALSE.

5.9.3 Internal Information

Internal information about a mailslot includes a mailslot file-system-specific identifier.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.9.4 Extended Attribute Information

The extended attribute information size is always returned as zero by the mailslot file system.

5.9.5 Access Information

Access information about a mailslot includes the granted access flags.

5.9.6 Name Information

Name information about a mailslot includes the name of the mailslot.

5.9.7 Position Information

Position information about a mailslot includes the current byte offset. The current byte offset is the number of bytes that are available to be read in the mailslot buffer.

5.9.8 Mode Information

Mode information about a mailslot includes the I/O mode of the mailslot.

5.9.9 Alignment Information

The alignment information class is not supported by the mailslot file system.

5.9.10 All Information

The all information class includes information that can be returned by all file systems and is described above under each of the individual subsections.

5.9.11 Mailslot Information

Mailslot information on a mailslot includes: The quota charged for the mailslot buffer, the maximum message size. An access of FILE_READ_ATTRIBUTE is required to query the mailslot information of a mailslot.

FileMailslotQueryInformation - Data type is *FILE_MAILSLOT_QUERY_INFORMATION.*

```
typedef struct _FILE_MAILSLOT_QUERY_INFORMATION {
    ULONG MaximumMessageSize;
    ULONG MailslotQuota;
    ULONG NextMessageSize;
    ULONG MessagesAvailable;
    TIME ReadTimeout;
} FILE_MAILSLOT_QUERY_INFORMATION;
```

FILE_MAILSLOT_QUERY_INFORMATION:

MaximumMessageSize - The size, in bytes, of the largest message than can be written to the mailslot.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

MailslotQuota - The amount of pool quota that is reserved for the mailslot buffer.

NextMessageSize - The size of the next message avaible in the mailslot. If no message is available a value of *MAILSLOT_NO_MESSAGE* is returned.

MessagesAvailable - The number of messages currently available at the mailslot.

ReadTimeout - The current read timeout for the mailslot. See **NtCreateMailslotFile** for a full description.

5.10 Set File Information

Information about a mailslot can be changed with the **NtSetInformationFile** function. Most information classes are supported for mailslots with the exception of link and position information.

Information can be set for mailslots. The following subsections describe the information that can be set for mailslots.

5.10.1 Basic Information

Basic information about a mailslot that can be set includes the creation time, the time of the last access, the time of the last write, the time of the last change, and the attributes of the mailslot.

The associated times included in this class can be set to any appropriate value. The file attribute field can only be set to **FILE_ATTRIBUTE_NORMAL**.

5.10.2 Disposition Information

The disposition information class is not supported by the mailslot file system.

Mailslots are always considered temporary and are deleted when the creator of the mailslot closes all of its handles.

5.10.3 Link Information

This information class is not supported by the mailslot file system.

5.10.4 Position Information

This information class is not supported by the mailslot file system.

5.10.5 Mode Information

Mode information about a mailslot that can be set includes the I/O mode of the mailslot.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.10.6 Mailslot Information

Maislot information on a maillot that can be set includes: The read timeout.

FileMailslotSetInformation - Data type is *FILE_MAILSLOT_SET_INFORMATION*.

typedef struct **_FILE_MAILSLOT_SET_INFORMATION** {
    PTIME *ReadTimeout*;
} **FILE_MAILSLOT_SET_INFORMATION**;

FILE_MAILSLOT_SET_INFORMATION:

> *ReadTimeout* - The read timeout for the mailslot. See **NtCreateMailslotFile** for more information.

5.11 Query Extended Attributes

This function is not supported by the mailslot file system.

5.12 Set Extended Attributes

This function is not supported by the mailslot file system.

5.13 Lock Byte Range

This function is not supported by the mailslot file system.

5.14 Unlock Byte Range

This function is not supported by the mailslot file system.

5.15 Query Volume Information

This function is not supported by the mailslot file system.

5.16 Set Volume Information

This function is not supported by the mailslot file system.

5.17 File Control Operations

The following subsections describe file control operations that can be performed using a handle that is open to mailslot. The peek function can only be executed using a handle that is open to the server end of a mailslot.

5.17.1 Peek

The peek file control operation reads data from a mailslot but does not actually remove the data. This operation may performed only using a server side handle to the mailslot.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The control code for this operation is *FSCTL_MAILSLOT_PEEK*.

This operation returns two buffers. The "input" buffer contains the parameter buffer for the peek operation. This buffer must be large enought to contain the strcuture specified below. The "output" buffer specifies the the data buffer. The parameter buffer has the following format:

```
typedef struct _FILE_MAILSLOT_PEEK_BUFFER {
    ULONG ReadDataAvailable;
    ULONG NumberOfMessages;
    ULONG MessageLength;
} FILE_MAILSLOT_PEEK_BUFFER;
```

FILE_MAILSLOT_PEEK_BUFFER:

ReadDataAvailable - The number of bytes of read data that are available in the mailslot buffer.

NumberOfMessages - The number of messages that are currently in the mailslot.

MessageLength - The number of bytes that are contained in the first message in the mailslot.

This function is similar to the **NtReadFile** function for a mailslot; however, no data is actually removed from the mailslot and the operation is always completed immediately, i.e., it never causes an I/O operation to be queued.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the output buffer became inaccessible after it was probed for write access.

If the I/O status *STATUS_BUFFER_OVERFLOW* is returned, then the peek I/O operation was completed successfully, but the size of the output buffer was not large enough to hold the entire input message. A full buffer of data is returned; the actual message size can be determined from information placed in the output buffer. The I/O status block contains the number of bytes that were read including the mailslot information.

If the I/O status *STATUS_SUCCESS* is returned, then the peek I/O operation was completed successfully and the I/O status block contains the number of bytes that were read including the mailslot information.

5.18 Flush Buffers

This function is not supported by the mailslot file system.

5.19 Set New File Size

This function is not supported by the mailslot file system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.20 Cancel I/O Operation

The **NtCancelIoFile** function can be used to cancel all I/O operations that were issued by the subject thread for the specified mailslot. Both read and write operations initiated by the subject thread are canceled.

5.21 Device Control Operations

No device control operations are supported by the mailslot file system.

5.22 Close Handle

The **NtClose** function can be used to close a handle to the specified mailslot.

If the specified handle is the last handle that is open to the server side of the specified mailslot, then the state of the mailslot is set to closing. Read and write operations that are pending are completed with an I/O status of *STATUS_PIPE_CLOSED*.

6. Win32 API Emulation

The following subsections discuss the emulation of the Win32 mailslot facilities using the capabilities provided by **NT OS/2**. Only those Win32 functions which require special handling with respect to mailslots are included.

6.1 CreateMailslot

This Win32 API creates a mailslot and opens a server side handle to the newly created object.

This API can be emulated with the **NtCreateMailslotFile** service.

The Win32 inheritance bit of the security attributes is the same as the **NT OS/2** handle attributes field of the object attributes parameter.

The Win32 access bits of the open mode are the same as the **NT OS/2** desired access parameter.

The Win32 message size is the same as the **NT OS/2** maximum message size.

The Win32 mailslot size is the same as the **NT OS/2** mailslot quota parameter.

6.2 GetMailslotInfo

This Win32 API obtains configuration and status information about the mailslot.

This API can be emulated with the **NtQueryInformationFile** service, with the information class *FileMailslotQueryInformation*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.3 SetMailslotInfo

This Win32 API set configuration information about the mailslot.

This API can be emulated with the **NtSetInformationFile** service, with the information class *FileMailslotSetInformation*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft, December 28, 1990

Revision 1.1, January 5, 1991

1. Removed default read timeout and write send class.

2. Changed NtFsControlFile function FSCTL_MAILSLOT_WRITE to use separate input buffers for parameters and data.

3. Changed discussion of OS/2 API implementation to a discussion of Win32 API implementation.

4. Several editorial changes.

Revision 1.2, March 11, 1990

1. Added read timeout to NtCreateMailslotFile and made it queryable and settable.

2. Added new information class, FileMailslotSetInformation.

3. Removed message priorities, and the file system control function FSCTL_MAILSLOT_WRITE and FSCTL_MAILSLOT_READ.

4. Changed file system control function FSCTL_MAILSLOT_PEEK to use separate buffers to return parameters and data.

5. Update Win32 API discussion to conform with updated Win32 mailslot APIs.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS Memory Management Guide For I/O

Author: *Lou Perazzoli*

Original Draft 1.0, December 15, 1990
Revision 1.1, January 8, 1991
Revision 1.2, January 28, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction 1

2. Overview 1

3. Processes and Working Sets 2

4. Probe and lock pages 2

5. Mapping Locked Pages 3

6. Mapping I/O space 4

7. Physically Contiguous Memory 4

8. Non Cached Memory 4

9. Obtaining physical addresses 5

10. Paged and NonPaged Pool 5

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the memory management support routines available to I/O drivers, their usage and limitations.

2. Overview

The typical device driver has to deal with a number of memory management related issues - allocating buffers, working with MDL's, mapping device registers, etc. By properly designing the interaction between the driver and the memory management support routines, drivers will perform better in throughput, latency and system impact. Architectural differences between various architectures should be considered such that drivers are written to be as portable as possible.

Memory management supports a 4-gigabyte virtual address space. It is important to understand the differences between virtual addresses and physical addresses (and on MIPS how physical addresses appear in the virtual address space).

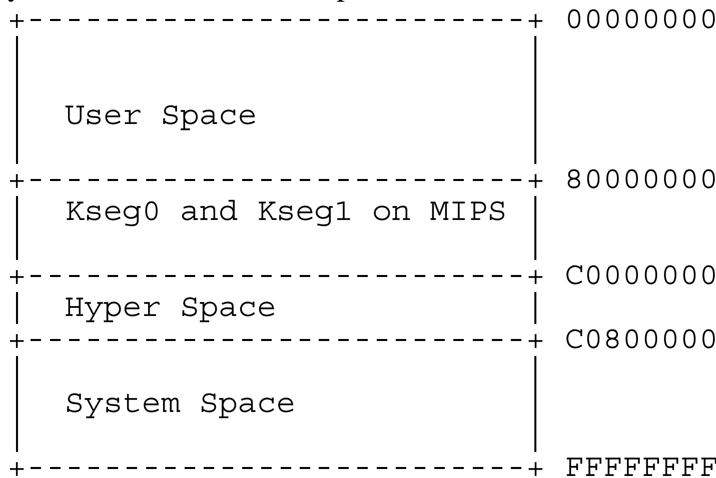
The 4 GB address space is divided into 3 regions:

- User space - Consists of 2 gigabytes which is unique for each address space. The page ownership for this region is user mode.
- Hyper space - Consists of 8 megabytes with a page ownership of kernel mode and is unique for each address space. Page table pages, working set lists, PTEs reserved for temporary mappings, and other address space unique structures reside in this region.
- System space - Consists of almost 2 gigabyte which is shared among all address spaces and has a page ownership of kernel mode.

The page ownership (user mode or kernel mode) is used for access checks for operations on virtual addresses.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Layout of Virtual Address Space:



System space contains a paged and a non-paged area. The paged area starts at the low addresses and grows upward, while the nonpaged area starts at the high addresses and grows downward.

3. Processes and Working Sets

Each process has a unique virtual address space which is independent from all other processes in both user space and hyper space. However, the system space portions have identical page translations and the non-paged portion of the system space can be referenced in any process at any IRQL. These items are very important for device drivers because when an interrupt occurs the processor could be executing in any thread's context.

As a thread executes and accesses non-valid virtual addresses (addresses that have no corresponding physical address) the pages are made valid (i.e., translated to a physical address), and are placed into the process's working set. Each process has a unique working set which consists of the set of all pageable addresses which are currently valid in the process. This includes both user space and system space addresses. This is important to note since a pageable system address that is resident in one process may not be valid in another process and a reference to that address may cause a page-fault to make the reference valid.

The working set has a minimum and maximum size. As the thread executes and faults more pages into the working set, the working set may exceed its minimum at which time the page fault routine determines if the working set is allowed to grow or if a page is to be removed from the working set. Hence, the working set acts as a process-specific quota.

4. Probe and lock pages

The **MmProbeAndLockPages** function takes as input an MDL which has the *StartVa, ByteOffset,* and *ByteCount* fields initialized. The function checks the specified range for access (read or write) and locks the physical memory corresponding the the virtual addresses and puts the "page frame numbers"

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

for the physical addresses into the MDL. In addition, the *Process* field of the MDL is initialized to the current process.

The **MmProbeAndLockPages** function keeps track of the number of pages in user-space each process has locked in memory and refuses to lock pages (it raises the exception STATUS_NO_MEMORY) if the total number of locked pages would exceed the working set minimum minus some small constant.

/Darryl should MmBuildMdlForNonPagedPool and MmMapLockedPages fill in the system VA field of the MDL if it is NULL?/

Another important aspect of **MmProbeAndLockPages** is that when the virtual address specified in the MDL is in the user's portion of the address space, when MmProbeAndLockPages returns the completed MDL, the user may change the address space. This means that the virtual address in the MDL may no longer correspond to the physical pages in the MDL. This causes no problem as long as the device never accesses the buffers through both the MDL (either physically or by mapping them in the system address space) and by the user's virtual address. This is a very important point.

To unlock the pages that were locked by **MmProbeAndLockPages** invoke the **MmUnlockPages** function specifying the same MDL that was used in the MmProbeAndLockPages call. This will cause the pages to be unlocked and the locked count to be decremented in the process.

If the buffer resides in the non-paged portion of system space and I/O completion is not invoked to unlock the buffer, the routine **MmBuildMdlForNonPagedPool** can be used to complete the MDL. This routine does not increment any reference counts or checks to ensure pages are resident, it merely updates the MDL with the corresponding page frame numbers.

5. Mapping Locked Pages

Certain devices, such as the standard AT disk, require the buffer to be accessed virtually rather than physically at high IRQL. But the pager, which is invoked when a virtual address does not have a corresponding physical address, can only be called at an IRQL of APC_LEVEL and below and at a mutex level below **MUTEX_LEVEL_WORKING_SET**.

To create virtual addresses that "map" the user's buffer and can be accessed at high IRQLs use the combination of **MmProbeAndLockPages** and **MmMapLockedPages**. The MmProbeAndLockPages function will complete the MDL and the MmMapLockedPages function will create a range of non-paged virtual addresses which map the physical buffer.

When invoking MmMapLockedPages the *AccessMode* argument should always be *KernelMode*. When the argument is specified as *UserMode* the buffer specified by the MDL is mapped into the user-mode portion of the current process and hence can only be referenced in the context of that process. This feature is used only by File System Processes (FSPs).

The **MmUnmapLockedPages** function deletes the mapping to the buffer. It is called with the *BaseAddress* that was returned by MmMapLockedPages and the same MDL that was passed into MmMapLockedPages.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The MmMapLockedPages and MmUnmapLockedPages have a cost which increases dramatically on a multi-processor system. When the pages are locked, the non-paged portion of system space is searched for an empty range to contain the buffer. The time for this search varies based on the number of pages in the MDL. Single page requests complete immediately, whereas multi-page requests may take slightly longer to locate a suitable range. When unmapping the pages, the addresses are marked as unused and returned and the **translation buffer is invalidated on all processors**. Note that if the request is for a single page then only a single address is invalidated if the underlying hardware supports single invalidation. Note that the 386 does not support single invalidation, but the 486 and the R4000 do.

The bottom line is that if the device supports DMA operations the driver should be designed such that **MmUnmapLockedPages** is never invoked.

6. Mapping I/O space

Most devices have control registers which reside in the I/O portion of the physical address space. In order to access these registers a corresponding non-pageable virtual address must be created which refers to the physical I/O address. This is accomplished using the **MmMapIoSpace** service. The returned address is the virtual address which corresponds to the specified physical I/O address. This virtual address is created either cached or non-cached depending on an argument. Note that only certain processors support non-cached memory via the translation hardware (MIPS and 486).

When the driver is being unloaded **MmMapIoSpace** is invoked to return the reserved address space back to the system.

7. Physically Contiguous Memory

On certain archaic systems, devices require the buffers to be physically contiguous in memory. On NT the memory management system has no support for memory compaction or other mechanisms to obtain physically contiguous pages. Hence a simple solution is required; when the system is initialized and non-paged pool created, all physical pages used for non-paged pool are contiguous.

Therefore during system initialization any allocation from non-paged pool is most likely physically contiguous. However, as the demand for nonpaged pool increases and non-paged pool is automatically expanded, the pages added are NOT physically contiguous. This means that after the system has been operating for a some period of time there is a possibility that no non-paged contiguous memory can be allocated.

Drivers that require physically contiguous memory should allocate their memory at driver initialization using the **MmAllocateContiguousMemory** function. The driver should then copy the user's buffer into or out of the physically contiguous area during its operation. Note depending on how the driver is designed, the copy may not require building a MDL for the user's buffer, rather it can use a simple RtlMoveMemory inside of a try/except block.

When the driver is being unloaded or the need for physically contiguous memory is no longer present, the memory should be deallocated with the **MmDeallocateContiguousMemory** function.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

8. Non Cached Memory

Certain devices require that buffers be shared between the device and the driver. These may be ring buffers which present a list of transfers to the device and a protocol is followed to insert and remove from the list, or other types of buffers. But the key thing about these buffers is that reads from and writes to the buffers must go directly to memory and not to the processor's cache where the device cannot see the changes.

The **MmAllocateNonCachedMemory** function allocates a range of nonpaged memory within system space and makes that memory non-cached. If insufficient memory is available, NULL is returned.

Currently, full pages are allocated to the request, so a request for 8 bytes allocates a full page. This is to avoid putting another pool type in the system and having the overhead of managing the pool as allocating noncached memory should be an infrequent function. It is anticipated that certain drivers will obtain some non cached pages at initialization and use only those pages for the life of the driver.

The **MmDeallocatedNonCachedMemory** function returns the non-cached memory back to the system pool.

9. Obtaining Physical Addresses

The **MmGetPhysicalAddress** function returns the physical address for a corresponding virtual address. This function should only be used to obtain the physical address of a virtual address that is in the non-pageable portion of the system.

10. Paged and NonPaged Pool

Paged and NonPaged pool requests have the following characterists:

- requests of PAGE_SIZE or less are always physically contiguous and do not cross a page boundary
- the returned address is aligned on a quadword boundary (low order 3 bits of virtual address are zero)
- pool allocations do not share cache lines with any other pool allocations.

11. Determining Non-Paged System Space Addresses

The **MmIsNonPagedSystemAddressValid** function allows the caller to determine if a given virtual address is within the non-paged portion of the system space and is currently mapped (valid). This includes such regions as non-paged pool, kernel stacks, and mapped locked pages.

12. Useful Macros

The following macros are also provided to aid in dealing with virtual addresses and buffer sizes:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- **ROUND_TO_PAGES**
  - Given a size, round the size up to the next page multiple.
- **BYTES_TO_PAGES**
  - Given a size, compute the number of pages required to contain a buffer of that size.
- **BYTE_OFFSET**

-Given a virtual address, return the byte offset for that virtual address.

- **PAGE_ALIGN**

-Given a virtual address, return the corresponding page aligned virtual address.

- **MM_IS_SYSTEM_VIRTUAL_ADDRESS**

-Given a virtual address, return *TRUE* if the virtual address is within system space.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, December 15, 1990.

Revision 1.1, January 8, 1991

1. Added MmIsAddressInNonPagedSystemSpace.

2. Editorial changes.

3. Added useful macros.

Revision 1.2, January 28, 1991

1. Added CacheEnable argument to MmMapIoSpace.

2. Changed MmIsAddressinNonPagedSystemSpace to MmIsNonPagedSystemAddressValid to more reflect the actions.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Mutant Specification

Author: *David N. Cutler*

Original Draft 1.0, October 19, 1989
Revision 1.1, November 12, 1989
Revision 1.2, November 28, 1989
Revision 1.3, January 5, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction.. 1
   1.1 Create Mutant Object... 1
   1.2 Open Mutant Object... 2
   1.3 Query Mutant Object.. 3
   1.4 Release Mutant Object 3

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the **Windows NT** *mutant object* which is used to emulate OS/2 2.0 Semaphore Mutexes. Although **Windows NT** provides other, more straightforward, capabilites to synchronize access to critical sections, this object has been included to enable more efficient emulation of the OS/2 2.0 capabilities.

Threads acquire ownership of a mutant object using the **Windows NT** wait services. Only one thread can own a mutant object at a time; however, the owner thread can recursively acquire the mutant object after first gaining ownership. If a thread terminates without releasing ownership of a mutant object, then the mutant object enters the *abandoned* state. The next thread that gains ownership of the mutant object will receive a return status that indicates that the mutant object was previously abandoned. Assigning ownership of an abandoned mutant object to another thread also clears the abandoned state of the mutant object.

Waiting for a mutant object causes the execution of the subject thread to be suspended until the thread can gain ownership of the mutant object. Satisfying the wait for a mutant object assigns ownership to the subject thread.

The following APIs are supported for the mutant object:

NtCreateMutant - Create mutant object and open handle
NtOpenMutant - Open handle to existing mutant object
NtQueryMutant - Get information about mutant object
NtReleaseMutant - Release ownership of a mutant object

1.1 Create Mutant Object

A mutant object can be created and a handle opened for access to the object with the **NtCreateMutant** function:

NTSTATUS
NtCreateMutant (
OUT PHANDLE *MutantHandle,*
IN ULONG *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL***,*
IN BOOLEAN *InitialOwner*
);

Parameters:

MutantHandle - A pointer to a variable that receives the mutant object handle value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

DesiredAccess - The desired types of access to the mutant object. The following object type specific access flags can be specified in addition to the *STANDARD_ACCESS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

MUTANT_QUERY_STATE - Query access to the mutant object is desired.

SYNCHRONIZE - Synchronization access (wait or release) to the mutant object is desired.

MUTANT_ALL_ACCESS - All possible types of access to the mutant object are desired.

ObjectAttributes - An optional pointer to a structure that specifies the object attributes; refer to Object Management Specification for details.

InitialOwner - A boolean value that determines whether the creator of the object desires immediate ownership of the mutant object.

If the *OBJ_OPENIF* flag is specified and a mutant object with the specified name already exists, then a handle to the existing object is opened and the *InitialOwner* parameter is ignored, provided the desired access can be granted. Otherwise, a new mutant object is created and a handle opened to the object with ownership as determined by the *InitialOwner* parameter. The status of the newly created mutant object is set to not abandoned.

1.2 Open Mutant Object

A handle can be opened to an existing mutant object with the **NtOpenMutant** function:

NTSTATUS
NtOpenMutant *(*
    OUT PHANDLE *MutantHandle,*
    IN ULONG *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
    );

Parameters:

MutantHandle - A pointer to a variable that receives the mutant object handle value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

DesiredAccess - The desired types of access to the mutant object. The following object type specific access flags can be specified in addition to the *STANDARD_ACCESS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

MUTANT_QUERY_STATE - Query access to the mutant object is desired.

SYNCHRONIZE - Synchronization access (wait or release) to the mutant object is desired.

MUTANT_ALL_ACCESS - All possible types of access to the mutant object are desired.

ObjectAttributes - A pointer to a structure that specifies the object attributes; refer to Object Management Specification for details.

If the desired types of access can be granted, then a handle is opened to the specified mutant object.

1.3 Query Mutant Object

The state of a mutant object can be queried with the **NtQueryMutant** function:

NTSTATUS
NtQueryMutant *(*
IN HANDLE *MutantHandle,*
IN MUTANTINFOCLASS *MutantInformationClass,*
OUT PVOID *MutantInformation,*
IN ULONG *MutantInformationLength,*
OUT PULONG *ReturnLength* **OPTIONAL**
);

Parameters:

MutantHandle - An open handle to a mutant object.

MutantInformationClass - The mutant information class for which information is to be returned.

MutantInformation - A pointer to a buffer that receives the specified information. The format and content of the buffer depend on the specified information class.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>MutantInformation Format by Information Class:</u>

MutantBasicInformation - Data type is *MUTANTBASICINFO*.

<u>MUTANTBASICINFO Structure</u>

LONG *CurrentCount* - The current ownership count of the mutant object.

BOOLEAN *AbandonedState* - The current abandoned state of the mutant object.

MutantInformationLength - Specifies the length in bytes of the mutant information buffer.

ReturnLength - An optional pointer to a variable that receives the number of bytes placed in the mutant information buffer.

This function provides the capability to determine the ownership and abandoned state of a mutant object.

1.4 Release Mutant Object

Ownership of a mutant object can be released with the **NtReleaseMutant** function:

NTSTATUS
NtReleaseMutant *(*
    IN HANDLE *MutantHandle,*
    OUT PLONG *PreviousCount* **OPTIONAL**
    );

<u>Parameters:</u>

MutantHandle - An open handle to a mutant object.

PreviousCount - An optional pointer to a variable that receives the previous ownership count of the mutant object.

A mutant object can only be released by a thread that currently owns the mutant object. When the mutant is released, the current count of the mutant object is incremented by one. If the resultant count is one, then the mutant object is no longer owned. Any threads that are waiting for the mutant object are examined to see if their wait can be satisfied.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, October 18, 1989

Revision 1.1, November 12, 1989

1. Added initial ownership parameter to NtCreateMutant.

Revision 1.2, November 28, 1989

1. Change access right required for wait access to be only *SYNCHRONIZE* access.

Revision 1.3, January 5, 1990

1. Change type name of object attributes parameter and refer to the Object Management Specification for the definition of this parameter.

2. Change the description of the desired access flags to include standard rights, object specific rights, and generic rights.

3. Delete the handle flags and object names parameters from the **NtOpenMutant** service and replace with a pointer to an object attributes structure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Named Pipe Specification

Author: *David N. Cutler & Gary D. Kimura*

Original Draft February 16, 1990
Revision 1.1, March 8, 1990
Revision 1.2, August 14, 1990
Revision 1.3, September 27, 1990
Revision 1.4, October 17, 1990
Revision 1.5, January 23, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction 1
2. Goals 1
3. Overview of OS/2 Named Pipes 1
4. Overview of NT OS/2 Named Pipes 3
    4.1 Implementation Alternatives 3
    4.2 Named Pipe Directories 4
    4.3 Read/Write Buffering Strategy 5
        4.3.1 OS/2 Read/Write Buffering Strategy 5
        4.3.2 NT OS/2 Read/Write Buffering Strategy 8
    4.4 Internal Read/Write Operations 13
        4.4.1 Special Read/Write Buffering 13
    4.5 Named Pipe States 13
5. NT OS/2 Named Pipe I/O Operations 16
    5.1 Create Named Pipe 16
    5.2 Create File 20
    5.3 Open File 20
    5.4 Read File 21
    5.5 Write File 22
    5.6 Read Terminal File 22
    5.7 Query Directory Information 22
    5.8 Notify Change Directory 22
    5.9 Query File Information 23
        5.9.1 Basic Information 23
        5.9.2 Standard Information 23
        5.9.3 Internal Information 23
        5.9.4 Extended Attribute Information 23
        5.9.5 Access Information 23
        5.9.6 Name Information 23
        5.9.7 Position Information 24
        5.9.8 Mode Information 24
        5.9.9 Alignment Information 24
        5.9.10 All Information 24
        5.9.11 Pipe Information 24
        5.9.12 Local Pipe Information 24
        5.9.13 Remote Pipe Information 26
    5.10 Set File Information 26
        5.10.1 Basic Information 26
        5.10.2 Disposition Information 26
        5.10.3 Link Information 27
        5.10.4 Position Information 27
        5.10.5 Mode Information 27
        5.10.6 Pipe Information 27
        5.10.7 Remote Pipe Information 27
    5.11 Query Extended Attributes 28

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.12 Set Extended Attributes 28
5.13 Lock Byte Range 28
5.14 Unlock Byte Range 28
5.15 Query Volume Information 28
5.16 Set Volume Information 28
5.17 File Control Operations 28
5.17.1 External File Control Operations 28
5.17.1.1 Assign Event 29
5.17.1.2 Disconnect 29
5.17.1.3 Listen 30
5.17.1.4 Peek 31
5.17.1.5 Query Event Information 32
5.17.1.6 Transceive 33
5.17.1.7 Wait For Named Pipe 34
5.17.1.8 Impersonate 35
5.17.2 Internal File Control Operations 36
5.17.2.1 Internal Read 36
5.17.2.2 Internal Write 36
5.17.2.3 Internal Transceive 36
5.18 Flush Buffers 36
5.19 Set New File Size 36
5.20 Cancel I/O Operation 37
5.21 Device Control Operations 37
5.22 Close Handle 37
6. OS/2 API Emulation 37
6.1 DosCallNmPipe 37
6.2 DosConnectNmPipe 37
6.3 DosDisconnectNmPipe 38
6.4 DosMakeNmPipe 38
6.5 DosPeekNmPipe 38
6.6 DosQNmPHandState 39
6.7 DosQNmPipeInfo 39
6.8 DosQNmPipeSemState 39
6.9 DosRawReadNmPipe 39
6.10 DosRawWriteNmPipe 39
6.11 DosSetNmPHandState 40
6.12 DosSetNmPipeSem 40
6.13 DosTransactNmPipe 40
6.14 DosWaitNmPipe 40

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification discusses the named pipe facilities of **NT OS/2**. Named pipes provide a full duplex interprocess communication (IPC) mechanism that can be used locally or across a network to access application servers. Named pipes provide the transport medium that is used for the Microsoft remote procedure call (RPC) capabilities.

Named pipes are used extensively by the OS/2 and LAN Manager components of the **NT OS/2** system, and therefore, must be implemented as efficiently as possible.

There are two manifestations of named pipes, those that are local to a system and those that are remote. This specification addresses both types of named pipes.

In addition to describing the **NT OS/2** named pipe facilities, this specification also discusses the way in which the OS/2 named pipe APIs are emulated.

2. Goals

The major goals for the named pipe capabilities of **NT OS/2** are the following:

1. Provide the basic primitives necessary to compatibly emulate the OS/2 named pipe capabilities.

2. Provide protection and security attributes for named pipes that are comparable to the capabilities provided for files and other **NT OS/2** objects.

3. Provide for LAN Manager server and client redirection of named pipes without having to enter the OS/2 subsystem.

4. Provide a fully qualified name space for named pipes that fits into the **NT OS/2** name structure in a straightforward manner.

5. Provide a high performance design and implementation of named pipes.

Although it is a major temptation, it is not a goal to "fix" the semantics of OS/2 named pipes. Minor discrepancies, however, will exist between OS/2 and **NT OS/2** named pipes where OS/2 capabilities or semantics are incompatible with those of **NT OS/2**, e.g., the named pipe naming and the asynchronous I/O model.

3. Overview of OS/2 Named Pipes

A named pipe provides a full duplex channel that can be used to implement an interprocess communication (IPC) mechanism between two processes. OS/2 uses named pipes to implement location-independent remote procedure call (RPC) capabilities and for communicating with servers on a remote system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Named pipes have two ends: 1) a client end, and 2) a server end. Both ends are full duplex—data written from one end can be read from the other end and vice versa.

The server end of a named pipe is created when a new instance of a named pipe is created, or when a previously created instance is reused. A new instance of a named pipe is created with the **DosMakeNmPipe** API in OS/2.

Before either the client or the server ends of a named pipe can be used, the server end must be connected. In OS/2 this is accomplished with the **DosConnectNmPipe** API.

Once an instance of a named pipe is created and the server end is connected, then the client end of the named pipe can be created using the OS/2 **DosOpen** API.

When both the server end of a named pipe is connected and the client end is opened, information can flow over the pipe using the OS/2 **DosRead** and **DosWrite** APIs.

Named pipes are created with five attributes:

1. A pipe type which is either message or byte stream.

2. A count that limits the maximum number of simultaneous instances of the named pipe that can be created.

3. An input buffer size that specifies the size of the buffer that is used for inbound data on the server side of the named pipe.

4. An output buffer size that specifies the size of the buffer that is used for outbound data from the server side of the named pipe.

5. A default timeout value that is to be used if a timeout value is not specified when the **DosWaitNmPipe** API is executed.

The type of a named pipe determines how information is written into the named pipe. If the named pipe is a message pipe, then information is written into the pipe in the form of messages which include the byte count and the data of the message. If the named pipe is a byte stream pipe, then only the data is written into the named pipe.

The maximum instance count is established when the first instance of a specific named pipe is created (i.e., one of a given name) and cannot later be modified. Thereafter, up to the maximum instance count of simultaneous instances of the named pipe can be created to provide an IPC mechanism between any pair of processes.

The input and output buffer sizes are considered hints to the system for the sizes of the buffers that are needed to buffer inbound and outbound data. The actual buffer sizes may be either the system default or the specified buffer sizes rounded up to the next allocation boundary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The default timeout value specifies a default for the amount of time that a client can wait for an available instance of a named pipe.

Once the first instance of a named pipe is created subsequent instances of an identically named pipe are subject to the maximum instances parameter. In addition, the type of pipe and the default timeout value are ignored and cannot be set when subsequent instances of the named pipe are created.

In addition to the five attribute parameters, two mode parameters can be specified when an instance of a named pipe is created or opened:

1. The read mode, which can be either message mode or byte stream mode, but which must be compatible with the type of the named pipe.

2. The blocking mode, which can be either blocking or nonblocking.

The read mode of a named pipe determines how data will be read from the pipe. If the named pipe is a message pipe, then data can be read in either message mode or byte stream mode. However, if the named pipe is a byte stream pipe, then data can only be read in byte stream mode.

The blocking mode determines what happens when a request cannot be satisfied immediately. If the mode is blocking, then an implied wait occurs until an operation is completed. Otherwise, the operation returns immediately with an error status.

Standard open parameters can also be specified when an instance of a named pipe is created or opened which define the access that is desired to the named pipe (e.g., read only, write only, or read/write access), whether the named pipe handle is inherited when a child process is created, and whether write behind is allowed on writes to the named pipe.

The open access parameters also specify the configuration of the named pipe when the first instance of a named pipe is created. A named pipe can have a full duplex or a simplex configuration. A full duplex named pipe allows data to flow in both directions, whereas a simplex named pipe only allows data to flow in one direction. The direction of data flow and configuration are determined by the read only (outbound), write only (inbound), and read/write (full duplex) open access parameters specified by the server when the first instance of a named pipe is created.

The server end of a named pipe can be reused by disconnecting the client end. In OS/2, this is accomplished using the **DosDisconnectNmPipe** API. The server end of a named pipe can also be disconnected by closing the respective file handle, but this deletes the instance of the named pipe and it cannot be reused.

The client end of a named pipe is disconnected by simply closing the respective file handle.

OS/2 supplies 14 APIs that are specific to named pipes. These APIs are intended mainly for use by a server. In addition, eleven standard OS/2 I/O system APIs can be executed using a file handle to a named pipe.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Overview of NT OS/2 Named Pipes

4.1 Implementation Alternatives

Named pipes must be integrated into the **NT OS/2** I/O system such that standard read and write requests can be used to read data from and write data to a named pipe. It also must be possible to accomplish LAN Manager server and client redirection of named pipes without having to call the OS/2 subsystem.

There are several ways of integrating named pipes into **NT OS/2** that meet these requirements:

1. Implement the named pipe capabilities as an installable file system and extend **NtCreateFile** so that the named pipe attributes required by OS/2 can be specified directly in the **NT OS/2** system service call.

2. Implement the named pipe capabilities as an installable file system and use extended attributes as the means of defining the named pipe attributes required by OS/2.

3. Implement named pipes as a separate object that is created with its own API, but which can be opened via a pipe driver.

4. Implement the named pipe capabilities as an installable file system and add an **NT OS/2** I/O system API that specifically creates an instance of a named pipe.

The first alternative requires an already complicated API to be further extended to accommodate yet another special case.

The second alternative overloads the use of extended attributes to have a special meaning for named pipes. Extended attributes are not the most efficient or convenient way of specifying the attribute values and would require special rules about when they could be read and written.

The third alternative would create a nonstandard object whose API was partly buried in the I/O system and partly in object-specific APIs.

The fourth alternative adds an additional API to the **NT OS/2** I/O system that has special meaning and is only applicable to named pipes.

The fourth alternative has been chosen as the means of implementing the named pipe capabilities in **NT OS/2**. Although this provides an additional I/O system API that is specific to named pipes, it is the most straightforward and efficient implementation.

4.2 Named Pipe Directories

In OS/2, named pipes have a rigid name syntax with the following form:

\PIPE\pipe-name

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This syntax is recognized by the OS/2 **DosOpen** API and is routed to the appropriate system component. The LAN Manager redirector is also capable of recognizing names of the following form:

\\server-name\PIPE\pipe-name

The redirector transforms the request into a tree connection to a server and then performs the appropriate SMB generation.

The **NT OS/2** named pipe driver will also implement a flat name space. The syntax for an **NT OS/2** named pipe is of the following form:

\Device\NamedPipe\pipe-name[1]

*\The object name space in **NT OS/2** is more general and hierarchical, and we would like named pipes to follow that scheme; however, because of issues involving persistent named pipes, and guaranteeing proper behavior given reparse the first named pipe driver will use a flat name space. Once the issues are resolved named pipes can be extended to existing file systems as a special file using reparse or by maintaining a named pipe database in a system file.*

The syntax for a remote **NT OS/2** named pipe is of the following form:

\Device\LanmanRedirector\server-name\Pipe\pipe-name

4.3 Read/Write Buffering Strategy

4.3.1 OS/2 Read/Write Buffering Strategy

The OS/2 named pipe capabilities use a two circular buffers for buffering inbound and outbound writes to a named pipe. This design is dictated by the synchronous I/O model of OS/2 and it controls the amount of system buffering space that is consumed. The data is copied twice for each write and read of a named pipe. One copy occurs when the data is written from a user buffer into a named pipe and another copy occurs when the data is read out of the named pipe into a user buffer.

An OS/2 named pipe can be either a message or byte stream pipe, which determines how write data is stored in the pipe buffers. Message pipes can be read in either message mode or byte stream mode. Byte stream pipes can only be read in byte stream mode. In addition, a blocking mode can be specified for each open of an instance of a named pipe. The blocking mode determines whether reads from, and writes to the named pipe block if sufficient data or space is not available in the named pipe.

---

[1]The string "\Device\NamedPipe" refers to the named pipe driver, while the string "\Device\NamedPipe\" represents the root directory of the named pipe file system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

A message named pipe stores the size of a message and the data for the message. A byte stream named pipe simply stores the data and no additional information. Reads from a message named pipe attempt to read a complete message from the pipe in either message mode or byte stream mode. If the complete message does not fit in the supplied read buffer, then a full buffer is returned along with an error status that signifies that there is more data in the message. Reads from a byte stream named pipe can only be made in byte stream mode and return the data that is currently in the pipe up to the size of the supplied buffer.

Each inbound and outbound buffer for a named pipe has a read lock, a write lock, a read semaphore, and a write semaphore. These are used to synchronize the reading and writing of data to and from the buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

When a write to a named pipe buffer begins, the write lock is acquired to prevent any other writer from writing into the buffer until the current write is finished. If the write must block because of a lack of available space in the buffer, then the reader semaphore is signaled, the writer lock is not released, and the writer waits for a reader to signal the write semaphore. The write lock is released at the completion of the write operation.

The following describes the OS/2 named pipe write logic.

```
if (message pipe) then
      if (blocking mode) then
            write message to pipe, synchronize with reader
            return size of message written
      else
            if (space for message header plus one byte) and
               (data buffer size greater than pipe buffer size) then
                  write data to pipe, synchronize with reader
                  return size of message written
            else
                  if (space for data buffer and message header) then
                        write data to pipe
                        return size of message written
                  else
                        return buffer overflow error
                  endif
            endif
      endif
else
      if (blocking mode) then
            write data to pipe, synchronize with reader
            return count of bytes written
      else
            if (space available in pipe buffer) then
                  write data to pipe (minimum data buffer/pipe space)
                  return count of bytes written
            else
                  return buffer overflow error
            endif
      endif
endif
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

When a read from a named pipe buffer begins, the read lock is acquired to prevent any other reader from reading from the buffer until the current read is finished. If the read must block because of a lack of available data in the buffer, then the writer semaphore is signaled, the reader lock is not released, and the reader waits for the read semaphore to be signaled. The read lock is released at the completion of the read operation.

The following describes the OS/2 named pipe read logic.

```
while (data not available in pipe) do
      if (blocking mode) then
            wait for available data in pipe
      else
            return no data available error
      endif
endwhile

if (message pipe) then
      if (data buffer size greater or equal message size) then
            if (message mode) then
                  read message from pipe, synchronize with writer
                  return size of message read
            else
                  read available data or message from pipe
                  if (complete message read) then
                        return size of message read
                  else
                        reduce size of message by available data bytes
                        return count of data bytes read
                  endif
            endif
      else
            if (message mode) then
                  read data from pipe, synchronize with writer
                  reduce size of message by data buffer size
                  return more data error
            else
                  read available data from pipe
                  reduce size of message by available data bytes
                  return count of data bytes read
            endif
      endif
else
      read available data from pipe
      return count of data bytes read
endif
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.3.2 NT OS/2 Read/Write Buffering Strategy

NT OS/2 supports an asynchronous I/O model and uses the concept of quotas to control the allocation of system buffers. In addition, **NT OS/2** supports I/O transfers that are buffered by the system rather than requiring buffers to be locked down and nonswappable. Therefore, the buffering scheme used for the **NT OS/2** implementation of named pipes differs markedly from that of OS/2.

The blocking mode of OS/2 is emulated in **NT OS/2** with a completion mode. The completion mode can be specified such that read and write operations are completed immediately or they are queued and subject to completion when space is available or data is present.

The inbound and outbound buffers for a named pipe are not actually allocated to real memory in **NT OS/2**. Instead, the creator of an instance of a named pipe is simply charged memory quota for these buffers. Writers and readers can use up to the quota charged to the creator without having any quota charged against themselves. If the quota charged to the creator is exhausted and a write request is queued rather than completed immediately, then the writer is charged for any additional quota that is required. Likewise, a reader is charged quota if no data is available, the quota charged to the creator is exhausted, and the read request is queued rather than completed immediately.

The named pipe capabilities of **NT OS/2** requires that the data be copied twice. However, reads can "pull" data from write buffers and writes can "push" data into read buffers.

The following is a somewhat simplified discussion of the buffering scheme used for named pipes in **NT OS/2**. Boundary conditions and differing pipe types and modes are not considered. The pipe type is assumed to be message, the read mode is assumed to be message, and the completion mode is assumed to be queued operation.

The exact behavior of the **NT OS/2** named pipe buffering depends on whether a read occurs before a write or vice versa.

If a write operation occurs before a read operation, then the writer's output buffer is probed for read accessibility in the requesting mode. A system buffer is allocated that is the required size to hold the write data and memory quota is charged to the writer if and only if the quota charged to the creator of the named pipe instance has been exhausted (e.g., because of a previous read or write request). A buffer header is initialized at the front of the system buffer, the write data is copied into the system buffer, and the buffer header is inserted into the first-in-first-out list of writers. If quota was not charged to the writer, then the writer's I/O request can be completed immediately. The system buffer will be deallocated and the creator's quota returned when a matching read arrives. Otherwise, the write request type is converted to a buffered request so that upon completion, the I/O system will deallocate the system buffer and return memory quota as appropriate.

At this point, the I/O operation is either pending or has been completed and control is returned to the caller. If another write request is received before the first set of write data is read, then the same operations are performed and the new request is placed at the end of the pending queue.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

At some subsequent point in time, a read request arrives at the read end of the pipe and it is determined that write data is available at the write end of the pipe. The input buffer is probed for write accessibility in the requesting mode. The read then proceeds to "pull" (copy) data directly from the system buffer that was previously allocated for the write data into the user's input buffer. At the completion of the copy, the read I/O request is completed.

Completion of the read request involves writing the I/O status block and setting the completion event. If the original write I/O request was completed at the time of the write, then the system buffer is deallocated and memory quota is returned for named pipe write buffering. However, if the write I/O was not completed at the time of the write, then completion of the write requires writing the I/O status block, setting the completion event, deallocating the system buffer, and returning quota to the writer.

If an access violation occurs during a copy from the output buffer to a system buffer, then the write operation is immediately terminated. Previously completed read I/O requests, if any, are not backed out. This has no effect on the integrity of the system. A malicious writer could easily accomplish the same effect by simply writing a shortened message. The write I/O status is set to access violation, the write I/O request is completed, and successful completion is returned as the service status.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following describes the **NT OS/2** named pipe write logic.

```
probe output buffer for read access
while (read pending) and (output buffer size not zero) do
      copy data from output buffer to read buffer
      if (read buffer greater or equal output buffer) then
            reduce output buffer size to zero
            set read I/O status to successful completion
      else
            reduce output buffer by read buffer size
            if (message mode read) then
                  set read I/O status to buffer overflow
            else
                  set read I/O status to successful completion
            endif
      endif
      remove read request from read pending list
      complete read I/O operation, return quota
endwhile
if (output buffer size not zero) then
      if (pipe quota available) then
            allocate write buffer, charge quota to pipe
            copy data from output buffer to write buffer
            insert write request in write pending list
            set write I/O status to successful completion
            complete write I/O operation
            return successful completion
      else
            if (queued operation) or ((message pipe) and
                  (output buffer not original size)) then
                  allocate write buffer, charge quota to writer
                  copy data from output buffer to write buffer
                  insert write request in write pending list
                  return operation pending
            else
                  if (output buffer not original size) then
                        set write I/O status to successful completion
                        complete write I/O request
                        return successful completion
                  else
                        abort write I/O operation
                        return no space available
                  endif
            endif
      endif
else
      set write I/O status to successful completion
      complete write I/O request
      return successful completion
endif
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If a read operation occurs before a write, then the reader's input buffer is probed for write accessibility in the requesting mode. A system buffer is allocated that is the required size to hold the input data and memory quota is charged to the reader if and only if the quota charged to the creator of the named pipe instance has been exhausted (e.g., because of a previous read or write request). A buffer header is initialized at the front of the system buffer and the header is inserted in a first-in-first-out list of readers. The read request type is converted to a buffered request so that upon completion, the I/O system will copy the received data from the system buffer into the reader's input buffer, deallocate the system buffer, and return memory quota as appropriate.

At this point, the I/O operation is pending and control is returned to the caller. If another read request is received before the first read is completed, then the same operations are performed and the new request is placed at the end of the pending queue.

At some subsequent point in time, a write request arrives at the write end of the pipe and it is determined that a read is pending at the read end of the pipe. The output buffer is probed for read accessibility in the requesting mode. The write then proceeds to "push" (copy) data directly from the output buffer into the system buffer that was previously allocated for the read operation. At the completion of the copy, the read and write I/O requests are both completed.

Completion of the write request involves writing the I/O status block and setting the completion event, whereas completion of the read request requires copying the read data from the system buffer to the reader's input buffer, deallocating the system buffer and returning the memory quota as appropriate, writing the I/O status block, and setting the completion event.

If an access violation occurs during a copy from a system buffer to the input buffer, then the read operation is immediately terminated. Previously completed write I/O requests are not backed out. This has no effect on the integrity of the system. A malicious reader could easily accomplish the same effect by simply reading and discarding information. The read I/O status is set to access violation, the read I/O request is completed, and successful completion is returned as the service status.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The following describes the **NT OS/2** named pipe read logic.

```
probe input buffer for write access
if (write not pending) then
      if (queued operation) then
            if (pipe quota available) then
                  allocate read buffer,charge quota to pipe
            else
                  allocate read buffer, charge quota to reader
            endif
            insert read request in read pending list
            return operation pending
      else
            abort read I/O operation
            return no data available
      endif
else
      set read I/O status to successful completion
      while (write pending) and (input buffer size not zero) do
            copy data from write buffer to input buffer
            if (input buffer greater or equal write buffer) then
                  if (message mode read) then
                        reduce input buffer size to zero
                  else
                        reduce input buffer by write buffer size
                  endif
                  set write I/O status to successful completion
                  remove write request from write pending list
                  complete write I/O operation, return quota
            else
                  reduce write buffer by input buffer size
                  reduce input buffer size to zero
                  if (message mode read) then
                        set read I/O status to buffer overflow
                  endif
            endif
      endwhile

      complete read I/O operation
      return successful completion
endif
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.4 Internal Read/Write Operations

4.4.1 Special Read/Write Buffering

In addition to the above buffering method provided for local named pipe clients and servers, **NT OS/2** provides another buffering method that can be used internally by the **NT OS/2** LAN Manager server. This method allows read and write requests to proceed such that no buffer allocation is needed for either the read or the write.

The **NT OS/2** LAN Manager server supplies the necessary system buffers directly and only one copy of the data is needed for a read or write operation. Typically, these buffers are the buffers that are used to receive and transmit data over the network. Thus, server side redirection can be performed with minimal overhead.

4.5 Named Pipe States

Named pipes can be in one of four states:

1. Disconnected

2. Listening

3. Connected

4. Closing

The initial state of a named pipe is *disconnected*. When the pipe is in this state, no client is connected to the pipe and a listen operation can be performed.

Performing a listen operation on a disconnected named pipe causes the pipe to transition to the *listening* state.

An open request performed by a client causes a named pipe in the listening state to enter the *connected* state. When a named pipe is in the connected state, data can flow through the pipe.

A named pipe that is in the connected state can transition to either the *disconnected* state or the *closing* state.

The disconnected state is entered when a disconnect operation is performed on the server end of a named pipe and causes both the input buffer and the output buffer to be flushed. No further access is allowed to the client end of the named pipe; however, the client end must still be closed.

The closing state is entered if a close operation is performed on either end of a named pipe and causes the input buffer of the closing end to be flushed. Any remaining data in the output buffer can be read from the opposite end of the named pipe with a read operation. When no data remains in the output buffer, an end of file indication is returned.

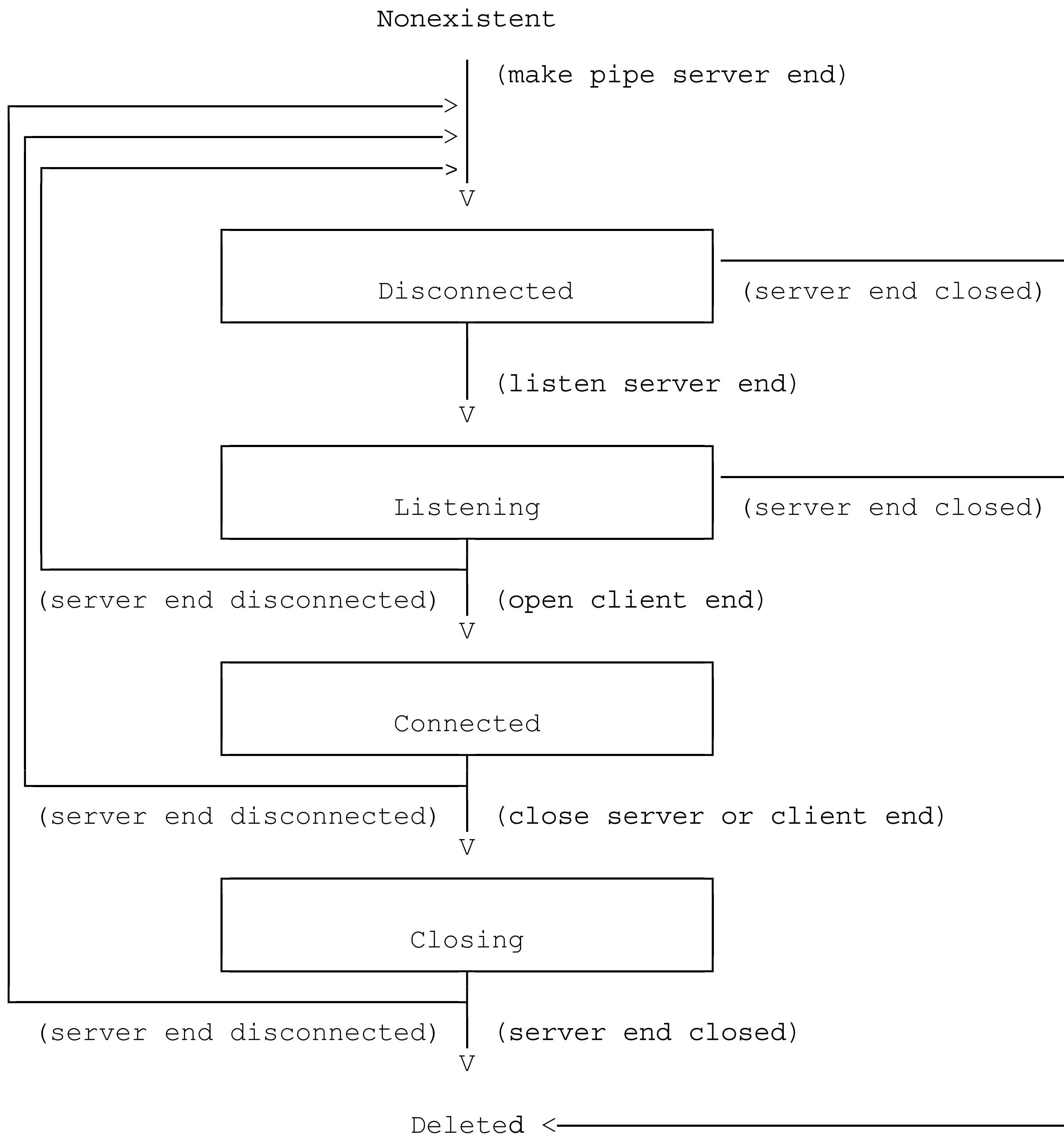
Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

A named pipe that is in the closing state because the client end of the pipe was closed can transition to the disconnected state by performing a disconnect operation on the server end of the pipe.

A named pipe that is in the closing state because the server end of the pipe was closed is deleted when the client end of the pipe is also closed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Named Pipe State Transition Diagram



Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. NT OS/2 Named Pipe I/O Operations

The following subsections describe the **NT OS/2** I/O operations with respect to named pipes. Additional information can be found in the **NT OS/2** I/O System Specification.

5.1 Create Named Pipe

The first instance of a specific named pipe or another instance of an existing named pipe can be created, and a server end handle opened with the **NtCreateNamedPipeFile** function. This function is only for creating local named pipes and not remote ones.

NTSTATUS
NtCreateNamedPipeFile (
 OUT PHANDLE *FileHandle,*
 IN ULONG *DesiredAccess,*
 IN POBJECT_ATTRIBUTES *ObjectAttributes,*
 OUT PIO_STATUS_BLOCK *IoStatusBlock,*
 IN ULONG *ShareAccess,*
 IN ULONG *CreateDisposition,*
 IN ULONG *CreateOptions,*
 IN ULONG *NamedPipeType,*
 IN ULONG *ReadMode,*
 IN ULONG *CompletionMode,*
 IN ULONG *MaximumInstances,*
 IN ULONG *InboundQuota,*
 IN ULONG *OutboundQuota,*
 IN PTIME *DefaultTimeout* **OPTIONAL**
 );

Parameters:

FileHandle - A pointer to a variable that receives the file handle value.

DesiredAccess - Specifies the type of access that the caller requires to the named pipe.

DesiredAccess Flags:

SYNCHRONIZE - The file handle may be waited on to synchronize with the completion of I/O operations.

READ_CONTROL - The ACL and ownership information associated with the named pipe may be read.

WRITE_DAC - The discretionary ACL associated with the named pipe may be written.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

WRITE_OWNER - Ownership information associated with the named pipe may be written.

FILE_READ_DATA - Data may be read from the named pipe.

FILE_WRITE_DATA - Data may be written to the named pipe.

FILE_CREATE_PIPE_INSTANCE - This access is needed to create subsequent instances of the named pipe.

FILE_READ_ATTRIBUTES - Named pipe attributes flags may be read.

FILE_WRITE_ATTRIBUTES - Named pipe attribute flags may be written.

The three following values are the generic access types that the caller may request along with their mapping to specific access rights:

GENERIC_READ - Maps to *FILE_READ_DATA* and *FILE_READ_ATTRIBUTES*.

GENERIC_WRITE - Maps to *FILE_WRITE_DATA* and *FILE_WRITE_ATTRIBUTES*.

GENERIC_EXECUTE - Maps to *SYNCHRONIZE*.

ObjectAttributes - A pointer to a structure that specifies the object attributes; refer to the I/O System Specification for details.

IoStatusBlock - A pointer to a structure that receives the final completion status. The actual action taken by the system is written into the *Information* field of this structure. For example, it indicates if a new named pipe and instance was created or just a new instance.

ShareAccess - Specifies the share access and configuration of the named pipe.

ShareAccess Flags:

FILE_SHARE_READ - Indicates that client end handles can be opened for read access to the named pipe.

FILE_SHARE_WRITE - Indicates that client end handles can be opened for write access to the named pipe.

CreateDisposition - Specifies the action to be taken if the named pipe does or does not already exist.

CreateDisposition Values:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_CREATE - Indicates that if the named pipe already exists, then the operation should fail. If the named pipe does not already exist, then the first instance of the named pipe should be created.

FILE_OPEN - Indicates that if the named pipe already exists, then another instance of the named pipe should be created. If the named pipe does not already exist, then the operation should fail.

FILE_OPEN_IF - Indicates that if a named pipe already exists, then another instance of the named pipe should be created. If the named pipe does not already exist, then the first instance of the named pipe should be created.

CreateOptions - Specifies the options that should be used when creating the first instance or a subsequent instance of a named pipe.

CreateOptions Flags:

FILE_SYNCHRONOUS_IO_ALERT - Indicates that all operations on the named pipe are to be performed synchronously. Any wait that is performed on behalf of the caller is subject to premature termination by alerts.

FILE_SYNCHRONOUS_IO_NONALERT - Indicates that all operations on the named pipe are to be performed synchronously. Any wait that is performed on behalf of the caller is not subject to premature termination by alerts.

NamedPipeType - Specifies the type of the named pipe. This parameter is only meaningful when the first instance of a named pipe is created.

NamedPipeType Values:

FILE_PIPE_MESSAGE_TYPE - Indicates that the named pipe is a message pipe. Data written to the pipe is stored such that message boundaries are maintained. Message named pipes can be read in message mode or in byte stream mode.

FILE_PIPE_BYTE_STREAM_TYPE - Indicates that the named pipe is a byte stream pipe. Data written to the pipe is stored as a continuous stream of bytes. Byte stream pipes can only be read in byte stream mode.

ReadMode - Specifies the mode in which the named pipe is read.

ReadMode Values:

FILE_PIPE_MESSAGE_MODE - Indicates that data is read from the named pipe a message at a time. This value may not be specified unless the named pipe is a message pipe.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FILE_PIPE_BYTE_STREAM_MODE - Indicates that data is read from the named pipe as a continuous stream of bytes. This value may be specified regardless of the type of the named pipe.

CompletionMode - Specifies whether I/O operations are to be queued or completed immediately when conditions are such that the I/O operation cannot be completed without being deferred for subsequent processing, e.g., a read operation on a named pipe that contains no write data.

CompletionMode Values:

FILE_PIPE_QUEUE_OPERATION - Indicates that I/O operations are to be queued pending completion at a later time if they cannot be immediately completed when the I/O operation is issued.

FILE_PIPE_COMPLETE_OPERATION - Indicates that I/O operations are not to be queued if they cannot be completed immediately when the I/O operation is issued.

MaximumInstances - Specifies the maximum number of simultaneous instances of the named pipe. This parameter is only meaningful when the first instance of a named pipe is created.

InboundQuota - Specifies the pool quota that is reserved for writes to the inbound side of the named pipe.

OutboundQuota - Specifies the pool quota that is reserved for writes to the outbound side of the named pipe.

DefaultTimeout - Specifies an optional pointer to a timeout value that is to be used if a timeout value is not specified when waiting for an instance of a named pipe. This parameter is only meaningful when the first instance of a named pipe is created.

This service either creates the first instance of a specific named pipe and establishes its basic attributes or creates a new instance of an existing named pipe which inherits the attributes of the first instance of the named pipe. If creating a new instance of an existing named pipe the user must have *FILE_CREATE_PIPE_INSTANCE* access to the named pipe object.

If a new named pipe is being created, then the Access Control List (ACL) from the object attributes parameter defines the discretionary access control for the named pipe. If a new instance of an existing named pipe is created, then the ACL is ignored.

If a new named pipe is created, then the configuration of the named pipe is determined from the *FILE_SHARE_READ* and *FILE_SHARE_WRITE* flags of the share access parameter. If both flags are specified, then the named pipe is a full duplex pipe and can be read and written by clients. If either one or the other is specified, but not both, then the named pipe is a simplex pipe and can only be read (outbound) or written (inbound) by clients. If neither one is specified, then

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

STATUS_INVALID_PARAMETER is returned. If a new instance of an existing named pipe is created, then the share access parameter is ignored.

If a new named pipe is created, then the type of the named pipe, the maximum instances, and the default timeout value are taken from their corresponding parameters. If a new instance of an existing named pipe is created, then these parameters are ignored.

The create options, completion mode, and read mode are set to their specified values.

The actual pool quota that is reserved for each side of the named pipe is either the system default, the system minimum, the system maximum, or the specified quota rounded up to the next allocation boundary.

The name of the named pipe is taken from the object attributes parameter, which must be specified.

An instance of a named pipe is always deleted when the last handle to the instance of the named pipe is closed.

If *STATUS_SUCCESS* is returned as the service status, then a new instance of a named pipe was successfully created. The *Information* field of the I/O status block indicates if this is the first instance of the named pipe (*FILE_CREATED*) or a new instance of an existing named pipe (*FILE_OPENED*).

If *STATUS_INVALID_PARAMETER* is returned as the service status, then an invalid value was specified for one or more of the input parameters.

If *STATUS_INSTANCE_NOT_AVAILABLE* is returned as the service status, the named pipe already exists and creating another instance would cause the maximum number of instances to be exceeded.

5.2 Create File

The **NtCreateFile** function can be used to open a client end handle to an instance of a specified named pipe.

In order to use this function to open a named pipe, the named pipe must already exist and the *CreateDisposition* value must be specified as either *FILE_OPEN* or *FILE_OPEN_IF*.

When a named pipe is opened, a search is conducted for an available instance of the specified named pipe. If an instance of the named pipe is found that has a state of listening, then the state of the named pipe is set to connected, the read mode is set to byte stream, the completion mode is set to queued operation, and the open I/O request is completed. If one or more listen I/O requests are pending for the server end of the named pipe, then the listen I/O requests are completed with a status of *STATUS_SUCCESS*.

If a named pipe of specified name cannot be found, then *STATUS_OBJECT_NAME_NOT_FOUND* is returned as the service status.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If an instance of the named pipe cannot be found with a state of listening, then *STATUS_PIPE_NOT_AVAILABLE* is returned as the service status.

5.3 Open File

The **NtOpenFile** function can be used to open a client end handle to an instance of a specified named pipe.

When a named pipe is opened, a search is conducted for an available instance of the specified named pipe. If an instance of the named pipe is found that has a state of listening, then the state of the named pipe is set to connected, the read mode is set to byte stream, the completion mode is set to queued operation, and the open I/O request is completed. If one or more listen I/O requests are pending for the server end of the named pipe, then the listen I/O requests are completed with a status of *STATUS_SUCCESS*.

If a named pipe of specified name cannot be found, then *STATUS_OBJECT_NAME_NOT_FOUND* is returned as the service status.

If an instance of the named pipe cannot be found with a state of listening, then *STATUS_PIPE_NOT_AVAILABLE* is returned as the service status.

5.4 Read File

The **NtReadFile** function can be used to read data from a named pipe. Data is read according to the read mode of the specified named pipe and I/O operations are completed according to the completion mode of the specified named pipe.

The byte offset and key parameters of the **NtReadFile** function are ignored by the named pipe file system.

The specified named pipe must be in the connected or closing state in order to read information from the pipe.

If *STATUS_PENDING* is returned as the service status, then the read I/O operation is pending and its completion must be synchronized using the standard **NT OS/2** mechanisms. Any other service status indicates that the read I/O operation has already been completed. If a success status is returned, then the I/O status block contains the I/O completion information. Otherwise, the service status determines any error that may have occurred.

If the specified handle is not open to a named pipe that is in the connected or closing state, then *STATUS_INVALID_PIPE_STATE* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the read buffer became inaccessible after it was probed for write access and the I/O status block contains the number of bytes that were read.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the I/O status *STATUS_END_OF_FILE* is returned, then there is no data in the pipe and the write end of the pipe has been closed.

The I/O status *STATUS_PIPE_EMPTY* is returned when there is no data in the pipe but the write end of the pipe is still opened and the pipe is opened for complete operations.

If the I/O status *STATUS_BUFFER_OVERFLOW* is returned, then the read I/O operation was completed successfully, but the size of the input buffer was not large enough to hold the entire input message. A full buffer of data is returned; additional data can be read from the message using the **NtReadFile** function. The I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_SUCCESS* is returned, then the read I/O operation was completed successfully and the I/O status block contains the number of bytes that were read.

If an event is associated with the write end of the specified named pipe and any data is actually read from the pipe, then the event is set to the Signaled state. Writers can use this information to synchronize their access to the named pipe.

5.5 Write File

The **NtWriteFile** function can be used to write data to a named pipe. Data is written according to the type of the specified named pipe and I/O operations are completed according to the completion mode of the specified named pipe.

The byte offset and key parameters of the **NtWriteFile** function are ignored by the named pipe file system.

The specified named pipe must be in the connected state in order to write information to the pipe.

If *STATUS_PENDING* is returned as the service status, then the write I/O operation is pending and its completion must be synchronized using the standard **NT OS/2** mechanisms. Any other service status indicates that the write I/O operation has already been completed. If a success status is returned, then the I/O status block contains the I/O completion information. Otherwise, the service status determines any error that may have occurred.

If the specified handle is not open to a named pipe that is in the connected state, then *STATUS_INVALID_PIPE_STATE* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the write buffer became inaccessible after it was probed for read access and the I/O status block contains the number of bytes that were written.

If the I/O status *STATUS_SUCCESS* is returned, then the write I/O operation was completed successfully and the I/O status block contains the number of bytes that were written.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If an event is associated with the read end of the specified named pipe and any data is actually written to the pipe, then the event is set to the Signaled state. Readers can use this information to synchronize their access to the named pipe.

A zero length write to a message type pipe adds a logical EOF to the to the pipe.

5.6 Read Terminal File

This function is not supported by named pipes.

5.7 Query Directory Information

The **NtQueryDirectoryFile** function can be used to enumerate files within the root named pipe file system directory (i.e., "\Device\NamedPipe\"). All the standard **NT OS/2** information classes are supported. **NtOpenFile** is used to open the root named pipe directory. This function is not supported for remote named pipes.

5.8 Notify Change Directory

The **NtNotifyChangeDirectoryFile** function can be used to monitor modifications to the root named pipe file system directory. The standard **NT OS/2** capabilities are supported. This function is not supported for remote named pipes.

5.9 Query File Information

Information about a file can be obtained with the **NtQueryInformationFile** function. All information classes, with the exception of extended attribute information, are supported for named pipes with special interpretation of the returned data as appropriate. An additional information class is also provided to return information that is specific to named pipes.

Information is returned by the named pipe file system for named pipes and for the named pipe root directory. The following subsections describe the information that is returned for named pipe entries. The information returned for the root directory is identical to the information that is returned by other file systems and is described in the **NT OS/2** I/O System Specification.

5.9.1 Basic Information

Basic information about a named pipe includes the creation time, the time of the last access, the time of the last write, the time of the last change, and the attributes of the named pipe. The file attribute value for a named pipe is **FILE_ATTRIBUTE_NORMAL**. This function is only supported by local named pipes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.9.2 Standard Information

Standard information about a named pipe includes the allocation size, the end of file offset, the device type, the number of hard links, whether a delete is pending, and the directory indicator. This function is only supported by local named pipes.

The allocation size is the amount of pool quota charged to the creator of an instance of a named pipe. This is the sum of the quota charged for the inbound and outbound buffers. The end of file offset is the number of bytes that are available in the inbound buffer. The device type is **FILE_DEVICE_NAMED_PIPE**, the number of hard links is one, delete pending is TRUE, and the directory indicator is FALSE.

5.9.3 Internal Information

Internal information about a named pipe includes a named pipe file-system-specific identifier. This value is unique for each instance of a named pipe.

5.9.4 Extended Attribute Information

The extended attribute information size is always returned as zero by the named pipe file system. This function is only supported by local named pipes.

5.9.5 Access Information

Access information about a named pipe includes the granted access flags. This function is only supported by local named pipes.

5.9.6 Name Information

Name information about a named pipe includes the name of the named pipe. This function is only supported by local named pipes.

5.9.7 Position Information

Position information about a named pipe includes the current byte offset. The current byte offset is the number of bytes that are available in the input buffer. This function is only supported by local named pipes.

5.9.8 Mode Information

Mode information about a named pipe includes the I/O mode of the named pipe. This function is only supported by local named pipes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.9.9 Alignment Information

The alignment information class is not supported by the named pipe file system. This function is only supported by local named pipes.

5.9.10 All Information

The all information class includes information that can be returned by all file systems and is described above under each of the individual subsections. This function is only supported by local named pipes.

5.9.11 Pipe Information

Pipe information for both local and remote named pipes include the read and completion mode for the specified end of the named pipe. An access of FILE_READ_ATTRIBUTE is required to query the pipe information of a named pipe.

FilePipeQueryInformation - Data type is *FILE_PIPE_INFORMATION*.

typedef struct **_FILE_PIPE_INFORMATION** {
    ULONG *ReadMode*;
    ULONG *CompletionMode*;
} **FILE_PIPE_INFORMATION**;

FILE_PIPE_INFORMATION:

ReadMode - The mode in which the named pipe is being read (*FILE_PIPE_MESSAGE_MODE* or *FILE_PIPE_BYTE_STREAM_MODE*).

CompletionMode - The mode in which I/O operations are handled (*FILE_PIPE_QUEUE_OPERATION* or *FILE_PIPE_COMPLETE_OPERATION*).

5.9.12 Local Pipe Information

Information for a local named pipe includes the type of the pipe, the maximum number of instances of the named pipe that can be created, the current number of instances of the named pipe, the quota charged for the input buffer, the number of bytes of data available in the input buffer, the quota charged for the output buffer, the quota available for writing into the output buffer, the state of the named pipe, and the end of the named pipe. An access of FILE_READ_ATTRIBUTE is required to query the local pipe information of a named pipe. This function is only supported by local named pipes.

FilePipeQueryInformation - Data type is *FILE_PIPE_LOCAL_INFORMATION*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

typedef struct **_FILE_PIPE_LOCAL_INFORMATION** {
    ULONG *NamedPipeType*;
    ULONG *NamedPipeConfiguration*;
    ULONG *MaximumInstances*;
    ULONG *CurrentInstances*;
    ULONG *InboundQuota*;
    ULONG *ReadDataAvailable*;
    ULONG *OutboundQuota*;
    ULONG *WriteQuotaAvailable*;
    ULONG *NamedPipeState*;
    ULONG *NamedPipeEnd*;
} **FILE_PIPE_LOCAL_INFORMATION**;

FILE_PIPE_LOCAL_INFORMATION:

NamedPipeType - The type of the named pipe (*FILE_PIPE_MESSAGE_TYPE* or *FILE_PIPE_BYTE_STREAM_TYPE*).

NamedPipeConfiguration - The configuration of the named pipe (*FILE_PIPE_INBOUND*, *FILE_PIPE_OUTBOUND*, *FILE_PIPE_FULL_DUPLEX*).

MaximumInstances - The maximum number of simultaneous instances of the named pipe that are allowed.

CurrentInstances - The current number of instances of the named pipe. For a remote named pipe this field is set to **MAXULONG**.

InboundQuota - The amount of pool quota that is reserved for buffering writes to the inbound side of the named pipe. For a remote named pipe this field is set to **MAXULONG**.

ReadDataAvailable - The number of bytes of read data that are available in the input buffer. For a remote named pipe this field is set to **MAXULONG**.

OutboundQuota - The amount of pool quota that is reserved for buffering writes to the outbound side of the named pipe. For a remote named pipe this field is set to **MAXULONG**.

WriteQuotaAvailable - The number of bytes of pool quota that are available for writing data. For a remote named pipe this field is set to **MAXULONG**.

NamedPipeState - The current state of the named pipe (*FILE_PIPE_DISCONNECTED_STATE*, *FILE_PIPE_LISTENING_STATE*, *FILE_PIPE_CONNECTED_STATE*, or *FILE_PIPE_CLOSING_STATE*).

NamedPipeEnd - The end of the pipe that is referred to by the specified open file handle (*FILE_PIPE_CLIENT_END* or *FILE_PIPE_SERVER_END*).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.9.13 Remote Pipe Information

Information for a remote named pipe includes the collect data time and the maximum collection count for the specified named pipe. An access of FILE_READ_ATTRIBUTE is required to query the pipe information of a named pipe. This function is only supported by remote named pipes.

FilePipeQueryInformation - Data type is *FILE_PIPE_REMOTE_INFORMATION*.

typedef struct **_FILE_PIPE_REMOTE_INFORMATION** {
    TIME *CollectDataTime*;
    ULONG *MaximumCollectionCount*;
} **FILE_PIPE_REMOTE_INFORMATION**;

FILE_PIPE_REMOTE_INFORMATION:

CollectDataTime - Specifies the amount of time that the workstation collects data to send to the remote named pipe before it sends it.

MaximumCollectionCount - Specifies the maximum number of bytes that the workstation stores before it sends data to the remote named pipe.

5.10 Set File Information

Information about a file can be changed with the **NtSetInformationFile** function. Most information classes are supported for local named pipes with the exception of link and position information.

Information can be set for named pipes. The following subsections describe the information that can be set for named pipes.

5.10.1 Basic Information

Basic information about a named pipe that can be set includes the creation time, the time of the last access, the time of the last write, the time of the last change, and the attributes of the named pipe. This function is only supported by local named pipes.

The associated times included in this class can be set to any appropriate value. The file attribute field can only be set to **FILE_ATTRIBUTE_NORMAL**.

5.10.2 Disposition Information

The disposition information class is not supported by named pipes.

Named pipes are always considered temporary and are deleted when the last handle is closed (i.e., when the last instance of a named pipe is closed and deleted the named pipe, itself, is also deleted.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.10.3 Link Information

This information class is not supported by named pipes.

5.10.4 Position Information

This information class is not supported by named pipes.

5.10.5 Mode Information

Mode information about a named pipe that can be set includes the I/O mode of the named pipe.

5.10.6 Pipe Information

Pipe information about a named pipe that can be set includes the read mode and completion mode of the named pipe. No special access is required to set the pipe information.

FilePipeSetInformation - Data type is *FILE_PIPE_INFORMATION*.

```
typedef struct _FILE_PIPE_INFORMATION {
        ULONG ReadMode;
        ULONG CompletionMode;
} FILE_PIPE_INFORMATION;
```

FILE_PIPE_INFORMATION:

> *ReadMode* - The mode in which the named pipe is to be read (*FILE_PIPE_MESSAGE_MODE* or *FILE_PIPE_BYTE_STREAM_MODE*).
>
> *CompletionMode* - The mode in which I/O operations are to be handled (*FILE_PIPE_QUEUE_OPERATION* or *FILE_PIPE_COMPLETE_OPERATION*).

If the type of the specified named pipe is a byte stream pipe and the new read mode is message mode, then *STATUS_INVALID_PARAMETER* is returned as the service status.

If the new completion mode for the specified named pipe is complete operations, the current completion mode is queue operations, and one or more I/O operations are currently queued to the specified end of the named pipe, then *STATUS_PIPE_BUSY* is returned as the service status and no pipe information is changed.

If the new read mode and the new completion mode are compatible with the current state of the specified named pipe, then the set information I/O request is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.10.7 Remote Pipe Information

Information about a remote named pipe that can be set includes the collect data time and the maximum collection count. No special access is required to set the pipe information.

FilePipeSetInformation - Data type is *FILE_PIPE_REMOTEINFORMATION*.

typedef struct **_FILE_PIPE_REMOTE_INFORMATION** {
    TIME *CollectDataTime*;
    ULONG *MaximumCollectionCount*;
} **FILE_PIPE_REMOTE_INFORMATION**;

FILE_PIPE_REMOTE_INFORMATION:

CollectDataTime - Sets the amount of time that the workstation can collect before sending it to the remote named pipe.

MaximumCollectionCount - Sets the maximum number of bytes that the workstation stores before sending data to the remote named pipe.

5.11 Query Extended Attributes

This function is not supported by named pipes.

5.12 Set Extended Attributes

This function is not supported by named pipes.

5.13 Lock Byte Range

This function is not supported by named pipes.

5.14 Unlock Byte Range

This function is not supported by named pipes.

5.15 Query Volume Information

This function is not supported by named pipes.

5.16 Set Volume Information

This function is not supported by named pipes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.17 File Control Operations

The following subsections describe file control operations that can be performed using a handle that is open to an instance of a named pipe. Certain functions can only be executed using a handle that is open to the server end of a named pipe. These functions are not legal for a handle that is open to the client end of a named pipe. The wait for named pipe instance function and the query event information function both require a handle that is open to the named pipe file system itself.

5.17.1 External File Control Operations

External file control operations can be executed by all users of the **NT OS/2** named pipe facilities and do not require any special privileges.

5.17.1.1 Assign Event

The assign event file control operation associates or disassociates an event object with either the client or server end of a named pipe. This function is only supported by local named pipes.

The control code for this operation is *FSCTL_PIPE_ASSIGN_EVENT*. The input buffer parameter specifies the event handle and key value that are to be associated with the respective end of the named pipe. The input buffer has the following format:

```
typedef struct _FILE_PIPE_ASSIGN_EVENT_BUFFER {
    HANDLE EventHandle;
    ULONG KeyValue;
    } FILE_PIPE_ASSIGN_EVENT_BUFFER;
```

FILE_PIPE_ASSIGN_EVENT_BUFFER:

EventHandle - A handle to an event object that is to be associated with the respective end of the named pipe, or null if the currently associated event object is to be disassociated.

KeyValue - The key value that is to be associated with the respective end of the named pipe. If the event handle is null, then this parameter is ignored.

If the event handle is null, then any event object that is currently associated with the respective end of the named pipe is disassociated and the key value is ignored.

If the event handle is not null, then **WRITE** access to the event is required. Any previously associated event object is disassociated and the specified event and key value are associated with the respective end of the named pipe.

This operation is always completed immediately and never causes an I/O operation to be queued.

Assigning an event object to either the client or server end of a named pipe provides additional synchronization capabilities when I/O operations are completed immediately rather than being queued.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Once an event object is assigned, the event will be set to the Signaled state every time information is read from, or written to, the opposite end of the named pipe, or the opposite end of the named pipe is closed. The event object associated with the client end of the named pipe is also set to the Signaled state when a disconnect operation is performed on the server end of the pipe.

5.17.1.2 Disconnect

The disconnect file control operation disconnects an instance of a named pipe from a client and causes the named pipe to enter the disconnected state. Disconnecting a named pipe causes all data in the pipe to be discarded and no further access to the named pipe is allowed until a listen operation is performed. The function is only valid from the server end of a named pipe.

The control code for this operation is *FSCTL_PIPE_DISCONNECT*. The input and output parameter buffers are not used.

If the specified handle is not open to the server end of a named pipe, then *STATUS_ILLEGAL_FUNCTION* is returned as the service status.

If the named pipe associated with the specified handle is in the disconnected state, then *STATUS_PIPE_DISCONNECTED* is returned as the service status.

If the named pipe associated with the specified handle is in the listening state, then the state of the named pipe is set to disconnected. If one or more listen I/O requests are waiting for a companion client open request, then the listen I/O requests are completed with a status of *STATUS_PIPE_DISCONNECTED*. The disconnect I/O request is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

If the named pipe associated with the specified handle is in the connected state, then the state of the named pipe is set to disconnected, all data in the input and output buffers is discarded, and outstanding client and server read and write I/O requests are completed with a status of *STATUS_PIPE_DISCONNECTED*. If an event object is associated with the client end of the named pipe, then the event is set to the Signaled state. The disconnect I/O request is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

If the named pipe associated with the specified handle is in the closing state, then the state of the named pipe is set to disconnected, all data in the input buffer is discarded, and outstanding server read I/O requests are completed with a status of *STATUS_PIPE_DISCONNECTED*. The disconnect I/O request is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

5.17.1.3 Listen

The listen file control operation is used to transition a named pipe from a disconnected state to a listening state. When a named pipe is in the listening state, client open requests can be satisfied and cause the named pipe to transition to the connected state. This function is only supported by local named pipes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The control code for this operation is *FSCTL_PIPE_LISTEN*. The input and output parameter buffers are not used.

If the specified handle is not open to the server end of a named pipe, then *STATUS_ILLEGAL_FUNCTION* is returned as the service status.

If the named pipe associated with the specified handle is in the closing state, then *STATUS_PIPE_CLOSING* is returned as the service status.

If the named pipe associated with the specified handle is in the connected state, then *STATUS_PIPE_CONNECTED* is returned as the service status.

If the named pipe associated with the specified handle is in the listening state and the completion mode associated with the server end handle is queue operations, then the listen I/O request is queued awaiting a companion client open request and *STATUS_PENDING* is returned as the service status. Otherwise (the completion mode is complete operations), *STATUS_PIPE_LISTENING* is returned as the service status.

If the named pipe associated with the specified handle is in the disconnected state, then the state of the pipe is set to listening and any outstanding wait for named pipe I/O requests are completed with a status of *STATUS_SUCCESS*.

If the completion mode associated with the server end handle is complete operations, then the listen I/O request is completed with an I/O status of *STATUS_PIPE_LISTENING* and *STATUS_SUCCESS* is returned as the service status.

If the completion mode associated with the server end handle is queue operations, then the listen I/O request is queued awaiting a companion client open request and *STATUS_PENDING* is returned as the service status. When a client open is performed, the listen I/O request is completed with an I/O status of *STATUS_PIPE_CONNECTED*.

5.17.1.4 Peek

The peek file control operation reads data from a named pipe in either byte stream or message mode, but does not actually remove the data from the pipe.

The control code for this operation is *FSCTL_PIPE_PEEK*. The output buffer parameter specifies the read buffer for the peek operation. The output buffer has the following format:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef struct _FILE_PIPE_PEEK_BUFFER {
        ULONG NamedPipeState;
        ULONG ReadDataAvailable;
        ULONG NumberOfMessages;
        ULONG MessageLength;
        CHAR Data[];
        } FILE_PIPE_PEEK_BUFFER;
```

FILE_PIPE_PEEK_BUFFER:

NamedPipeState - The current state of the named pipe (*FILE_PIPE_DISCONNECTED_STATE, FILE_PIPE_LISTENING_STATE, FILE_PIPE_CONNECTED_STATE*, or *FILE_PIPE_CLOSING_STATE*).

ReadDataAvailable - The number of bytes of read data that are available in the input buffer.

NumberOfMessages - The number of messages that are currently in the named pipe. If the named pipe is a message pipe, then this field contains the number of messages. Otherwise, this field contains zero.

MessageLength - The number of bytes that are contained in the first message in the named pipe. If the named pipe is a message type pipe, then this field contains the size of the first message. Otherwise, this field contains zero.

Data - A buffer that receives data read from the named pipe. The number of bytes of data that were read from the named pipe can be calculated from the I/O status block.

The specified named pipe must be in the connected or closing state in order to read information from the pipe.

This function is nearly identical to the **NtReadFile** function for a named pipe; however, no data is actually removed from the pipe and the operation is always completed immediately, i.e., it never causes an I/O operation to be queued.

If the specified handle is not open to a named pipe that is in the connected or closing state, then *STATUS_INVALID_PIPE_STATE* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the output buffer became inaccessible after it was probed for write access and the I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_END_OF_FILE* is returned, then there is no data in the pipe and the write end of the pipe has been closed.

If the I/O status *STATUS_BUFFER_OVERFLOW* is returned, then the peek I/O operation was completed successfully, but the size of the output buffer was not large enough to hold the entire input

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

message. A full buffer of data is returned; the actual message size can be determined from information placed in the output buffer. The I/O status block contains the number of bytes that were read including the named pipe information.

If the I/O status *STATUS_SUCCESS* is returned, then the peek I/O operation was completed successfully and the I/O status block contains the number of bytes that were read including the named pipe information.

5.17.1.5 Query Event Information

The query event information file control operation returns information about each named pipe that a specified event object is associated with in the current process. It does not return information about named pipes that are associated with the specified event object in other processes. This function can only be executed using a handle that is open to the named pipe file system itself. This function is only supported by local named pipes.

The control code for this operation is *FSCTL_PIPE_QUERY_EVENT*. The input buffer specifies the handle for the event object that is to be queried. The output buffer parameter specifies the information buffer for the query operation. Each entry returned in the output buffer has the following format:

```
typedef struct _FILE_PIPE_EVENT_BUFFER {
    ULONG NamedPipeState;
    ULONG EntryType;
    ULONG ByteCount;
    ULONG KeyValue;
    ULONG NumberRequests;
    } FILE_PIPE_EVENT_BUFFER;
```

<u>FILE_PIPE_EVENT_BUFFER:</u>

NamedPipeState - The current state of the named pipe (*FILE_PIPE_DISCONNECTED_STATE, FILE_PIPE_LISTENING_STATE, FILE_PIPE_CONNECTED_STATE*, or *FILE_PIPE_CLOSING_STATE*).

EntryType - The type of entry (*FILE_PIPE_READ_DATA* or *FILE_PIPE_WRITE_SPACE*).

ByteCount - The number of bytes of read data that are available (entry type is *FILE_PIPE_READ_DATA*) or the number of bytes of available write space (entry type is *FILE_PIPE_WRITE_SPACE*).

KeyValue - The key value that is associated with the named pipe.

NumberRequests - The number of read I/O requests that are queued (entry type is *FILE_PIPE_WRITE_SPACE*) or the number of write I/O requests that are queued (entry type is *FILE_PIPE_READ_DATA*) to the opposite end of the named pipe.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This operation is always completed immediately and never causes an I/O operation to be queued.

If a named pipe that is associated with the specified event has both read data available and write space available, then two entries are returned in the output buffer.

If the specified handle is not an event object, then *STATUS_INVALID_PARAMETER* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the information buffer became inaccessible after it was probed for write access and the I/O status block contains the number of bytes of information that were returned.

If the I/O status *STATUS_SUCCESS* is returned, then the query event I/O operation was completed successfully and the I/O status block contains the number of bytes of information that were returned.

5.17.1.6 Transceive

The transceive file control operation performs a write operation followed by a read operation on a named pipe such that no other operation can occur between the write and read operations on the corresponding end of the pipe.

The control code for this operation is *FSCTL_PIPE_TRANSCEIVE*. The output buffer parameter specifies the read buffer and the input buffer parameter specifies the data to be written.

The specified named pipe must be in the connected state in order to perform a transceive operation on the pipe. The named pipe must also be a message pipe, and the read mode of the named pipe must be message mode. The completion mode is ignored for the transceive operation and operations are always queued.

If *STATUS_PENDING* is returned as the service status, then the transceive I/O operation is pending and its completion must be synchronized using standard **NT OS/2** mechanisms. Any other service status indicates that the transceive I/O operation has already been completed. If a success status is returned, then the I/O status block contains the I/O completion information. Otherwise, the service status determines any error that may have occurred.

If the specified handle is not open to a named pipe that is in the connected state, then *STATUS_INVALID_PIPE_STATE* is returned as the service status.

If the read mode associated with the specified handle is not message mode, then *STATUS_INVALID_READ_MODE* is returned as the service status.

If a read I/O operation is already pending for the inbound side of the specified named pipe, or there is currently available data in the inbound side of the named pipe, then *STATUS_PIPE_BUSY* is returned as the service status.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the read buffer or the write buffer became inaccessible after it was probed for write access (read buffer) or read access (write buffer) and the I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_BUFFER_OVERFLOW* is returned, then the transceive I/O operation was completed successfully, but the size of the output buffer was not large enough to hold the entire input message. A full buffer of data is returned; additional data can be read from the message using the **NtReadFile** function. The I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_SUCCESS* is returned, then the transceive I/O operation was completed successfully and the I/O status block contains the number of bytes that were read.

If an event is associated with the opposite end of the specified named pipe, then the event is set to the Signaled state when the write part of the transceive operation is completed and when the read part of the transceive operation is completed. Readers and writers can use this information to synchronize their access to the named pipe.

5.17.1.7 Wait For Named Pipe

The wait for named pipe file control operation waits for an instance of a named pipe with a specified name to attain a state of listening. This function can only be executed using a handle that is open to the named pipe file system root directory (i.e., "\Device\NamedPipe\") or redirector (i.e., "\Device\LanmanRedirector").

The control code for this operation is *FSCTL_PIPE_WAIT*. The input buffer parameter specifies the device relative name of the named pipe, and an optional timeout value. The input buffer has the following format:

```
typedef struct _FILE_PIPE_WAIT_FOR_BUFFER {
    TIME Timeout;
    ULONG NameLength;
    BOOLEAN TimeoutSpecified;
    CHAR Name[]
    } FILE_PIPE_WAIT_FOR_BUFFER;
```

<u>FILE_PIPE_WAIT_FOR_BUFFER:</u>

Timeout - Supplies a new timeout value is use other than the default timeout for the named pipe. This value is only read if TimeoutSpecified is TRUE. A minimum large integer value (i.e., 0x8000000000000000) means to wait indefinitely.

NameLength - Supplies the length of the name of the named pipe found in this buffer.

TimeoutSpecified - Indicates if an overriding timeout value has been specified.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Name - Supplies the name of the named pipe. The name does not include the "\Device\NamedPipe\" or "\Device\LanmanRedirector\" prefix.

If an instance of a named pipe with the specified name is currently in the listening state, then the wait for named pipe I/O function is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status. Otherwise, the wait for named pipe I/O request is placed in the wait queue of the specified named pipe and *STATUS_PENDING* is returned as the service status.

If an instance of the specified named pipe does not attain a listening state within the specified timeout period (either the optional one supplied in this function or the default timeout period specified when the original instance of the named pipe was created). then the wait for named pipe I/O request is completed with a status of *STATUS_PIPE_WAIT_TIMEOUT*.

5.17.1.8 Impersonate

The impersonate file control operation allows the server end of the pipe to impersonate the client end. Whenever this function is called the named pipe file system changes the caller's thread to start impersonating the context of the last message read from the pipe. Only the server end of the pipe is allowed to invoke this function. This function is only supported by local named pipes.

The control code for this operation is *FSCTL_PIPE_IMPERSONATE*. The output and input parameter buffers are not used.

If the specified handle is not open to the server end of a named pipe, then *STATUS_ILLEGAL_FUNCTION* is returned as the service status.

If the named pipe associated with the specified handle is in the disconnected state, then *STATUS_PIPE_DISCONNTECTED* is returned as the service status.

If a read operation has never been completed to the server end of the named pipe, then *STATUS_CANNOT_IMPERSONATE* is returned as the service status.

If the impersonation is successful then the I/O function is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

5.17.2 Internal File Control Operations

Internal file control operations can only be executed by components that execute in kernel mode and directly build and submit I/O requests to the named pipe file system. These functions are only supported by local named pipes.

5.17.2.1 Internal Read

The internal read file control operation provides the capability to perform a read operation directly into a system buffer. No quota is charged nor are any buffers allocated by the named pipe file system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The control code for this operation is *FSCTL_PIPE_INTERNAL_READ*. The output buffer parameter specifies the system buffer into which information is to be read

5.17.2.2 Internal Write

The internal write file control operation provides the capability to perform a write operation directly from a system buffer. No quota is charged nor are any buffers allocated by the named pipe file system.

The control code for this operation is *FSCTL_PIPE_INTERNAL_WRITE*. The input buffer parameter specifies the system buffer from which information is to be written.

5.17.2.3 Internal Transceive

The internal transceive control operation provides the capability to perform a transceive operation directly into a system buffer. No quota is charged nor are any buffers allocated by the named pipe file system.

The control code for this operation is *FSCTL_PIPE_INTERNAL_TRANSCIEVE*. The input buffer parameter specifies the buffer from which information is to be written, while the output buffer parameter specifies the system buffer into which information is to be read.

5.18 Flush Buffers

The **NtFlushBuffersFile** function can be used to wait until all currently buffered write data is read from the opposite end of the specified named pipe.

5.19 Set New File Size

This function is not supported by named pipes.

5.20 Cancel I/O Operation

The **NtCancelIoFile** function can be used to cancel all I/O operations that were issued by the subject thread for the specified named pipe. Both read and write operations initiated by the subject thread are canceled.

5.21 Device Control Operations

No device control operations are supported by the named pipe file system.

5.22 Close Handle

The **NtClose** function can be used to close a handle to the specified named pipe.

If the specified handle is the last handle that is open to the corresponding end of the specified named pipe, then the state of the named pipe is set to closing. Read and write operations that are pending for the inbound side of the named pipe are completed with an I/O status of *STATUS_PIPE_CLOSED*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Write operations that are pending for the outbound side of the named pipe are allowed to complete and cause the close operation to remain pending until the opposite end of the named pipe is closed, disconnected, or the information is read from the pipe.

If an event is associated with the opposite end of the specified named pipe, then the event is set to the Signaled state. Readers and writers can use this information to synchronize their access to the named pipe.

6. OS/2 API Emulation

The following subsections discuss the emulation of the OS/2 named pipe facilities using the capabilities provided by **NT OS/2**. Only those OS/2 functions which require special handling with respect to named pipes are included.

6.1 DosCallNmPipe

This OS/2 API combines the function of an open, write, read, and a close of a named pipe.

This service can be emulated with the **NtOpen**, **NtFsControlFile** (*FSCTL_PIPE_TRANSCEIVE*), and **NtClose** services. There is no **NT OS/2** facility that will perform this function in a single operation.

6.2 DosConnectNmPipe

This OS/2 API causes an instance of a named pipe that is in the disconnected state to transition to the listening state and continues the execution of any clients that are waiting for an available instance of the specified named pipe. This function can only be executed using a handle that is associated with the server end of a named pipe.

This API can be emulated with the **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_LISTEN*. The OS/2 subsystem or client DLL issues the listen I/O request. If the completion mode associated with the specified named pipe handle is queue operations and the request cannot be immediately satisfied, then *STATUS_PENDING* is returned. For this case, the OS/2 subsystem or client DLL must wait for the I/O operation to complete.

6.3 DosDisconnectNmPipe

This OS/2 API causes an instance of a named pipe to enter the disconnected state. All data in the input and output buffers of the pipe are discarded and any outstanding read or write I/O requests are completed with an error status. This function can only be executed using a handle that is associated with the server end of a named pipe.

This API can be emulated with the **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_DISCONNECT*. The OS/2 subsystem or client DLL issues the disconnect I/O request.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.4 DosMakeNmPipe

This OS/2 API creates an instance of a named pipe and opens a server side handle to the newly created instance. If the newly created instance is the first instance of the named pipe, then the attributes of the named pipe are also defined.

This API can be emulated with the **NtCreateNamedPipeFile** service.

The OS/2 inheritance bit of the open mode is the same as the **NT OS/2** handle attributes field of the object attributes parameter.

The OS/2 write-behind bit of the open mode is the opposite of the **NT OS/2** *FILE_WRITE_THROUGH* flag of the create options parameter. Therefore, a particular OS/2-compatible behavior can be specified with the **NT OS/2** parameter.

The OS/2 access bits of the open mode are the same as the **NT OS/2** desired access parameter.

The **NT OS/2** share access flags are used to determine the configuration of the named pipe (i.e., full duplex or simplex).

The OS/2 wait bit of the pipe mode is the same as the **NT OS/2** completion mode parameter.

The OS/2 read bit of the pipe mode is the same as the **NT OS/2** read mode parameter.

The OS/2 pipe type bit of the pipe mode is the same as the **NT OS/2** pipe type parameter.

The OS/2 maximum instances field of the pipe mode is the same as the **NT OS/2** maximum instances parameter.

The OS/2 outbound buffer size is the same as the **NT OS/2** outbound quota parameter.

The OS/2 inbound buffer size is the same as the **NT OS/2** inbound quota parameter.

The OS/2 default timeout is the same as the **NT OS/2** default timeout parameter.

6.5 DosPeekNmPipe

This OS/2 API allows information to be read from a named pipe without actually removing the data from the pipe.

This API can be emulated with **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_PEEK*. The OS/2 subsystem or client DLL issues the peek I/O request. The request is completed immediately and the information returned in the output buffer and I/O status block can be used to generate the output values required by the OS/2 API.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.6 DosQNmPHandState

This OS/2 API returns information about the instance of a named pipe that is open to the specified handle.

This API can be emulated with the **NtQueryInformationFile** service by specifying the *FilePipeQueryInformation* information class.

6.7 DosQNmPipeInfo

This OS/2 API returns information about the instance of a named pipe that is open to the specified handle.

This API can be emulated with the **NtQueryInformationFile** service by specifying the *FilePipeQueryInformation* and *FileNameInformation* information classes.

6.8 DosQNmPipeSemState

This OS/2 API returns information about all named pipes that are associated with a specified semaphore handle.

This API can be emulated with **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_QUERY_EVENT*. The OS/2 subsystem or client DLL issues the query event I/O request. The request is completed immediately and the information returned in the output buffer and I/O status block can be used to generate the output values required by the OS/2 API.

6.9 DosRawReadNmPipe

This OS/2 API provides the capability to read all the available data, including message headers, from a named pipe.

This is an undocumented function in OS/2 and will not be implemented as a user-visible function by the OS/2 subsystem.

There seems to be no real use for this function.

6.10 DosRawWriteNmPipe

This OS/2 API provides the capability to write data, including message headers, to a named pipe.

This is an undocumented function in OS/2 and will not be implemented as a user-visible function by the OS/2 subsystem.

The only known user-level need for this function is to enable the writing of a zero length message to a message pipe. This capability will be provided in a different manner by the **NT OS/2** name pipe file system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.11 DosSetNmPHandState

This OS/2 API sets information about the instance of a named pipe that is open to the specified handle.

This API can be emulated with the **NtSetInformationFile** service by specifying the *FilePipeSetInformation* information class.

6.12 DosSetNmPipeSem

This API associates a semaphore and key value with a named pipe.

This API can be emulated with **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_ASSIGN_EVENT*.

6.13 DosTransactNmPipe

This OS/2 API combines the function of a write operation and a read operation on a named pipe. The transact operation is performed on the named pipe such that no other operation can occur between the write and read operations.

This API can be emulated with the **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_TRANSCEIVE* and then waiting for the I/O request to complete.

6.14 DosWaitNmPipe

This OS/2 API provides the ability for a client to wait until an instance of a named pipe with a specified name attains a state of listening.

This API can be emulated with the **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_WAIT* and then waiting for the I/O request to complete.

The I/O request will automatically be completed if the default timeout interval that was specified when the original instance of the named pipe was created is exceeded. If a timeout value is specified by the user, then the overriding timeout period should be used in the FSCTL pipe wait call.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft, February 16, 1990

Revision 1.1, March 8, 1990

1. Incorporate technical and editorial changes from internal review.

Revision 1.2, August 14, 1990

1. Removed directory hierarchy.
2. Removed raw mode read and write.
3. Added optionally timeout parameter to wait for named pipe.
4. Removed all references to EAs and symbolic links.
5. Minor editoral changes.

Revision 1.3, September 27, 1990

1. Removed owner information query/set operation.
2. Changed unbuffered read/write to internal read/write.
3. Added internal transceive operation.
4. Minor editoral changes.

Revision 1.4, October 17, 1990

1. Added impersonation.

Revision 1.5, January 23, 1991

1. Clarify that NtCreateNamedPipeFile and directory query are for local pipes only.
2. In query Pipe information state which fields remote pipes returns as **MAXULONG**.
3. Remove FILE_WRITE_THROUGH option in NtCreateNamedPipeFile.
4. Change wait for named pipe to take a handle to the root directory and not the file system itself.
5. Add remote named pipes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Subsystem Design Rationale

Author: *Mark H. Lucovsky*

Revision 1.3, June 1, 1989
Original Draft, May 26, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. The NT OS/2 Mission

The **NT OS/2** group was formed with a clear mission:

- o To design and implement an OS/2-compatible operating system for non-x86 hardware platforms
- o To support the APIs required by POSIX (IEEE Std 1003.1-1988) at a level required to pass government validation
- o To support symmetric multiprocessing
- o To provide C2 security features with a path to B1 and beyond
- o To provide easy portability to other 32-bit architectures
- o To design and implement the first functional system by the 3rd quarter of 1990
- o To target the system for a **Microsoft**-designed i860 PC hardware platform, followed shortly thereafter by an i860mp or N11 multi-processor server system

Conclusions from the January 1989 System Retreat indicated that **NT OS/2** is critical to the long-term growth of **Microsoft**. The design of the system must accommodate current and future needs of **Microsoft**. The design must be maintainable, and easily extensible.

2. Design Goals

In order to achieve our mission, the following set of prioritized goals was established:

1. Robustness. The highest priority for **NT OS/2** is robustness. The inner workings of the system should be straightforward and well defined. A complete and formal design on all components of the system must be produced and interfaces and behavior must be well specified. The system must be designed without "magic".

2. Extensibility and maintainability. **NT OS/2** must be designed with the future in mind. It should be easily extensible to meet the needs of our OEM customers and our own needs over time. The system should also be designed for maintainability.

   Given the state of the API sets that **NT OS/2** must support, its design must accommodate changes and future additions to those sets.

3. Portability. **NT OS/2** must be designed for portability. The system architecture must be portable across a number of platforms. There are portions of the actual implementation that will require a port when moving from platform to platform. The effort required to port **NT OS/2** from one platform to another must be less than, or equal to, an equivalent port of a UNIX or Mach system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Performance. Superior performance in **NT OS/2** is important. Algorithms and data structures that will lead to a high level of performance and that will provide us with the flexibility needed to achieve our other goals must be incorporated into the design. The granularity of locking, the various types of locks used in the system, the amount of time spent at an elevated interrupt level or with interrupts completely disabled must be carefully designed so that **NT OS/2** is a responsive system which can compete in a number of markets.

In addition to these goals, compatibility with OS/2 APIs and POSIX compliance are system constraints in **NT OS/2**.

3. Design Alternatives Investigated

Several design alternatives for **NT OS/2** were considered during the design phase.

The first design layered the POSIX API set on top of a slightly extended OS/2 API set. As the design progressed, it became apparent that this design would lead to a system that could not achieve the goals of robustness, maintainability, or extensibility. Problems encountered with a similar attempt in OS/2 led to considerable change in the base system capabilities, which further strengthened the belief that this was a poor alternative.

The next design implemented both OS/2 and POSIX API sets directly in the **NT OS/2** executive. This was an improvement on the previous design, but the large number of "chicken wire" and "voodoo" interfaces required by this design threatened the goals of extensibility and maintainability.

The third design implemented OS/2 and POSIX as protected subsystems outside the **NT OS/2** executive. Success with this type of client/server architecture in the academic community and at other research sites provides strong evidence that this design will allow **NT OS/2** to meet its goals of robustness, extensibility, maintainability, portability, and performance, and thus, achieve its mission. Therefore, this design was chosen for **NT OS/2**.

(The final section of this document examines the three **NT OS/2** design alternatives in greater detail.)

4. The NT OS/2 Design

The **NT OS/2** system design consists of a highly functional executive, which executes in kernel mode, and exports a native API (a set of system services). Operating system environments such as OS/2 and POSIX are implemented as protected subsystems outside the executive.

A protected subsystem executes in user mode as a regular (native) process. The subsystem may have amplified privileges, but it is not considered a part of the executive and, therefore, cannot bypass the system security architecture, or in any other way corrupt the system. Subsystems communicate with their clients and each other using a high-performance local (cross-process) procedure call, or LPC, mechanism. (A round-trip LPC completes in approximately 100usec on the i860.)

This **NT OS/2** design satisfies each of the goals for the system. The following attributes of the design ensure the primary goal of robustness:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- The kernel mode portion of the system exports well-defined APIs that, in general, do not have mode parameters or other "magical flags". Therefore, the APIs are simple to implement, easy to test, and easy to document.

- A formal design is being produced for all portions of the **NT OS/2** system prior to coding. This effort has led to well-documented interfaces for native services and internal functions.

- The partitioning of major components, such as PM, OS/2, and POSIX, into separate subsystems is resulting in simple, elegant designs in the subsystems. Each subsystem is optimized to implement only those features needed to provide its API set.

- With the prevalent use of frame-based exception handlers, **NT OS/2** and its subsystems are able to catch programming errors and filter bad or inaccessible parameters in an efficient and reliable manner.

The **NT OS/2** design also meets its goals of maintainability and extensibility through the following features:

- The **NT OS/2** design is simple and well documented. This, coupled with a common coding standard used throughout the system, should enable a programmer to work on any piece of the system without having to consult the "gurus" to learn about hidden rules, side effects, or "magical" programming tricks.

- By using subsystems to implement major portions of the system, **NT OS/2** isolates and controls dependencies. For example, the only piece of the **NT OS/2** system affected by the changing Cruiser design is the OS/2 subsystem. The design of the process structure, memory management, synchronization primitives, and so on, does not have to be put on hold. The same holds true for the evolving POSIX standards.

- As the needs of **Microsoft** grow, the **NT OS/2** system is prepared to accommodate those needs. Subsystems that provide additional functionality can be added to the system without impacting the base system. New subsystems can be added without having to modify the **NT OS/2** executive or release a new version of the system.

  Subsystems such as DOS, Windows, or Xenix can be added to the system if necessary. OEMs could continue to provide limited support for operating system environments other than the **Microsoft**-provided OS/2 and POSIX environments.

- Using the subsystem or "building block" approach, it is possible to envision a configuration that includes only the OS/2 subsystem. POSIX could be a revenue-producing, licensable option. If the option were not used, no system resources would be sacrificed.

- Subsystems need not bypass the security features present in **NT OS/2**. Rather, they can use the security features to their fullest extent.

NT OS/2 portability is ensured by the following:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Except for small, well-isolated sections of code, **NT OS/2** is written in C. The system is being developed on prototype compilers with limited functionality, and still, the design has yielded portable code.

- Using the UNIX and Mach porting experience of engineers on the project, the group has established that the **NT OS/2** will port to other platforms at least as easily as the UNIX or Mach operating systems. The effort involved in porting **NT OS/2** to another 32-bit, paged architecture, using readily available compilers, is small.

NT OS/2 is a high-performance system designed to run on high-performance hardware. We believe that the system will perform better than any system providing equivalent functionality on equivalent hardware. The following attributes of the system promote high performance:

- Algorithms and execution paths through the system have been carefully optimized to increase performance. Also, the modular nature of the system allows performance optimization by replacing entire components.

- System calls, exceptions (page faults), LPC, thread creation, and I/O have undergone scrutiny to ensure their speed. The round-trip time for a null system call is currently on the order of 3usec (on a 40Mhz i860). Given this number, **NT OS/2** performs better than most systems even after equalizing processor speeds.

- Ensuring high performance is an ongoing activity in the implementation of **NT OS/2**.

5. Performance in the Subsystem Model

Before committing the **NT OS/2** design to a subsystem, or client/server model, time was spent analyzing the Presentation Manager. One of the deficiencies in the current implementation of PM is that it must manage global state without having any way to protect the state. We worked with one of the designers and implementors of PM to develop a solution to this problem by making PM a protected subsystem (which executes in its own process context rather than in the context of the thread that called a PM entry point).

Before proceeding with the PM design, the **NT OS/2** LPC mechanism was designed. We felt that if the LPC design were solid, it could be modeled, and we could determine whether or not PM performance would be acceptable using a subsystem design model.

Ideas present in several high-performance LPC mechanisms were incorporated into the **NT OS/2** design:

- The ability to efficiently pass small amounts of data, as was done in Stanford's V system, is included.

- The idea of mapping large messages or passing large parameters "out-of-band" is similar to the mechanism used in Carnegie Mellon's Mach system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- The ability to pass message data through memory shared between the client application and the subsystem is similar to the technique used in an experimental system under development at the University of Washington, and which also appears in DEC's Topaz system.

With the design of the **NT OS/2** LPC mechanism complete, a model was created to measure the performance impact of running PM as a protected subsystem.

The model consisted of the following pieces of modified system software:

- OS/2 Kernel Modifications. A special version of OS/2 1.1 was built. This version of the system had an additional system service that simulated a context switch from the calling thread back to the calling thread.* All of the work involved in switching address spaces was simulated as well.

- *pmwin.dll* and *pmgpi.dll*. A new version of each of these libraries was created. For each entry point, the cost of marshalling its parameters into and out of a message buffer was simulated; two calls to the new context switch routine were done; and finally, a call was made to the original version of the entry point.

By running PM applications using the modified system software, we were able to determine exactly how much overhead PM would incur when run as a subsystem.

Several test cases were run on the model. These included running the PMBENCH benchmark suite, running PMDRAW and drawing complicated pictures, running various configurations of PM Excel and scrolling, drawing charts, and performing other screen manipulations, and finally, running a journaled interactive session with multiple PM applications doing different tasks, including menu and dialog box operations.

Before running our tests, we did not know what to expect. We felt that if the system did not feel sluggish, then the subsystem approach might be acceptable. After running all of our tests, we were surprised. The system performed so well that we could not tell the difference between the subsystem version of PM and the normal version of PM.

The following table shows a condensed listing of our benchmark results:

LPC PM Overhead	Standard PM Time	Subsystem PM Time	Difference

* This simulation involved invalidating mapping information, saving and restoring registers, and saving and restoring the mapping information.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PMBENCH Test Suite	5.14%	***	***	***
PMDRAW monticello	16.88%	12.403s	14.497s	2.094s
PMDRAW fish	8.80%	11.887s	12.940s	1.053s
Excel Scroll 1's	3.25%	30.880s	31.885s	1.005s
Excel Scroll Big	0.84%	63.060s	63.590s	0.530s
Excel Chart	9.65%	12.900s	14.145s	1.245s
Interactive	1.20%	335.510s	339.670s	4.160s
				=======
Average Overhead	2.16%	466.640s	476.727s	10.087s

From the results of our study, we felt that the additional overhead imposed by running PM as a protected subsystem was acceptable given the benefits of such a design. While there is measurable overhead, it is not detectable when sitting in front of a machine running interactive or graphics-intensive applications.

After determining that PM could be run as a protected subsystem without incurring unacceptable performance degradation, we looked at other areas of the system that would be cleaner to implement as a separate subsystem but would not impact overall system performance.

Given that OS/2 and POSIX had to be treated as partitioned code within the executive, they were natural candidates for implementation as protected subsystems. We believe that real OS/2 (and POSIX) applications will be more dependent on the performance of PM than any other portion of the system. The ratio of PM to operating system service calls is likely to range from 10:1 to 100:1. If PM is a good candidate for implementation as a protected subsystem, then operating system environments such as OS/2 or POSIX are also good (if not better) candidates.

6. Standards

During the initial design phase of **NT OS/2**, a great deal of time was spent examining ways to design a system that could support both the OS/2 and POSIX API sets. This job was complicated by the fact that both of the API sets we planned to support were moving targets. In fact, the Cruiser specification was not yet available; it is still evolving.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.1 OS/2 Standards

Our initial OS/2 API set centers around the evolving 32-bit Cruiser, or OS/2 2.0 API set. (The design of Cruiser APIs is being done in parallel with the **NT OS/2** design.) In some respects, this standard is harder to deal with than the POSIX standards. OS/2 is tied to the Intel x86 architecture and these dependencies show up in a number of APIs. Given the nature of OS/2 design (the joint development agreement), we have had little success in influencing the design of the 2.0 APIs so that they are portable and reasonable to implement on non-x86 systems. In addition, the issue of binary compatibility with OS/2 arises when the system is back-ported to an 80386 platform. This may involve 16-bit as well as 32-bit binary compatibility.

6.2 POSIX Standards

Our initial POSIX efforts center around the IEEE Std 1003.1-1988 (or Draft 13). The spec is vague in several areas and contains several optional features.

In order to sell in certain federal government markets, a POSIX implementation must be compliant with FIPS 151. This FIPS requires that certain optional features of POSIX be implemented, and also requires portions of other POSIX standards (1003.2, "Applications and Utilities"). In addition, the FIPS requires a certification of conformance. This certificate can be obtained by passing a certified POSIX test suite. The current set of test suites are developed by third parties, and do test for compliance with the POSIX spec. Unfortunately for us, the test suites were developed on UNIX systems that claim POSIX compliance. The test suites end up testing a lot of UNIX folklore that happens to be permissible under an interpretation of the POSIX spec.

To further complicate POSIX compliance, additional drafts of 1003.1, which are close to approval, have been proposed. The effects of approval are unknown. It is not clear if future additions to POSIX will be required under future FIPS, or if additions will be made optional. The government standards body that is issuing the FIPS is apparently ready to add any approved POSIX drafts to its FIPS. The latest draft under consideration (1003.1a), would add a number of features from Berkeley UNIX 4.3 to POSIX. It is anticipated that a new FIPS will be issued which requires these features in order to participate in certain government markets.

7. An Analysis of the Design Alternatives

Once the mission and goals of **NT OS/2** were clear, the design work was started. The most difficult portion of the design centered around the issue of how to provide OS/2 and POSIX compliance on the same system without failing to achieve our mission or compromising our goals.

Combining the APIs of multiple operating systems in a single system is always a difficult task. It does not matter whether the APIs are similar or different. The most striking example of this problem is the poor integration of UNIX variants found in the current UNIX market.

In the beginning (1982-1984), there were basically two branches in the UNIX tree. The BSD branch with Berkeley UNIX 4.2 and 4.3, and the AT&T System V branch with System V.2 and V.3.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Companies that offered pure systems in either camp were the norm. Companies in the scientific and engineering markets supported BSD while business-oriented companies supported System V:

- Sun 1.0-2.x was pure BSD
- DEC's ULTRIX was pure BSD
- Sequent was pure System V
- Altos was pure System V

After some time, companies began to offer systems with mixed features. This began with systems advertising "System V with BSD networking." Soon, nearly all companies offered systems with some features from both environments. Applications could call APIs from either set. If the API specified different behavior for a System V or a BSD implementation, it was usually a tossup as to which semantics were followed.

The current state of System V and BSD integration is the root of nearly all the confusion in the current UNIX marketplace. To port an application that was originally BSD to a system that is "System V with BSD features" requires elaborate configuration files that "pick and choose" the APIs. With each port to a new system, the configuration options and combinations must be expanded to accommodate the new system. The popular UNIX editor, emacs, is a perfect example of this. The emacs editor comes with nearly 50 configuration files. Each file describes a derivative of UNIX that has different features and supports a certain mix of BSD and System V APIs.

A major design issue in **NT OS/2** is to avoid the integration-of-features problem present in the current UNIX marketplace. **Microsoft** cannot afford to present POSIX and OS/2 integration as poorly as most of the UNIX vendors have.

In the selected **NT OS/2** design, an application that uses OS/2 APIs may only use OS/2 APIs. The POSIX API set is not available to the application. The reverse restriction is also true. POSIX applications may not call OS/2 APIs.

7.1 POSIX Layered on OS/2

The first alternative examined the feasibility of layering the POSIX API set as a runtime package on top of a native system service interface based on an OS/2 API set.

Using this approach, the **NT OS/2** executive would export an OS/2 2.0 API set. If there were functions that required extensions in order to make this work, we were prepared to make those extensions. An example of this approach is supporting POSIX *fork()* and *exec()* using OS/2's DosExecPgm().

We proposed adding a flag to DosExecPgm that would take one of the following values:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. The API should work exactly as the current DosExecPgm function works (that is, a new process is created and its address space is initialized so that it maps the image specified as the program name parameter).

2 The API should create a process and the address space should be an image of the address space of the calling process. Thread 1 should be created in the new process and its initial context should be identical to the context of the calling thread at the time of the call. The only exception is that thread 1 in the new process must return with a different return value than that returned by the calling thread.

3 The API should clean the address space of the process, terminate any threads in the process, create a new address space such that it maps the specified program image file, and create thread 1 so that it begins execution at the entry point specified in the image.

To implement OS/2 DosExecPgm, the API would be called with flag value 1. POSIX's *fork()* and *exec()* would be implemented using flag values 2 and 3.

On the surface, the above technique seems to work, but it is complicated. Complications arise in the following areas:

- File descriptors owned by a process would be dealt with differently in all three variations of DosExecPgm().

- File locks held by the process at the time of the API call would be handled differently for all three cases. In fact, since file locking itself is different, the case is really an 8-way case.

- Outstanding timers or process alarms have at least three different actions.

- Signals pending, or the state of a process's signal or exception handlers, is affected by the various API options.

The list of problems with this API is large, as should be clear from the above list. More important, the problem seems to scale exponentially. Simple operations like opening or creating files, establishing signal or exception handlers, reading from and writing to the terminal, or even manipulating regular files all have problems and virtually all require a mode argument.

One of the other serious problems with this design alternative is that it presents a poor integration of OS/2 and POSIX. It would be difficult to separate OS/2 calls from POSIX calls. Multi-threaded OS/2 applications that, either on purpose or as a result of a programming error, call DosExecPgm specifying a POSIX-oriented option would have disastrous effects. We could always say that this could not happen, but in order to achieve the robustness goals of the **NT OS/2** system, the executive would have to be coded so that it could handle all possible incorrect parameter combinations.

After determining that layering POSIX on top of OS/2 would bury much of POSIX in the executive, and would cause most of the overlapping APIs to require a mode parameter, we looked at ways of implementing the POSIX API set directly inside the **NT OS/2** executive.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.2 OS/2 and POSIX in the Executive

By implementing both the OS/2 and POSIX API sets directly within the executive, we were able to work on a layered, controllable design. The system would yield two API layers, one layer exporting OS/2 APIs and the other layer exporting POSIX APIs. The API layers would be implemented on top of an executive support layer.

The executive support layer would implement basic executive services such as process and address space management, thread creation/deletion/control, security, an I/O system and a file system. The executive support layer would control, create, and delete all state in the system. The API layers would simply call the executive support layer with appropriate parameters. They would not maintain state.

As we progressed with this design, it became clear that it was nearly identical to our initial design. Our proposals for the design of the process structure were not much different from the extensions that we had planned for DosExecPgm(). The primary difference was that the parameter combinations passed to the executive layer were controllable. Since the parameters came from the system code that implemented the API layers, we were able to make rules and declare that certain parameter combinations could not occur. This made the executive layer somewhat easier to write, but the rules for calling the executive became rather elaborate.

For **NT OS/2** to remain a product that could carry **Microsoft** through the 1990's, maintainability, extensibility, and robustness had to be ensured. It seemed that almost everything became an exception. The well-defined interfaces within the process structure became littered with exceptions and kludges needed to support the demands of POSIX's job control option or OS/2's complex process/command subtree relationships. Simple functions, such as waiting on a child process (common to both OS/2 and POSIX), became difficult to implement because the executive had to manage two slightly different cases.

As each new issue arose, the solution always seemed to have a common theme...

The terminal driver could look to see if the application writing to the terminal was a POSIX application. If so, then if the terminal was not the controlling terminal for the process, but the process was not ignoring SIGSTOP, then the process could be signaled and its parent notified.

or...

When a process terminates, look to see if it was an OS/2 application or if it was a POSIX application. If it was an OS/2 application that was *exec*'d using EXEC_SYNC, then after termination is complete, the process ID is available for re-use. If it was a POSIX application, then if the parent was not PID 1, signal it. If the process was a session group leader, then generate a SIGHUP signal to all members of the session group with the same controlling terminal, and possibly free the controlling terminal.

The more the design progressed, the more the system started to look like a bowl of spaghetti. Problems arose due to subtle differences between OS/2 and POSIX in almost all areas. The following are a few examples of the problems:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Process ID (PID). The POSIX job control option (required by FIPS 151) is difficult to implement correctly even on a BSD UNIX system. Process relationships and the lifetime of a PID are complex. A POSIX PID has nothing in common with an OS/2 PID other than sharing the same acronym.

  The standard solution to this sort of problem usually involved a "table off to the side" that could keep track of the differences. We had "tables off to the side" for POSIX and OS/2 process IDs, POSIX sessions, job control sessions, controlling terminal IDs, file and file system serial numbers (device, inode pairs, etc.), and others.

- o Exception handling. POSIX requires an exception handling mechanism based on signals that are similar to signals found in Berkeley UNIX 4.3. This architecture is drastically different from the current 16-bit OS/2 exception architecture and even more different than portions of the proposed OS/2 2.0 exception architecture.

  The exception architectures of both systems involve large portions of the entire system. The keyboard, video, and terminal drivers are involved, as is the process structure, system service dispatcher, trap handler, and so on.

  Trying to tie together these different pieces of the system in a way in which they could all participate in exception handling was seriously compromising the design of the system.

  The solution to this sort of problem usually involved adding fields to the process or thread structures to keep track of this. It became clear that our process and thread structures were going to be large. Much of the overhead was due to link words and pointers to the "tables off to the side," or to fields that were needed only if the process or thread represented a POSIX application (or OS/2 application).

- o Security. POSIX security impacts major pieces of the system. As the design progressed, it became clear that POSIX security was at odds with the Cruiser-like security scheme being designed for **NT OS/2**. Many features of the security scheme would have to be bypassed in order to implement the "hodge podge" of security features/APIs that appear in POSIX.

The list of chicken wire fixes is endless. Nearly all areas of the system are involved, including timers, time-of-day format, file locks, pipes, and many others.

The only advantage that this solution had over the previous one was that the API layer could call the executive support layer with a known set of parameter combinations. The executive support layer did not have to deal with illegal parameter combinations.

NT OS/2 had to explore some new alternatives. What we needed was a mechanism that would allow the OS/2 API layer to manage and control all state for all of the OS/2 applications in the system, and to allow the POSIX API layer to do the same for all of its applications. It was this realization that brought us to the current design strategy for **NT OS/2**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7.3 POSIX and OS/2 as Subsystems

The system architecture chosen for **NT OS/2** allows it to achieve its goals and, therefore, fulfill its mission. **NT OS/2** is designed with a small, non-preemptible kernel, which executes in kernel mode. A small but highly functional, preemptible, interruptible, and reentrant executive, which also executes in kernel mode and which exports a number of system service APIs, is layered on top of the kernel.

The APIs exported by the executive do not implement either the OS/2 or POSIX API sets. Instead, they export a set of APIs that allow both an OS/2 API set and a POSIX API set to be implemented entirely in user mode as separate processes running as protected subsystems. Using this approach, an OS/2 or POSIX API is emulated using the following sequence:

- An application calls the local stub for an API function.
- The stub packages the arguments into a message and transmits the message to either an OS/2 or POSIX subsystem using the **NT OS/2** local procedure call mechanism.
- The subsystem receives the message, implements the API, and replies to the application using LPC.
- The local stub receives the reply and returns the results to the application.

The APIs exported by the **NT OS/2** executive are powerful, but at the same time, are simple and straightforward. There are no cases in which a single flag parameter changes the entire meaning of an API. This design technique allows **NT OS/2** to achieve its goals of robustness, extensibility, and maintainability.

Implementing OS/2 and POSIX as subsystems allows each subsystem to implement only the set of semantics required by that subsystem. The requirements of the subsystems do not translate into "tables off to the side" or extra fields in data structures managed by the executive. When a subsystem needs to keep track of additional state associated with an object, it does so in its own data structures managed in the address space of the subsystem. This technique leads to more elegant solutions to problems posed by OS/2's process relationships or by POSIX's job control data structures.

Rather than having to bypass most of the security features present in **NT OS/2**, subsystems are able to use the security features to their fullest extent. The security architecture, along with the high performance LPC mechanism and powerful process structure and memory management APIs allow the subsystems to increase the robustness, extensibility and maintainability of the system while at the same time decreasing the demands on system resources.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Object Management Specification

Author: *Steven R. Wood*

Revision 1.6, May 24, 1991
Original Draft February 17, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview 1
  1.1 What is an Object? 1
  1.2 Object Management Goals 1
  1.3 Object Data Structures 1
  1.4 Object Header 1
  1.5 Object Types 2
  1.6 Object Handles 3
  1.7 Object Attributes Structure 4
  1.8 Resource Quotas and Objects 5
  1.9 Object Retention 6
  1.10 Exclusive Object Handles 6
  1.11 Object Name Space 7
  1.12 Preventing Deadlock 8

2. Object Executive APIs 9
  2.1 Creating Object Types 9
  2.2 Object Type Procedure Templates 11
    2.2.1 Object Dump Procedure 12
    2.2.2 Object Open Procedure 12
    2.2.3 Object Close Procedure 13
    2.2.4 Object Delete Procedure 14
    2.2.5 Object Parse Procedure 15
    2.2.6 Object Security Procedure 17
  2.3 Creating An Object 18
  2.4 Creating an Instance of an Object 20
  2.5 Open Object by Name 23
  2.6 Open Object by Pointer 24
  2.7 Referencing An Object 26
  2.8 Reference Object by Name 27
  2.9 Reference Object by Pointer 29
  2.10 Making an Object Temporary 30
  2.11 Dereferencing an Object 30
  2.12 Object Management during Process Creation and Deletion 31
    2.12.1 Process Creation Hook 31
    2.12.2 Process Deletion Hook 31
  2.13 Dump Object Support 32
  2.14 Check Traverse Access 34
  2.15 Check Create Instance access 35
  2.16 Check Create Object Access 36
  2.17 Check Implicit Object Access 37
  2.18 Checking Access for Object Reference 38
  2.19 Locking a security descriptor 39
  2.20 Unlocking a security descriptor 39
  2.21 Query an object's Security Descriptor field 39

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.22 Set an object's Security Descriptor field 40
2.23 Query an object's Security information 41
2.24 Release an object's Security information 41
2.25 Set Security Quota Charged for object 42
2.26 Validate security information against quota 43

3. Object System Services 43
3.1 Create Directory Object 43
3.2 Open Object Directory 45
3.3 Query Object Directory 46
3.4 Create Symbolic Link 48
3.5 Open Symbolic Link 49
3.6 Query Symbolic Link 50
3.7 Wait For Single Object 50
3.8 Wait for Multiple Objects 51
3.9 Duplicate Handle 53
3.10 Close Handle 54
3.11 Making an Object Temporary 55
3.12 Query Object 56
3.13 Set Security Descriptor for an Object 58
3.14 Query Security Descriptor for an Object 59

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This specification describes the Object Management for the NT OS/2 system. Object Management is provided by a set of routines that are available within the NT OS/2 executive and invoked from kernel mode. This specification also describes generic object management user level NT routines and support for directories.

1.1 What is an Object?

An object is an opaque data structure that defines a protected entity that is implemented and manipulated by the operating system. A particular object type is described by the set of operations that may be performed upon it (wait, create, clear, set, cancel,...) and its relationship to other objects. All objects have the same standard set of rules for creation, deletion, protection, access, management, and naming.

1.2 Object Management Goals

- Provide an extensible, well defined mechanism for the definition and manipulation of executive data structures.

- Provide uniform rules for object retention. This is especially important in a multiprocessor system.

- Provide uniform security and protection that allows certification at C2 and beyond without modification.

- Provide a mechanism to add new object types to the system without modifying existing system code. This means that only the object type specific routines should have knowledge of the internal structure of a particular object type.

- Provide orthogonal specification of APIs which operate on objects.

- Provide attributes on objects to support POSIX compatibility.

- Provide a naming hierarchy which is integrated with the file system and mimics the OS/2 and POSIX file system directory hierarchy.

1.3 Object Data Structures

An instance of an object type is represented by a data structure which contains a standard object header and an object type specific object body. The object management routines operate on the object header, while the object type's specific routines operate on the object body.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.4 Object Header

The object header contains information used by the object management routines to manipulate the object. The following items are maintained in the object header:

- o Pointer to the name of the object, if any.
- o Pointer to the directory object which contains this object's name, if any.
- o Pointer to the SecurityDescriptor for the object, if any.
- o AccessMode of the object, either KernelMode only or UserMode and KernelMode.
- o Pointer to the Owner Process of the object for exclusive objects, if any.
- o Retention counts for the object.
- o Pointer to an optional handle count data base, that maintains a per process handle count for a given object.
- o Pointer to the object type structure that defines the type of the object.
- o Permanent / temporary attribute.
- o Paged and nonpaged pool quota charges associated with the object.
- o Structure control linking all objects of the same type together.

1.5 Object Types

Every object has an object type. The object type is defined by an Object Type Descriptor structure. An object type is nothing more than an object whose object body contains the following information:

- o Type specific mutex.
- o Structure control linking all objects of the same type together.
- o Dispatcher object offset.
- o Pool type to use when allocating space for objects of this type.
- o Invalid object attribute bits.
- o Mapping vector to map generic access bits into standard and/or specific access bits.
- o Valid access bits.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Pointer to a type specific dump procedure, if any.
- o Pointer to a type specific delete procedure, if any.
- o Pointer to a type specific open procedure, if any.
- o Pointer to a type specific close procedure, if any.
- o Pointer to a type specific parse procedure, if any.
- o Pointer to a type specific security procedure, if any.

These items are used to manage type specific attributes of each object. The type specific mutex is acquired whenever an object of that type is being created, deleted, or having its security descriptor examined or modified. This prevents race conditions between object creation and deletion.

The SecurityDescriptor associated with an object type descriptor is examined for OBJECT_TYPE_CREATE access by the ObCreateObject function every time an object of the corresponding object type is created. This provides a mechanism to grant or deny the ability to create objects of a specific type on an individual identifier basis using the SecurityDescriptor associated with the object type descriptor structure.

The name is used to uniquely identify the type. All type names are stored in the \ObjectTypes object directory.

The pool type determines whether the object header and object body are allocated from paged pool or non-paged pool.

The dispatcher offset is used to implement a generic wait function. Waiting on an object waits on the offset within the object body specified by the dispatcher offset. This allows a program to wait on multiple objects of different types or a single object of unknown type, without having to know the object type.

The six type specific procedures are called whenever a type specific action must be performed from within the context of the object manager.

1.6 Object Handles

An object handle is a 32-bit opaque pointer to an object. There may be more than one handle for a given object, as a result of sharing via inheritance or naming. Associated with each handle is a pointer to the object, a granted access mask that was computed at the time the handle was created and handle attributes such as where the handle should be inherited on child process creation.

Object handles are created by inserting an object into an object table. An object table consists of an array of object table entries. An object handle is an index into an object table to the object table entry for that handle. The object table entry contains the information associated with the handle (i.e. the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

pointer to the object, the granted access mask and the handle attributes). There is an object table associated with each process. Thus handles are process specific, and meaningless outside of the context in which the handle was created. All object handles associated with a process are automatically "closed" upon that process terminating.

Each object table has a mutex associated with it. This mutex is acquired any time the object table is examined or modified.

The low order 2 bits of a 32-bit object handle are set to zero by the system when a handle is created and are ignored by all system services that accept a handle. This allows applications to encode application specific type information in the low order two bits.

In the debugging version of the system, part of each 32-bit object handle is reserved for a serial number that is also stored in the associated object table entry. When an object handle is used to reference an object, the serial number in the 32-bit handle is compared with that in the object table entry and an error is returned if they don't match. This will catch cases when an old handle is reused inadvertantly.

When creating a handle to an object, the caller may specify a *DesiredAccess* parameter. The Object Manager probes the security descriptor associated with an object with the *DesiredAccess* parameter. If all requested access bits are allowed by the security descriptor then the access is granted, and the *DesiredAccess* parameter is stored in the object table entry as the granted access mask.

Some objects may require a more sophisticated access control scheme than simply checking the bits in the security descriptor. For example, a particular kind of access to an object may be granted by being given explicit permission via the security descriptor, or by having a privilege, or by having a particular kind of access to the object's container. In order to accomodate access schemes such as these, the caller may create an AccessState structure (via SeCreateAccessState). An AccessState structure contains the desired access mask, a record of the currently granted access mask, and room for a set of privileges. The caller performs whatever kind of access checking is necessary to suit it's needs, clearing bits in the imbedded *DesiredAccess* mask as appropriate. When all of the object specific logic is complete, the structure is then passed to the object manager for whatever security processing remains.

When referencing an object via an object handle, the caller also specifies a *DesiredAccess* parameter. However, in this case, the test for access is nothing more than a bit test against the granted access mask stored in the associated object table entry. Thus object handle creation encapsulates the security check for NT OS/2. Please refer to the Local Security chapter for a description of the bits defined for *DesiredAccess*, and for a description of the AccessState structure.

1.7 Object Attributes Structure

When a handle to an object is created, the object is specified with an Object Attributes structure. The structure identifies the object by name, specifies attributes about the object and/or handle being created and specifies an optional security descriptor to associate with the created object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

typedef struct **_OBJECT_ATTRIBUTES** {
ULONG *Length*;
HANDLE *RootDirectory*;
PSTRING *ObjectName*;
ULONG *Attributes*;
PVOID *SecurityDescriptor*;
PVOID *SecurityQualityOfService*; \par} **OBJECT_ATTRIBUTES**,
***typedef**OBJECT_ATTRIBUTES

OBJECT_ATTRIBUTES Structure:

Length —Specifies the length of this structure. Must be set to sizeof(OBJECT_ATTRIBUTES).

RootDirectory —An optional handle to a directory object that specifies where to start the name lookup. If this field is specified, then the *ObjectName* field must also be specified.

If this field is not specified and the *ObjectName* field is specified, then the name lookup begins in the root directory of the object name space.

ObjectName —A pointer to an object name string. The form of the name is:

[\name...\name]\object_name

The name must begin with a leading path separator character (\) if the *RootDirectory* field is NOT specified. If the *RootDirectory* field is specified, then it must NOT begin with a leading path separator as the name is relative to that directory.

Attributes —A set of flags that control attributes about the object and the handle.

Attributes Flags:

OBJ_INHERIT —The open handle is to be inherited by child process's whenever the calling process creates a new process.

OBJ_EXCLUSIVE —The object is to be accessed exclusively by the current process. Invalid if *OBJ_INHERIT* also specified.

OBJ_PERMANENT —The object is to be created as a permanent object.

OBJ_CASE_INSENSITIVE —Indicates that the name lookup should be performed in a manner which ignores the case of *ObjectName* rather than performing an exact match search.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

OBJ_OPENIF —Return a handle to an already existing object if an object by the same name already exists. If the name does not exist, and the call is a create, then create the name.

SecurityDescriptor —An optional pointer to a security descriptor to associate with this object. See the Local Security Specification for a description of a Security Descriptor. If an object is created without a security descriptor, then access to the object will be uncontrolled.

SecurityQualityOfService —An optional pointer to the security quality of service parameters specified by the client for this communication session.

1.8 Resource Quotas and Objects

Objects are allocated from system memory, either paged or nonpaged pool. When an object is created the resource charges are specified and stored in the object's header. When a process creates a handle for an object, the resource charges stored in the object's header are levied against the process. This occurs whenever any handle is created to an object. So if process A creates an object and a handle to go with it, it gets charged quota for that object. If process A then creates process B, such that process B inherits a handle to the object, then process B is also charged quota for the same object. The same is true if process A creates a second handle to the same object.

The resource charge is removed whenever a handle is closed. The resource charge includes the space for the object header, the object body, the handle table entry, the object name, if any and the security descriptor, if any. If there is no security descriptor, then a fixed amount is charged (256 bytes) in case the process later attaches a security descriptor to the object with the **NtSetSecurityObject** system service.

1.9 Object Retention

Once an instance of an object has been created, two fields and the permanent flag contained within the object's header, control retention. The fields are named *HandleCount* and *PointerCount*.

The *HandleCount* represents the number of references to this object from various object tables. This count is incremented each time an object is inserted into an object table. It is decremented each time a handle is closed, either with **NtClose** or as a result of process termination. If this count becomes zero, a check is made to determine if an attempt should be made to delete the object's name. If the permanent flag in the object's header is false and the object has a name, then an attempt is made to delete the object's name by conditionally removing its directory entry. Conditional deletion means that the necessary mutexes are released, the directory mutex is acquired, the directory entry is located and the *HandleCount* is checked again. If the count is still zero, the object's name is deleted. This is done because the object was declared as temporary and the last handle to the object has been closed.

Once the conditional deletion of the object's name has occurred, the *PointerCount* for the object is decremented.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The *PointerCount* represents the number of pointers in existence which refer to the object. When an object is first created with the ObCreateObject function, this count is set to one to account for the reference returned to the caller. In addition, if the object has a name, the count is set to two to account for the pointer from the directory object which contains the name. This count is incremented for each object table that refers to the object.

The *PointerCount* is also updated as the object is referenced and dereferenced. When the *PointerCount* is decremented to zero, the object is deleted as there are no pointers outstanding. The *PointerCount* is never allowed to be decremented below the value of the *HandleCount*.

1.10 Exclusive Object Handles

Exclusive object handles provide a method of obtaining exclusive access to a system wide resource such as a tape drive. The semantics provided by exclusive handles cannot be provided by access protection because access protection determines who can access an object, while an exclusive handle essentially "reserves" an object.

Exclusive object handles are provided by specifying *OBJ_EXCLUSIVE* in the object attributes structure.

Exclusive object creation has the following rules:

- o Any instance of an object whose type allows exclusion, may be opened or created for exclusion provided the *HandleCount* is zero.

- o Any instance of an object which has a non-zero *HandleCount* and is not marked as exclusive cannot be opened for exclusion.

- o Any instance of an object which has a non-zero *HandleCount* and is marked as exclusive can only be opened for exclusion from the owning process. This allows the owning process to open an exclusive object multiple times.

Finally, exclusive object handles may not be inherited by other processes. This means that an error will be returned if both *OBJ_EXCLUSIVE* and *OBJ_INHERIT* are specified in the object attributes structure.

1.11 Object Name Space

The Object Manager manages the global name space for NT OS/2. This name space is used to access all named objects that are contained in the local machine environment. Some of the objects that can have names are:

directory objects

object type objects

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

symbolic link objects

semaphore and event objects

process and thread objects

section and segment objects

port objects

device objects

file system objects

file objects

The object name space is modelled after OS/2 file naming convention, where directory names in a path are separated by a backslash (\). Case insensitivity is optional whenever a name lookup is performed. Case is always preserved when a name is inserted into a directory.

During system initialization, the Object Manager creates the root directory of the object name space. The **NtCreateDirectoryObject** system service can be used to create other directories within the object name space. The **ObInsertObject** function can be used to create object names within a directory object.

The entire object name space is guarded by a single mutex. This mutex is acquired whenever an portion of the directory structure is examined or modified.

A name lookup occurs whenever a new object is being inserted or an existing object is being opened by name. The name lookup is accomplished by searching in the root directory for the first name in the path. If no matching name is found, an error is returned.

The root directory defaults to the actual root directory of the global name space. However, then specifying an object name, a root directory handle may also be specified. This is the only form of relative name lookup supported by the Object Manager.

If a matching name is found and there are more tokens left in the name string, the corresponding object header is examined. If the object is not a directory object, its corresponding object type structure is examined for a parse routine. If no parse routine exists, an error status code is returned. Otherwise, the directory mutex is released, and the parse routine is called.

The parse routine can return one of three values: STATUS_SUCCESS to indicate that the object was found, STATUS_REPARSE to indicate that a reparse should occur or an error status code to indicate that the name was not found or invalid.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The parse procedure is passed pointers to both the complete name string and the remaining portion of the name string. If the parse routine returns reparse it should deallocate the original string and allocate the new string to parse, or modify the original string.

After the Object Manager's system initialization, the object name space looks like:

```
\                           - Root Directory
\ObjectTypes                - Object Type Name Directory
\ObjectTypes\Type           - Type Object Type
\ObjectTypes\Directory      - Directory Object Type
\ObjectTypes\SymbolicLink - Symbolic Link Object Type
```

Other components of system initialization will create additional type, directory and object names within the object name space.

1.12 Preventing Deadlock

To detect deadlock, the kernel associates a level number with each mutex. If an attempt is made to acquire a mutex with a level number less than a currently owned mutex a system bugcheck occurs. Associated with the Object Management routines are three levels of mutex.

- The lowest level is the object table mutex.
- The next higher level is the directory mutex.
- The highest level is the type specific mutex.

2. Object Executive APIs

2.1 Creating Object Types

New object types can be added to the system with the **ObCreateObjectType** function:

NTSTATUS
ObCreateObjectType(
  IN PSTRING *TypeName*,
  IN POBJECT_TYPE_INITIALIZER *ObjectTypeInitializer*,
  IN PULONG *DispatcherObjectOffset* **OPTIONAL**,
  IN PSECURITY_DESCRIPTOR *SecurityDescriptor* **OPTIONAL**,
  OUT POBJECT_TYPE **ObjectType*
)

Parameters:

TypeName —A required pointer to a name string. This name must not contain the path separator character (OBJ_NAME_PATH_SEPARATOR), otherwise the STATUS_INVALID_OBJECT_NAME error status code is returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ObjectTypeInitializer —A required pointer to a structure that specifies type specific information about the new object type being created.

OBJECT_TYPE_INITIALIZER Structure:

ULONG *Length* —Specifies the size of this data structure in bytes.

ULONG *InvalidAttributes* —Specifies object attributes that are invalid for objects of this type. An attempt to specify any these attributes when creating an object of this type will result in the STATUS_INVALID_PARAMETER error status code being returned. This field may not specify any bits that are not contained in OBJ_VALID_ATTRIBUTES.

GENERIC_MAPPING *GenericMapping* —Specifies the mapping of the GENERIC_READ, GENERIC_WRITE and GENERIC_EXECUTE access rights for this object type.

ULONG *ValidAccessMask* —Specifies the valid access bits that may be specified with the *DesiredAccess* parameter when creating a handle to an object of this type. The mask is only used to remove unsupported access bits and does not cause an error if an unsupported access bit is specified. Thus specifying a *DesiredAccess* of -1 (all ones) will result in requesting a *DesiredAccess* equal to the *ValidAccessMask* for the type of object being created.

POOL_TYPE *PoolType* —Specifies the type of pool, one of NonPagedPool or PagedPool. This parameter must specify NonPagedPool if the *DispatcherObjectOffset* parameter is specified. The STATUS_INVALID_PARAMETER error status code is returned if the later condition is not met.

BOOLEAN *MaintainHandleCount* —Specifies whether a handle count data base should be maintained. If TRUE, then for each object of this type, a data base is kept that keeps track of how many handles to that object each process currently has. This allows the Open/Close object type procedures to implement special logic when the first handle to an object is created and when the last handle to an object within a process is closed. If this field is TRUE then at least one of the OpenProcedure or CloseProcedure fields must be non-NULL, otherwise the STATUS_INVALID_PARAMETER error status code is returned.

OB_DUMP_METHOD *DumpProcedure* —An optional pointer to the procedure to invoke on object dumping. This procedure is useful for the debugging version of NT OS/2 to allow a uniform way to dump the contents of an object in human readable form.

If this field is NULL, no routine is called when an object is dumped.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

OB_OPEN_METHOD *OpenProcedure* —An optional pointer to the procedure to invoke whenever a handle to an object of this type is created.

If this field is NULL, no routine is called when a handle to an object of this type is created.

OB_CLOSE_METHOD *CloseProcedure* —An optional pointer to the procedure to invoke whenever a handle to an object of this type is destroyed.

If this field is NULL, no routine is called when a handle to an object of this type is destroyed.

OB_DELETE_METHOD *DeleteProcedure* —An optional pointer to the procedure to invoke on object deletion. This procedure is responsible for deallocating any pool which was allocated by object type specific routines and performing any "cleanup" operations. When the *DeleteProcedure* returns, the object management routines deallocate the object structure, unlinks the object from its object type structure, etc.

If this field is NULL, no routine is called before deallocating the object structure.

OB_PARSE_METHOD *ParseProcedure* —An optional pointer to the parse routine for this object type. If, during name parsing, an object of this type is encountered and additional parse tokens exist, this routine is invoked.

OB_SECURITY_METHOD *SecurityProcedure* —An optional pointer to the procedure to invoke whenever the *SecurityDescriptor* associated with an object is set or queried via the **NtSetSecurityObject** and **NtQuerySecurityObject** system services. Note that another procedure (SeAssignSecurity) and not this procedure is used to insert an original security descriptor on an object.

If this field is NULL, then the SeDefaultObjectMethod will be called instead.

SecurityDescriptor —An optional pointer to a Security Descriptor. This descriptor will be attached to the type object. Any attempt to create an object of this type will require the OBJECT_TYPE_CREATE access right.

DispatcherObjectOffset —An optional pointer to the offset into the object body of a kernel dispatcher object for wait operations. If this value is not specified then an object of this type cannot be used as an argument to the **NtWaitForSingleObject** and **NtWaitForMultipleObjects** system services.

ObjectType —A pointer to a variable which receives the location of the object type structure created.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>Return Value:</u>

Status code that indicates whether or not the operation was successful.

The create object type function creates an object type structure. This function returns a pointer to the object type structure via the *ObjectType* parameter.

The *TypeName* is inserted into the \ObjectTypes object directory. If the name already exists, then this function will return an error.

This function returns one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_INVALID_PARAMETER —one of the parameters was invalid.
- STATUS_OBJECT_NAME_INVALID —the type name string contained a path separator character (OBJ_NAME_PATH_SEPARATOR).
- STATUS_NO_MEMORY —unable to allocate NonPagedPool for the object type structure.

2.2 Object Type Procedure Templates

This section describes the six different procedure types that can be associated with an object type. These procedures are called whenever certain actions are performed upon an object whose object type structure contains the addresses of these procedures.

2.2.1 Object Dump Procedure

```
VOID
typedef
(*OB_DUMP_METHOD)(
    IN PVOID Object,
    IN POB_DUMP_CONTROL DumpControl OPTIONAL
    )
```

<u>Parameters:</u>

Object —A pointer to the object's body.

DumpControl —An optional pointer to a dump control structure. This structure specifies the output stream and the detail level. If not specified then output should be sent to the standard output stream. Default detail level is 1.

<u>OB_DUMP_CONTROL Structure:</u>

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PVOID *Stream* —an opaque pointer to an output stream.

ULONG *DetailLevel* —level of detail to show, along with some modifiers. See ObDumpObject description for values.

This function is called whenever one of the ObDumpObject functions is called for an object of this type. This procedure is free to write to the output stream an ASCII representation of its contents. The content is governed by the *DetailLevel* parameter.

2.2.2 Object Open Procedure

VOID
typedef
(***OB_OPEN_METHOD**)(
    IN OB_OPEN_REASON *OpenReason*,
    IN PEPROCESS *Process*,
    IN PVOID *Object*,
    IN ACCESS_MASK *GrantedAccess*,
    IN ULONG *HandleCount* **OPTIONAL**
)

Parameters:

OpenReason —Indicates one of four specific reasons for the handle being created. These are:

OpenReason Values:

ObCreateHandle —a handle to a new object is being created via the **ObInsertObject** interface.

ObOpenHandle —a handle to an existing object is being created via the **ObInsertObject**, **ObOpenObjectByName** or the **ObOpenObjectByPointer** interface.

ObDuplicateHandle —a handle to an existing object is being created via the **NtDuplicateObject** system service.

ObInheritHandle —a handle to an existing object is being created a a result of object inheritence during process creation.

Process —Specifies a pointer to the process for which the handle has been created.

Object —Specifies a pointer to the object for which the handle has been created.

GrantedAccess —Specifies the granted access mask associated with the newly created handle.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

HandleCount —Optional parameter, that is non-zero if the *MaintainHandleCount* in the associated object type structure is TRUE. If non-zero then represents the number of handles to the specified *Object* that have been created in the object table associated with the specified *Process*. Interesting value is 1, which means this is the first handle to the specified *Object* for the specified *Process*.

This function is called whenever a handle to an object is created. The *OpenReason* parameter specifies the reason the handle is being created.

This function is called after the handle has actually been inserted in the object table for the specified process, but before the object type mutex has been released. This means that the function must not attempt to manipulate any object handles itself, as it my result in an attempt to recusively acquire the object type mutex.

2.2.3 Object Close Procedure

VOID
typedef
(***OB_CLOSE_METHOD**)(
    IN PEPROCESS *Process* **OPTIONAL**,
    IN PVOID *Object*,
    IN ACCESS_MASK *GrantedAccess*,
    IN ULONG *HandleCount*
)

Parameters:

Process —Specifies a pointer to the process for which the handle has been destroyed.

Object —Specifies a pointer to the object for which the handle is been destroyed.

GrantedAccess —Specifies the granted access mask that was associated with the destroyed handle.

HandleCount —Optional parameter, that is non-zero if the *MaintainHandleCount* in the associated object type structure is TRUE. If non-zero then represents the number of handles to the specified *Object* that have been created in the object table associated with the specified *Process*, including the handle that has just been destroyed. Interesting value is 1, which means this is the last handle to the specified *Object* for the specified *Process*.

This function is called whenever a handle to an object is destroyed.

This function is called after the handle has actually been deleted from the object table for the specified process, but before the object type mutex has been released. This means that the function must not attempt to manipulate any object handles itself, as it my result in an attempt to recusively acquire the object type mutex. Also, the object name, if any, is still valid when this function is called.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2.4 Object Delete Procedure

VOID
typedef
(***OB_DELETE_METHOD**)(
    IN PVOID *Object*
)

Parameters:

Object —A pointer to the object's body.

This function is called whenever the *PointerCount* associated with the object is decremented to zero, and the object is a temporary object. See the section on *Object* Retention for a description of how the *PointerCount* can become zero.

2.2.5 Object Parse Procedure

NTSTATUS
typedef
(***OB_PARSE_METHOD**)(
    IN PVOID *ParseObject*,
    IN POBJECT_TYPE *ObjectType*,
    IN OUT PACCESS_STATE *AccessState*,
    IN KPROCESSOR_MODE *AccessMode*,
    IN ULONG *Attributes*,
    IN OUT PSTRING *CompleteName*,
    IN OUT PSTRING *RemainingName*,
    IN OUT PVOID *Context* **OPTIONAL**,
    IN PSECURITY_QUALITY_OF_SERVICE *SecurityQos* **OPTIONAL**,
    OUT PVOID ***Object*
)

Parameters:

ParseObject —a pointer to the object, whose type contains this procedure as its *ParseProcedure*.

ObjectType —A pointer that supplies the type of object being referenced.

AccessState —A pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

Attributes —A set of flags that control the object attributes.

OBJ_CASE_INSENSITIVE —Indicates that the name lookup should be performed in a manner which ignores the case of the *ObjectName* rather than performing an exact match search.

CompleteName —A pointer to the complete path name being parsed.

RemainingName —A pointer to the portion of the complete path name that remains to be parsed.

Context —An optional pointer that is passed uninterpreted to the *ParseProcedure*. It is the same *Context* parameter that was passed to the routine that triggered the name lookup.

SecurityQos —An optional pointer to the security quality of service parameters specified by the client for this communication session.

Object —A pointer to a variable which receives the address of the object that the remaining name parsed to.

Return Value:

Status code that indicates whether or not the operation was successful.

CompleteName and *RemainingName* both point to the same string, with *RemainingName* describing a suffix of the *CompleteName*. Storage for the name string is from paged or nonpaged pool. This allows parse routines to allocate storage for a new name, copy any information necessary into the newly allocated storage, and deallocate the storage containing the previous name string. The Buffer fields in the *CompleteName* and *RemainingName* structures would then be updated to point to the newly allocated string and the *Length* fields would be updated as appropriate.

This function is called whenever an object is looked up by name. See the *Object* Name Space section for a description about how name lookup is performed.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_REPARSE —a success status code that tells the object manager to start the parse over at the beginning of the *CompleteName* string. The assumption being that the function modified the *CompleteName* string to point to a new name, such as the target of a symbolic link.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- STATUS_OBJECT_PATH_SYNTAX_BAD —if the parse failed because of an ill formed path name.
- STATUS_OBJECT_PATH_NOT_FOUND —if the parse was terminated because a path component was not found and there were characters remaining to parse.
- STATUS_OBJECT_NAME_NOT_FOUND —if the parse was terminated because a path component was not found and there were no more characters remaining to parse.
- STATUS_OBJECT_PATH_INVALID —if the parse succeeded and matched an object, but there were more characters remaining to be parsed.
- STATUS_ACCESS_DENIED —if any of the access tests involved in creating the object failed.

2.2.6 Object Security Procedure

NTSTATUS
typedef
(***OB_SECURITY_METHOD**)(
    IN PVOID *Object*,
    IN SECURITY_OPERATION_CODE *OperationCode*,
    IN PSECURITY_INFORMATION *SecurityInformation*,
    IN OUT PSECURITY_DESCRIPTOR *SecurityDescriptor*,
    IN OUT PULONG *CapturedLength*,
    IN OUT PSECURITY_DESCRIPTOR **ObjectsSecurityDescriptor*,
    IN POOL_TYPE *PoolType*,
    IN PGENERIC_MAPPING *GenericMapping*
)

Parameters:

Object —A pointer to an object

OperationCode —Indicates one of three specific operations that the method can perform.

OperationCode Values:

SetSecurityDescriptor —used to alter the security descriptor protecting an operation. The security method will take the input security descriptor and apply the portions of it specified by the *SecurityInformation* argument to the object.

QuerySecurityDescriptor —used to return to the caller a copy of the portions of object's security descriptor requested by the *SecurityInformation* argument. The information will be returned in the form of a security descriptor in the *SecurityDescriptor* buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

DeleteSecurityDescriptor —used when an instance of an object is being deleted. The method will cleanup (and delete as necessary) any storage associated with the object's security descriptor.

AssignSecurityDescriptor —used when an instance of an object is being created and security is being assigned to the object for the first time. The method will take the contents of the *SecurityDescriptor* field and assign it to the object.

SecurityInformation —Specifies which security information is being set or queried.

SecurityDescriptor —Points to buffer to either set or read the security descriptor from. This buffer will be probed and captured as necessary by this procedure. This parameter is ignored for the delete operation.

This parameter is ignored for the delete operation.

CapturedLength —For a query operation this specifies the size, in bytes, of the output security descriptor buffer and on return contains the number of bytes needed to store the complete security descriptor. If the length needed is greater than the length supplied the operation will fail. This parameter is ignored for the set and delete operations. It is expected to be point into kernel space, ie, it need not be probed and it will not change.

ObjectsSecurityDescriptor —This supplies the address of a variable pointing to the current object's security descriptor. This parameter will be used if the object's security descriptor is stored as part of the object header (this occurs as the default method). If this parameter is used then the procedure will deallocate and reallocate pool as necessary to hold the object's security descriptor. Alternate methods (e.g., the file system) will not use this parameter and instead will have the underlying file system store the descriptor (this means that system wide file object handles are not allowed).

This parameter is ignored for the assign operation.

PoolType —Specifies the type of pool to allocate for the object's security descriptor if needed. This parameter is ignored for the query and delete operations.

Return Value:

Status code that indicates whether or not the operation was successful.

Before calling this procedure the object manager will have determined that the requested action is allowed according to the granted access rights and privileges of the caller.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.3 Creating An Object

The data structures for an object are created with the ObCreateObject function:

NTSTATUS
ObCreateObject(
    IN KPROCESSOR_MODE *ProbeMode*,
    IN POBJECT_TYPE *ObjectType*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
    IN KPROCESSOR_MODE *OwnershipMode*,
    IN OUT PVOID *ParseContext* **OPTIONAL**,
    IN ULONG *ObjectBodySize*,
    IN ULONG *PagedPoolCharge*,
    IN ULONG *NonPagedPoolCharge*,
    OUT PVOID ***Object*
)

Parameters:

ProbeMode —Specifies one of UserMode or KernelMode. This is the mode used when probing the *ObjectAttributes* structure.

ObjectType —An pointer to the object type structure describing the type of object to create.

ObjectAttributes —An optional pointer to an *Object Attributes* structure. Refer to the *Object Attributes* discussion for details.

OwnershipMode —Specifies one of UserMode or KernelMode. For existing objects, this parameter is ignored.

The *OwnershipMode* controls the interpretation of the *SecurityDescriptor*. If the *OwnershipMode* is KernelMode and the object does not have a *SecurityDescriptor* then no access to the object with an *AccessMode* of UserMode is allowed. If the *OwnershipMode* is KernelMode and the *AccessMode* is KernelMode then the *SecurityDescriptor* is examined to determine access.

If the *OwnershipMode* is UserMode and the *AccessMode* is KernelMode then the access is always allowed. If the *OwnershipMode* is UserMode and the *AccessMode* is UserMode then the *SecurityDescriptor* is examined to determine access.

ParseContext —An optional pointer that is passed uninterpreted to any *ParseProcedure* that is called during the course of performing the name lookup.

ObjectBodySize —Size of the object body in bytes.

PagedPoolCharge —The number of bytes of paged pool to charge to the current process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NonPagedPoolCharge —The number of bytes of nonpaged pool to charge to the current process.

Object —A pointer to a variable which receives the address of the newly created object.

Return Value:

Status code that indicates whether or not the operation was successful.

Creating an object causes a block of storage from pool to be allocated. The size of the block is the sum of the object header size and the object body size. The object header is initialized and the *PointerCount* is set to 1 and the *HandleCount* is set to zero.

The address of the uninitialized object body is returned via the *Object* parameter. It is the responsibility of the object type specific creation routine to initialize the object body.

The *ObjectAttributes* parameter is considered unprobed and thus is probed by this function, using the mode specified in the *ProbeMode* parameter.

The *Attributes* field of the *ObjectAttributes* parameter is validated and stored in the object header.

The *RootDirectory* field of the *ObjectAttributes* parameter is captured into the object header at this time. The handle is not referenced at this time. It will be referenced when **ObInsertObject** is called to insert the object into an object table.

If specified, any string structure specified by the *ObjectName* field of the *ObjectAttributes* parameter is captured into the object header at this time. The actual buffer pointer to by the string structure is not probed at this time. Instead it is probed when **ObInsertObject** is called to insert the object into an object table.

The *SecurityDescriptor* field of the *ObjectAttributes* parameter is captured into the object header at this time. The pointer is not probed until **ObInsertObject** is called to insert the object into an object table. If for some reason the attempt to insert the object fails, **ObInsertObject** will clear the field in the object header before attempting to dereference the object.

The *SecurityQualityOfService* field of the *ObjectAttributes* parameter is captured into the object header at this time. The purpose of capturing it into the object header is to facilitate passing the QOS information to **ObInsertObject**. Rather than put a pointer to the QOS information into the *Object* header, the *SecurityQos* field is temporarily used to hold the pointer to the QOS structure. Note that in the case of an error, this field must be zero'd out before the object is freed, to prevent the pointer from being interpreted as a quantity of pool memory to be freed.

The *ParseContext* parameter is also captured into the object header for later use when **ObInsertObject** is called.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Memory for the object header and object body is allocated from the pool type specified in the object type descriptor. The amount of quota to charge is calculated. Quota includes the memory for the object header and body, plus any additional quota specified by the *PagedPoolCharge* and *NonPagedPoolCharge* parameters. The total quota to charge is remembered in the object header. This will allow the quota to be charged each time a handle is created for this object, using the either **ObOpenObjectByName** function or the **ObInsertObject** function.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_INVALID_PARAMETER —one of the parameters was invalid.
- STATUS_OBJECT_NAME_INVALID —an object name was specified in the *ObjectAttributes* structure, but it has a zero length.
- STATUS_NO_MEMORY —no memory to allocate the object.
- STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid.
- STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary.

2.4 Creating an Instance of an Object

An instance of an object is created by inserting the new created object into the calling process's object table and obtaining an object handle. This is accomplished with the **ObInsertObject** function:

NTSTATUS
ObInsertObject(
    IN PVOID *Object*,
    IN PACCESS_STATE *PassedAccessState* **OPTIONAL**,
    IN ACCESS_MASK *DesiredAccess* **OPTIONAL**,
    IN ULONG *ObjectPointerBias*,
    OUT PVOID ***NewObject* **OPTIONAL**,
    OUT PHANDLE *Handle*
)

<u>Parameters:</u>

Object —A pointer to the object's body. The object must be one that was returned by **ObCreateObject**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PassedAccessState —An optional pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

DesiredAccess —An optional parameter describing the desired types of access to the object. The interpretation of this field is object type dependent. Simple access requests (ie, those that intend to compare the desired access to the Dacl on the object) need only pass a *DesiredAccess* mask, rather than constructing an *AccessState* structure.

ObjectPointerBias —Value to increment the *PointerCount* by. This occurs whether or not the object is successfully inserted into the object table.

NewObject —An optional pointer to a variable that will will receive the pointer to the referenced object's body. A pointer to the referenced object's body is returned only if the *ObjectPointerBias* field is not zero and the argument is present. If the argument is supplied and the *ObjectPointerBias* is zero, then NULL is returned in the pointer.

Handle —A pointer to a variable that will receive the object handle value.

Return Value:

Status code that indicates whether or not the operation was successful.

Inserting the object into a table causes an object handle to be allocated from the appropriate table thereby making the object visible. If the object was given a name, the name is visible to all threads that have "read" or "execute" access to the directory path that contains the name.

The *ObjectName* field of the *ObjectAttributes* parameter to **ObCreateObject** is extracted from the object header and probed for accessiblility. Storage is then allocated for a copy of the string, so that any parse procedures called can reallocate the string for reparse operations. The *Attributes* and *ParseContext* fields that were captured into the object header are used along with the captured *ObjectName* as additional parameters to the name lookup procedure.

During the creation of a new object's instance, checks are performed to ensure that the name of the object, if any, is unique within the specified directory. If the name is not unique, the newly created object is deleted and the *OBJ_OPENIF* option is used to determine the appropriate action.

If *OBJ_OPENIF* was specified, the object instance with the collided name is examined to see if the desired access can be granted. If so, a handle is created to the collided object. If *OBJ_OPENIF* was not specified, an error status is returned to the caller.

In the process of creating or opening a named object, several different security operations may be performed. For each subdirectory in the object's path, the current subject must have TRAVERSE access to that subdirectory in order for the name search to continue. The interface to perform this test

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

is **ObCheckTraverseAccess**. **ObCheckTraverseAccess** will be called by the object manager as appropriate if the object does not have an object-specific parse routine. For those objects that do specify parse routines, it is the responsiblity of the parse routine to check traverse access to each subdirectory. **ObCheckTraverseAccess** may generate audit messages.

If the object is being created, it is necessary to check to make sure that the subject has the ability to create an object in the specified directory. Note that this is a different access type than the ability to traverse the parent directory. The interface that performs this test is **ObCheckCreateObjectAccess**. Like **ObCheckTraverseAccess**, this routine will be called by the object manager unless there exists an object-specific parse routine, in which case it is the responsibility of the parse routine to make the call.

Finally, a new handle to the object is created, and the count of outstanding handles to the object is incremented in the object header. Depending on whether the object is being created or simply opened, the parse routine must call either **ObCheckCreateInstanceAccess** or ObpCheckObjectAccess respectively.

The **ObInsertObject** function automatically dereferences the specified object, even if the operation fails for any reason. This means that the *Object* value is no longer usable when this function returns. This is due to the fact that at the completion of the **ObInsertObject** function, the object handle could now be deleted by another thread of execution causing the storage for the object to be deallocated or the name could have collided, causing the original object to be deleted.

The *ObjectPointerBias* parameter provides a mechanism for ensuring a pointer to the object can be utilized. When the *ObjectPointerBias* is not zero, the value is added to the *PointerCount* in the object header referenced by the handle. This prevents the object from being deleted. The *NewObject* parameter receives the pointer to the object body referred to by the object. This may be a different object then the one which was inserted due to name collisions.

This is typically the last operation that is performed when an instance of an object is created, and the handle and status value are returned to the caller.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_OBJECT_NAME_EXISTS —the object name already existed and *OBJ_OPENIF* was specified. This is a warning status code.
- STATUS_OBJECT_TYPE_MISMATCH —the object name already existed, but was a different type than specified by the *ObjectType* parameter.
- STATUS_OBJECT_NAME_COLLISION —the object name already existed and *OBJ_OPENIF* was not specified.
- STATUS_OBJECT_PATH_SYNTAX_BAD —if the parse failed because of an ill formed path name.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o STATUS_OBJECT_PATH_NOT_FOUND —if the parse was terminated because a path component was not found and there were characters remaining to parse.
- o STATUS_OBJECT_NAME_NOT_FOUND —if the parse was terminated because a path component was not found and there were no more characters remaining to parse.
- o STATUS_OBJECT_PATH_INVALID —if the parse succeeded and matched an object, but there were more characters remaining to be parsed.
- o STATUS_ACCESS_DENIED
- o STATUS_QUOTA_EXCEEDED
- o STATUS_NO_MEMORY

2.5 Open Object by Name

An object can be opened by name with the **ObOpenObjectByName** function:

NTSTATUS
ObOpenObjectByName(
  IN POBJECT_ATTRIBUTES *ObjectAttributes*,
  IN POBJECT_TYPE *ObjectType* **OPTIONAL**,
  IN KPROCESSOR_MODE *AccessMode*,
  IN OUT PACCESS_STATE *PassedAccessState* **OPTIONAL**,
  IN ACCESS_MASK *DesiredAccess* **OPTIONAL**,
  IN OUT PVOID *ParseContext* **OPTIONAL**,
  OUT PHANDLE *Handle*
)

Parameters:

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

ObjectType —A optional pointer to the object type structure for the object's type.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

ParseContext —An optional pointer that is passed uninterpreted to any *ParseProcedure* that is called during the course of performing the name lookup.

PassedAccessState —An optional pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

argument indicates whether any of the privileges or all of the privileges are needed to open the object.

DesiredAccess —The desired types of access to the object. The interpretation of this field is object type dependent. Simple access requests (ie, those that intend to compare the desired access to the Dacl on the object) need only pass a *DesiredAccess* mask, rather than constructing an *AccessState* structure.

Handle —A pointer to a variable that will receive the object handle.

Return Value:

Status code that indicates whether or not the operation was successful.

Opening an object by name causes a name search to be performed. If this function completes successfully, a pointer to the named object's body is inserted into the specified object table.

Successful opening of an object by name causes the *HandleCount* and *PointerCount* for the specified object to be incremented.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_OBJECT_TYPE_MISMATCH —the object name was found, but was a different type than specified by the *ObjectType* parameter.
- STATUS_OBJECT_PATH_SYNTAX_BAD —if the parse failed because of an ill formed path name.
- STATUS_OBJECT_PATH_NOT_FOUND —if the parse was terminated because a path component was not found and there were characters remaining to parse.
- STATUS_OBJECT_NAME_NOT_FOUND —if the parse was terminated because a path component was not found and there were no more characters remaining to parse.
- STATUS_OBJECT_PATH_INVALID —if the parse succeeded and matched an object, but there were more characters remaining to be parsed.
- STATUS_ACCESS_DENIED
- STATUS_QUOTA_EXCEEDED
- STATUS_NO_MEMORY

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.6 Open Object by Pointer

A handle to an object can be opened by pointer with the **ObOpenObjectByPointer** function:

NTSTATUS
ObOpenObjectByPointer(
  IN PVOID *Object*,
  IN ULONG *HandleAttributes*,
  IN PACCESS_STATE *PassedAccessState* **OPTIONAL**,
  IN ACCESS_MASK *DesiredAccess* **OPTIONAL**,
  IN POBJECT_TYPE *ObjectType* **OPTIONAL**,
  IN KPROCESSOR_MODE *AccessMode*,
  OUT PHANDLE *Handle*
)

Parameters:

Object —A pointer to the object that is being opened.

HandleAttributes —The attributes to associated with the handle. Same as the *Attributes* field in the *ObjectAttributes* structure. Refer to the *Object Attributes* discussion for details.

PassedAccessState —An optional pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

DesiredAccess —The desired types of access to the object. The interpretation of this field is object type dependent. Simple access requests (ie, those that intend to compare the desired access to the Dacl on the object) need only pass a *DesiredAccess* mask, rather than constructing an *AccessState* structure.

ObjectType —A optional pointer to the object type structure for the object's type.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

Handle —A pointer to a variable that will receive the object handle.

Return Value:

Status code that indicates whether or not the operation was successful.

Opening an object by pointer the *HandleCount* and *PointerCount* for the specified object to be incremented and a handle to the object created.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_OBJECT_TYPE_MISMATCH
- o STATUS_ACCESS_DENIED
- o STATUS_QUOTA_EXCEEDED
- o STATUS_NO_MEMORY

2.7 Referencing An Object

A user mode routine refers to an instance of an object through an object handle. In order for the executive to operate upon the object, access validation must be performed on the object handle, and the object handle must be converted to a pointer to the desired object's body. This is accomplished with the **ObReferenceObjectByHandle** function:

NTSTATUS
ObReferenceObjectByHandle(
    IN HANDLE *Handle*,
    IN ACCESS_MASK *DesiredAccess*,
    IN POBJECT_TYPE *ObjectType* **OPTIONAL**,
    IN KPROCESSOR_MODE *AccessMode*,
    OUT PVOID ***Object*,
    OUT POBJECT_HANDLE_INFORMATION *HandleInformation* **OPTIONAL**
)

Parameters:

Handle —An open handle to an object.

DesiredAccess —The desired types of access to the object. The interpretation of this field is object type dependent.

ObjectType —An optional pointer to the object type structure for the object's type. If this value is omitted, no type check is performed.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

Object —A pointer to a variable that will receive a pointer to the object's body.

HandleInformation —An optional pointer to XXXXXXXXXX

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>Return Value:</u>

Status code that indicates whether or not the operation was successful.

This function uses the specified object handle as an index into the process object table. The index is validated against the object table bounds and converted into a pointer to a specific entry in the object table.

If the *AccessMode* is KernelMode, the desired access is always allowed.

If the *AccessMode* is UserMode, the desired access is compared to the granted access field stored within the table. If all of the bits in the *DesiredAccess* mask are set in the granted access mask, then access is granted. Otherwise the STATUS_ACCESS_DENIED error status code is returned.

If the desired access is allowed, a pointer to the object header is obtained from the table. If the specified *ObjectType* is supplied, it is compared to the object type field within the object header, and if they are equal a pointer to the object body is returned to the caller as the function value, and the *PointerCount* field in the object header is incremented.

Incrementing the *PointerCount* field prevents the object from being deleted while it is being operated upon.

A pointer to the object body is retreived from the object table entry and returned to the caller via the *Object* parameter.

This function may return one of the following status codes:

- STATUS_SUCCESS ––normal, successful completion.
- STATUS_OBJECT_TYPE_MISMATCH
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE

2.8 Reference Object by Name

An object can be referenced by name with the **ObReferenceObjectByName** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
ObReferenceObjectByName(
    IN PSTRING *ObjectName*,
    IN ULONG *Attributes*,
    IN PACCESS_STATE *PassedAccessState* **OPTIONAL**,
    IN ACCESS_MASK *DesiredAccess* **OPTIONAL**,
    IN POBJECT_TYPE *ObjectType*,
    IN KPROCESSOR_MODE *AccessMode*,
    IN OUT PVOID *ParseContext* **OPTIONAL**,
    OUT PVOID ***Object***
)

Parameters:

ObjectName —A pointer to a string which specifies the name of the object to open.

Attributes —A set of flags that control the object attributes.

OBJ_CASE_INSENSITIVE —Indicates that the name lookup should be performed in a manner which ignores the case of the *ObjectName* rather than performing an exact match search.

PassedAccessState —An optional pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

DesiredAccess —The desired types of access to the object. The interpretation of this field is object type dependent. Simple access requests (ie, those that intend to compare the desired access to the Dacl on the object) need only pass a *DesiredAccess* mask, rather than constructing an *AccessState* structure.

ObjectType —A pointer to the object type structure for the object's type.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

ParseContext —An optional pointer that is passed uninterpreted to any *ParseProcedure* that is called during the course of performing the name lookup.

Object —A pointer to a variable that will receive a pointer to the object's body.

Return Value:

Status code that indicates whether or not the operation was successful.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Referencing an object by name causes a name search to be performed. If this function completes successfully, a pointer to the named object's body is returned as the function value. The name search is accomplished by acquiring the directory mutex, and searching in the root directory for the first name in the path. If no matching name is found, an error status code is returned.

If a matching name is found and there are more tokens left in the name string, the corresponding object header is examined. If the object is not a directory object, its corresponding object type structure is examined for a parse routine. If no parse routine exists, an error status code is returned. Otherwise, the directory mutex is released, and the parse routine is called.

The parse routine is responsible for either returning a pointer to an object, which can be referenced as a result of the parse, or returning a unique value, OBJ_REPARSE to indicate that the name lookup should start over from the beginning of the string.

If the value returned is OBJ_REPARSE, the directory mutex is acquired and name parsing beings using the complete string as the name. This requires the parse routine to deallocate the previous string and allocate the new string to parse, or modify the original string.

Successful referencing of an object by name causes the *PointerCount* for the specified object to be incremented.

This function may return one of the following status codes:

- o STATUS_SUCCESS ––normal, successful completion.
- o STATUS_OBJECT_TYPE_MISMATCH ––the object name was found, but was a different type than specified by the *ObjectType* parameter.
- o STATUS_OBJECT_PATH_SYNTAX_BAD ––if the parse failed because of an ill formed path name.
- o STATUS_OBJECT_PATH_NOT_FOUND ––if the parse was terminated because a path component was not found and there were characters remaining to parse.
- o STATUS_OBJECT_NAME_NOT_FOUND ––if the parse was terminated because a path component was not found and there were no more characters remaining to parse.
- o STATUS_OBJECT_PATH_INVALID ––if the parse succeeded and matched an object, but there were more characters remaining to be parsed.
- o STATUS_ACCESS_DENIED
- o STATUS_NO_MEMORY

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.9 Reference Object by Pointer

NTSTATUS
ObReferenceObjectByPointer(
    IN PVOID *Object*,
    IN ACCESS_MASK *DesiredAccess*,
    IN POBJECT_TYPE *ObjectType*,
    IN KPROCESSOR_MODE *AccessMode*
)

Parameters:

Object —A pointer to the object's body.

DesiredAccess —A mask representing the desired access to the object.

ObjectType —A pointer to the object type structure for the object.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_OBJECT_TYPE_MISMATCH

2.10 Making an Object Temporary

An object can be made temporary with the ObMakeTemporaryObject function:

VOID
ObMakeTemporaryObject(
    IN PVOID *Object*
)

Parameters:

Object —A pointer to an object.

This is a generic function and operates on any type of object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Making an object temporary causes the permanent flag of the associated object to be cleared. A temporary object has a name as long as its *HandleCount* is greater than zero. When the *HandleCount* becomes zero, the name is deleted and the *PointerCount* adjusted appropriately.

2.11 Dereferencing an Object

A referenced object is dereferenced with the ObDereferenceObject function:

VOID
ObDereferenceObject(
    IN PVOID *Object*
)

Parameters:

Object —A pointer to the object's body.

When an object is dereferenced, its *PointerCount* is decremented and retention checks are performed.

2.12 Object Management during Process Creation and Deletion

The *Process* Structure component uses these function during process creation and deletion to initialize and cleanup the object table associated with a process.

2.12.1 Process Creation Hook

The *Process* Structure component calls the *Object* Management component at process creation time via the **ObInitProcess** function.

NTSTATUS
ObInitProcess(
    PEPROCESS *ParentProcess* **OPTIONAL**,
    PEPROCESS *NewProcess*
)

Parameters:

ParentProcess —An optional pointer to the process to inherit any handles from.

NewProcess —A pointer to the process that is being created.

Return Value:

Status code that indicates whether or not the operation was successful.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This functions creates an object table for the *NewProcess*. It then scans the object table associated with the *ParentProcess*, if any, and creates copies of all handles that were created with the *OBJ_INHERIT* attribute.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_QUOTA_EXCEEDED
- o STATUS_NO_MEMORY

2.12.2 Process Deletion Hook

The *Process* Structure component calls the *Object* Management component at process deletion time via the **ObKillProcess** function.

VOID
ObKillProcess(
    PEPROCESS *Process*
)

Parameters:

Process —A pointer to the process that is being destroyed.

This function scans the object table associated with the process being destroyed and calls **NtClose** for each valid handle.

2.13 Dump Object Support

Objects are displayed using the **ObDumpObjectByHandle**, **ObDumpObjectByName** and **ObDumpObjectByPointer** functions. These functions display the contents of an object or objects to a specified output stream with a specified level of information. The default output stream is standard output.

NTSTATUS
ObDumpObjectByHandle(
    IN HANDLE *Handle*,
    IN POB_DUMP_CONTROL *DumpControl* **OPTIONAL**
)

Parameters:

Handle —An open handle to an object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

DumpControl —An optional pointer to a dump control structure. This structure specifies the output stream and the detail level. If not specified then output should be sent to the standard output stream. Default detail level is 1.

OB_DUMP_CONTROL Structure:

PVOID *Stream* —an opaque pointer to an output stream.

ULONG *DetailLevel* —level of detail to show, along with some modifiers.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE

NTSTATUS
ObDumpObjectByName(
    IN PSTRING *ObjectName*,
    IN ULONG *Attributes*,
    IN POB_DUMP_CONTROL *DumpControl* **OPTIONAL**
)

Parameters:

ObjectName —A pointer to a string which specifies the name of the object to open.

Attributes —A set of flags that control the object attributes.

OBJ_CASE_INSENSITIVE —Indicates that the name lookup should be performed in a manner which ignores the case of the *ObjectName* rather than performing an exact match search.

DumpControl —See **ObDumpObjectByHandle** description for meaning of this parameter.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED

NTSTATUS
ObDumpObjectByPointer(
    IN PVOID *Object*,
    IN POB_DUMP_CONTROL *DumpControl* **OPTIONAL**
)

Parameters:

Object —A pointer to the object's body.

DumpControl —See **ObDumpObjectByHandle** description for meaning of this parameter.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED

2.14 Check Traverse Access

A parse routine calls **ObCheckTraverseAccess** for each section of a pathname to see if the caller has Traverse access to that directory.

BOOLEAN
ObCheckTraverseAccess(
    IN PVOID *DirectoryObject*,
    IN ACCESS_MASK *TraverseAccess*,
    IN PACCESS_STATE *AccessState*,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *PreviousMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

DirectoryObject —The object header of the object being examined.

TraverseAccess —The access mask corresponding to traverse access for this directory type.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

AccessState —Checks for traverse access will typically be incidental to some other access attempt. Information on the current state of that access attempt is required so that the constituent access attempts may be associated with each other in the audit log.

TypeMutexLocked —Supplies a boolean indicating whether or not the object's type mutext is locked.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise. *AccessStatus* contains the status code to be passed back to the caller. It is not correct to simply pass back STATUS_ACCESS_DENIED, since this will have to change with the advent of mandatory access control.

This routine is to be called by *Object* parse methods as they parse the component subdirectories of a path. On each subdirectory, they must call **ObCheckTraverseAccess**, which will examine the security descriptors on the object to determine if it is legal to traverse that directory. If it returns failure, the value returned in *AccessStatus* must be propogated back to the user.

This routine will generate audit records as appropriate.

2.15 Check Create Instance access

A parse routine calls ObCheckCreateInstance to determine if the caller is allowed to create an instance of an object.

BOOLEAN
ObCheckCreateInstanceAccess(
    IN PVOID *Object*,
    IN ACCESS_MASK *CreateInstanceAccess*,
    IN PACCESS_STATE *AccessState* **OPTIONAL**,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *PreviousMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

Object —The object header of the object being examined.

CreateInstanceAccess —The access mask corresponding to create access for this object type.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

AccessState —Checks for create access will typically be incidental to some other access attempt. Information on the current state of that access attempt is required so that the constituent access attempts may be associated with each other in the audit log.

TypeMutexLocked —Indicates whether the type mutex for this object's type is locked. The type mutex is used to protect the object's security descriptor from being modified while it is being accessed.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise. *AccessStatus* contains the status code to be passed back to the caller.

Routine Description:

Parse routines must call this routine to check for Create Instance access to the object. If the attempt fails, the caller must propagate the result returned in *AccessStatus* back to the user, rather than simply returning STATUS_ACCESS_DENIED.

Note that checking for the ability to create an object of a given type is different from creating the object itself. This attempt may be audited, even if the attempt to create the object ultimately fails.

2.16 Check Create Object Access

A parse routine calls **ObCheckCreateObjectAccess** to see if it may create an object in the passed directory.

BOOLEAN
ObCheckCreateObjectAccess(
    IN PVOID *DirectoryObject*,
    IN ACCESS_MASK *CreateAccess*,
    IN PACCESS_STATE *AccessState* **OPTIONAL**,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *PreviousMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

DirectoryObject —The object header of the object being examined.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

CreateAccess —The access mask corresponding to create access for this directory type.

AccessState —Checks for traverse access will typically be incidental to some other access attempt. Information on the current state of that access attempt is required so that the constituent access attempts may be associated with each other in the audit log.

TypeMutexLocked —Indicates whether the type mutex for this object's type is locked. The type mutex is used to protect the object's security descriptor from being modified while it is being accessed.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise. *AccessStatus* contains the status code to be passed back to the caller.

Routine Description:

This routine checks to see if we are allowed to create an object in the given directory. If the attempt fails, the caller must propagate the result returned in *AccessStatus* back to the user, rather than simply returning STATUS_ACCESS_DENIED.

This routine may generate audit messages as appropriate.

2.17 Check Implicit Object Access

Check object access when there will be no handle allocated.

BOOLEAN
ObCheckImplicitObjectAccess(
    IN PVOID *Object*,
    IN OUT PACCESS_STATE *AccessState*,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *AccessMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

ObjectHeader —The object header of the object being examined.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

AccessState —The ACCESS_STATE structure containing accumulated information about the current access attempt.

TypeMutexLocked —Indicates whether the type mutex for this object's type is locked. The type mutex is used to protect the object's security descriptor from being modified while it is being accessed.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise

Routine Description:

This routine is used to perform access validation for reasons other than opening or creating an object. For example, a file system may want to determine of a subject has FILE_LIST_DIRECTORY access to a directory as part of some other access validation. For access operations on objects that are being opened or created, use ObpCheckObjectAccess.

The routine performs access validation on the passed object. The remaining desired access mask is extracted from the *AccessState* parameter and passed to the appropriate security routine to perform the access check.

Note that the RemainingDesiredAccess field in the *AccessState* parameter is not modified.

2.18 Checking Access for Object Reference

This routine is to be used to determine if a reference by name should be permitted.

BOOLEAN
ObCheckObjectReference(
    IN PVOID *Object*,
    IN OUT PACCESS_STATE *AccessState*,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *AccessMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

ObjectHeader —The object header of the object being examined.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

AccessState —The ACCESS_STATE structure containing accumulated information about the current attempt to gain access to the object.

TypeMutexLocked —Indicates whether the type mutex for this object's type is locked. The type mutex is used to protect the object's security descriptor from being modified while it is being accessed.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise

Routine Description:

The routine performs access validation on the passed object. The remaining desired access mask is extracted from the *AccessState* parameter and passes to the appropriate security routine to perform the access check.

If the access attempt is successful, SeAccessCheck returns a mask containing the granted accesses. The bits in this mask are turned on in the PreviouslyGrantedAccess field of the *AccessState*, and are turned off in the RemainingDesiredAccess field.

This routine differs from ObpCheckObjectAccess in that it calls a different audit routine.

2.19 Locking a security descriptor

Call **ObLockSecurityDescriptor** before reading or writing an object's security descriptor.

VOID
ObLockSecurityDescriptor(
    IN PVOID *Object*
)

Parameters:

Object —supplies a pointer to the object whose security descriptor is being examined.

Return Value: None.

Routine Description:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function acquires the object type mutex for the passed object, which will protect the object's security descriptor from modification by another thread.

2.20 Unlocking a security descriptor

Call **ObLockSecurityDescriptor** before reading or writing an object's security descriptor.

VOID
ObUnlockSecurityDescriptor(
    IN PVOID *Object*
)

Parameters:

Object —supplies a pointer to the object whose security descriptor is being examined.

Return Value: None.

Routine Description:

This function releases the object type mutex for the passed object, which has been protecting the object's security descriptor from modification by another thread.

2.21 Query an object's Security Descriptor field

This routine allows components outside of OB to retrieve the Security Descriptor pointer in an object's header. The contents of this pointer does not necessarily reflect the actual security descriptor attached to an object.

VOID
ObQueryObjectSecurityDescriptor(
    IN PVOID *Object*,
    OUT PSECURITY_DESCRIPTOR ***SecurityDescriptor*
)

Parameters:

Object —Supplies a pointer to the object

SecurityDescriptor —Returns the contents of the object header's *SecurityDescriptor* field, which may be NULL.

Routine Description:

Takes a pointer to an object and returns a pointer to the security descriptor contained in the header.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.22 Set an object's Security Descriptor field

This routine permits components outside of OB to set the security descriptor field in an object's header.

VOID
ObAssignObjectSecurityDescriptor(
    IN PVOID *Object*,
    IN PSECURITY_DESCRIPTOR *SecurityDescriptor*,
    IN POOL_TYPE *PoolType*
)

Parameters:

Object —Supplies a pointer to the object

SecurityDescriptor —Supplies a pointer to the security descriptor to be assigned to the object.

PoolType —Supplies the type of pool memory used to allocate the security descriptor.

Routine Description:

Takes a pointer to an object and sets the *SecurityDescriptor* field in the object's header. Performs security quota calculations and places the security quota for this object into the object's header.

2.23 Query an object's Security information

This routine will return a copy of the passed object's security descriptor, regardless of where the security descriptor is stored.

NTSTATUS
ObGetObjectSecurity(
    IN PVOID *Object*,
    OUT PSECURITY_DESCRIPTOR ***SecurityDescriptor*,
    OUT PBOOLEAN *MemoryAllocated*
)

Parameters:

Object —Supplies the object being queried.

SecurityDescriptor —Returns a pointer to the object's security descriptor.

MemoryAllocated —indicates whether we had to allocate pool memory to hold the security descriptor or not. This should be passed back into **ObReleaseObjectSecurity**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

STATUS_SUCCESS —The operation was successful. Note that the operation may be successful and still return a NULL security descriptor.

STATUS_INSUFFICIENT_RESOURCES —Insufficient memory was available to satisfy the request.

Routine Description:

Given an object, this routine will find its security descriptor. It will do this by calling the object's security method.

It is possible for an object not to have a security descriptor at all. Unnamed objects such as events that can only be referenced by a handle are an example of an object that does not have a security descriptor.

2.24 Release an object's Security information

This routine frees the memory allocated by a previous call to **ObGetObjectSecurity**.

VOID
ObReleaseObjectSecurity(
  IN PSECURITY_DESCRIPTOR *SecurityDescriptor*,
  IN BOOLEAN *MemoryAllocated*
)

Parameters:

SecurityDescriptor —Supplies a pointer to the security descriptor to be freed.

MemoryAllocated —Supplies whether or not we should free the memory pointed to by *SecurityDescriptor*.

Routine Description:

This function will free up any memory associated with a queried security descriptor.

2.25 Set Security Quota Charged for object

Each object, when it is created, is alloted a certain amount of pool memory for security information. The amount is a function of the size of the Group and Dacl information in the object's security descriptor. The sum of the sizes of these items is passed to this routine, which will calculate the amount of pool memory to charge based on that sum, and place the resultant quantity into the object's header.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
ObSetSecurityQuotaCharged(
    IN PVOID *Object*,
    IN OUT PULONG *SecurityQuotaCharged*,
    IN POOL_TYPE *PoolType*
)

Parameters:

Object —Supplies the object to be updated.

SecurityQuotaCharged —Supplies the proposed amount of quota to be charged for security information for each handle to this object. Will return the actual amount charged.

PoolType —The type of pool memory that will be allocated to hold the security information for this object.

Routine Description:

Sets the *SecurityQuotaCharged* field for the passed object. Updates the *PagedPoolCharge* or *NonPagedPoolCharge* with the new amount, depending on the value of *PoolType*.

2.26 Validate security information against quota

Any attempt to grow the security information on an object must have the resulting size checked against the maximum amount of pool memory that may be used for the object's security information.

NTSTATUS
ObValidateSecurityQuota(
    IN PVOID *Object*,
    IN ULONG *NewSize*
)

Parameters:

Object —Supplies a pointer to the object whose information is to be modified.

NewSize —Supplies the size of the proposed new security information.

Return Value:

STATUS_SUCCESS —New size is within alloted quota.

STATUS_QUOTA_EXCEEDED —The desired adjustment would have exceeded the permitted security quota for this object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Routine Description:

This routine will check to see if the new security information is larger than is allowed by the object's pre-allocated quota.

3. Object System Services

The following routines provide an interface for user mode applications to manipulate and query objects.

3.1 Create Directory Object

Directory objects are created with the **NtCreateDirectoryObject** function:

NTSTATUS
NtCreateDirectoryObject(
    OUT PHANDLE *DirectoryHandle*,
    IN ULONG *DesiredAccess*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
)

Parameters:

DirectoryHandle —A pointer to a variable that will receive the directory object handle.

DesiredAccess —The desired types of access to the directory. The following object type specific access flags can be specified in addition to the STANDARD_RIGHTS_REQUIRED flags described in the *Object Attributes* section.

DesiredAccess Flags:

DIRECTORY_QUERY —Query access to the directory is desired.

DIRECTORY_TRAVERSE —Name lookup access to the directory is desired.

DIRECTORY_CREATE_OBJECT —Name creation access to the directory is desired.

DIRECTORY_CREATE_SUBDIRECTORY —Subdirectory creation access to the directory is desired.

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

Return Value:

Status code that indicates whether or not the operation was successful.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Directory objects are an integral part of the object management functions and as such are manipulated indirectly as a result of other operations. For example, when an object is created, its name, if any, is "inserted" in a directory object and the *PointerCount* fields of both the directory object and the named object are incremented. The named object's header contains a pointer to the directory object which contains the name.

A single mutex is utilized to guard the directory structure. It must be acquired any time a directory is accessed for examination or manipulation.

The directory object's body contains the information necessary to translate an object name to a pointer to the object. Incrementing the *PointerCount* field in the directory object's header for each name in the directory prevents the directory object from being "deallocated" with outstanding names.

If a directory object is temporary and the *HandleCount* becomes zero, then an attempt is made to delete the directory object's name by conditionally removing its directory entry. Conditional deletion means that the necessary mutexes are released, the directory mutex is acquired, the directory entry which contains the directory object is located and the *HandleCount* is checked again. If the count is still zero, the directory object's name is deleted. This is done because the directory object was declared as temporary and the last handle to the object has been closed.

If the directory's name is deleted, the *PointerCount* has not yet been decremented to account for the lack of a name. Any names which still reside within the directory object are deleted. This is accomplished by acquiring the directory mutex and finding a valid name within the directory. From the valid name, the corresponding object is located and its name field and backpointer are removed, its *PointerCount* is decremented, and the permanent flag is set false. If the resulting *PointerCount* of the named object is now zero, the directory mutex is released and the object type specific delete routine is invoked.

This procedure is repeated until all valid names within the directory have been deleted, at which time the directory mutex is released, and the *PointerCount* for the directory is decremented.

Even though a directory object's name has been removed, the directory object remains until all names contained within it have been removed. This means that certain objects which had names will no longer have names once the directory object's name has been removed. This condition is detected by a NULL backpointer in the path of directory objects.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_ACCESS_DENIED
- o STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid. Or the *DirectoryHandle* pointer was invalid.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary. Or the *DirectoryHandle* pointer was not aligned on a 4 byte boundary.

3.2 Open Object Directory

NTSTATUS
NtOpenDirectoryObject(
  OUT PHANDLE *DirectoryHandle*,
  IN ACCESS_MASK *DesiredAccess*,
  IN POBJECT_ATTRIBUTES *ObjectAttributes*
)

Parameters:

DirectoryHandle —A pointer to a variable that will receive the directory object handle.

DesiredAccess —The desired types of access to the directory. The following object type specific access flags can be specified in addition to the STANDARD_RIGHTS_REQUIRED flags described in the *Object Attributes* section.

DesiredAccess Flags:

DIRECTORY_QUERY —Query access to the directory is desired.

DIRECTORY_TRAVERSE —Name lookup access to the directory is desired.

DIRECTORY_CREATE_OBJECT —Name creation access to the directory is desired.

DIRECTORY_CREATE_SUBDIRECTORY —Subdirectory creation access to the directory is desired.

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_ACCESS_DENIED

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid. Or the *DirectoryHandle* pointer was invalid.
- o STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary. Or the *DirectoryHandle* pointer was not aligned on a 4 byte boundary.

3.3 Query Object Directory

The names in a directory object can be queried using the **NtQueryDirectoryObject** function:

NTSTATUS
NtQueryDirectoryObject(
    IN HANDLE *DirectoryHandle*,
    OUT PVOID *Buffer*,
    IN ULONG *Length*,
    IN BOOLEAN *ReturnSingleEntry*,
    IN BOOLEAN *RestartScan*,
    IN OUT PULONG *Context*,
    OUT PULONG *ReturnLength* **OPTIONAL**
)

Parameters:

DirectoryHandle —handle of directory object being queried.

Buffer —pointer to where directory entries are to be returned. The format is array of structures containing the following fields:

OBJECT_DIRECTORY_INFORMATION Structure:

STRING *Name* —*Name* of an object in the directory

STRING *TypeName* —Type name of the object

The *Buffer* fields of each name string point to memory allocated at the end of the storage pointed to by the *Buffer* parameter. This the array of Directory Entries grows down and the actual characters for each string grow up and if they meet in the middle, then the operation stops and this function returns to the caller.

Length —maximum number of bytes that can be stored in the location pointed to by the *Buffer* parameter.

ReturnSingleEntry —TRUE forces the query to stop after a single entry has been returned. Otherwise the query will return as many entries as there is room for in the output buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

RestartScan —TRUE forces the query to start with the first name in the directory. Otherwise the query picks up with the next name after the last name returned by the previous call to **NtQueryDirectoryObject** for this directory object.

Context —A pointer to a context value. This value is used by this system service to remember its position within a directory object. The input value is ignored if the *RestartScan* parameter is TRUE.

ReturnLength —optional pointer to a variable that will receive the actual number of bytes stored in the location pointed to by the *Buffer* parameter.

Return Value:

Status code that indicates whether or not the operation was successful.

This function returns one or more entries from the directory object specified by the *DirectoryHandle* parameter.

This function remembers its current position across calls by storing a 32-bit number into the location pointed to by the *Context* parameter. This number is a logical index into the directory. It is not a pointer. This will prevent deletions that happen between calls from turning a *Context* value into a garbage quantity. It may become inaccurate due to insertions and deletions, but it will not bug check the system.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE

3.4 Create Symbolic Link

NTSTATUS
NtCreateSymbolicLinkObject(
    OUT PHANDLE *LinkHandle*,
    IN ULONG *DesiredAccess*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes*,
    IN PSTRING *LinkTarget*
)

Parameters:

LinkHandle —Supplies a pointer to a variable that will receive the symbolic link object handle.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

DesiredAccess —The desired types of access to the symbolic link object. The following object type specific access flags can be specified in addition to the STANDARD_RIGHTS_REQUIRED flags described in the *Object Attributes* section.

DesiredAccess Flags:

SYMBOLIC_LINK_QUERY —Query access to the symbolic link is desired.

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

LinkTarget —Supplies the target name for the symbolic link object.

Return Value:

Status code that indicates whether or not the operation was successful.

This function creates a symbolic link object, sets its initial value to value specified in the *LinkTarget* parameter, and opens a handle to the object with the specified desired access.

The symbolic link object type has a parse procedure that implements the symbolic link semantics. Basically if the parse procedure is called and if the remaining string is not null, then the remaining string value is concatenated with the target name string stored in the symbolic link object, separated by a path separator character. The result replaces the complete string and the OBJ_REPARSE is returned to trigger the reparse.

If the remaining string is null, then it assumes the caller is trying to open the symbolic link and returns a pointer to the symbolic link object body. This will fail with STATUS_OBJECT_TYPE_MISMATCH if the caller did not specify the symbolic link object type.

Otherwise the symbolic link parse procedure returns NULL to indicate an error.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid. Or the *LinkTarget*, *LinkTarget*->Buffer or the *LinkHandle* pointer were invalid.
- STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary. Or the *LinkTarget* or *LinkHandle* pointer were not aligned on a 4 byte boundary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.5 Open Symbolic Link

NTSTATUS
NtOpenSymbolicLinkObject(
    OUT PHANDLE *LinkHandle*,
    IN ULONG *DesiredAccess*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
)

Parameters:

LinkHandle —Supplies a pointer to a variable that will receive the symbolic link object handle.

DesiredAccess —The desired types of access to the symbolic link object. The following object type specific access flags can be specified in addition to the STANDARD_RIGHTS_REQUIRED flags described in the *Object Attributes* section.

DesiredAccess Flags:

SYMBOLIC_LINK_QUERY —Query access to the symbolic link is desired.

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

Return Value:

Status code that indicates whether or not the operation was successful.

This function opens a handle to a symbolic link object with the specified desired access.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid. Or the *LinkHandle* pointer was invalid.
- STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary. Or the *LinkHandle* pointer was not aligned on a 4 byte boundary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.6 Query Symbolic Link

NTSTATUS
NtQuerySymbolicLinkObject(
    IN HANDLE *LinkHandle*,
    OUT PSTRING *LinkTarget*
)

Parameters:

LinkHandle —Supplies a handle to a symbolic link object.

LinkTarget —Supplies a pointer to a record that is to receive the target name of the symbolic link object.

Return Value:

Status code that indicates whether or not the operation was successful.

This function queries the state of an symbolic link object and returns the requested information in the string pointed to by the *LinkTarget* parameter.

3.7 Wait For Single Object

A wait operation on a waitable object is accomplished with the **NtWaitForSingleObject** function:

NTSTATUS
NtWaitForSingleObject(
    IN HANDLE *Handle*,
    IN BOOLEAN *Alertable*,
    IN PTIME *TimeOut* **OPTIONAL**
)

Parameters:

Handle —An open handle to a waitable object.

Alertable —A boolean value that specifies whether the wait is alertable.

TimeOut —An optional pointer to a time-out value that specifies the absolute or relative time over which the wait is to be completed.

Return Value:

Status code that indicates whether or not the operation was successful.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Waiting on an object checks the current state of the object. If the current state of the object allows continued execution, any adjustments to the object state are made (for example, decrementing the semaphore count for a semaphore object) and the thread continues execution. If the current state of the object does not allow continued execution, the thread is placed into the wait state pending the change of the object's state or time-out.

This function requires SYNCHRONIZE access to the passed handle.

This function may return one of the following success status codes that indicates how the wait was satisfied:

- o A value of STATUS_TIME_OUT indicates that the wait was terminated due to the *TimeOut* conditions.
- o A value of STATUS_SUCCESS indicates the specified object attained a Signaled state thus completing the wait.
- o A value of STATUS_ABANDONED indicates the specified object attained a Signaled state but was abandoned.

This function may return one of the following error status codes if the wait was not satisfied:

- o STATUS_ALERTED
- o STATUS_USER_APC
- o STATUS_HANDLE_NOT_WAITABLE
- o STATUS_ACCESS_DENIED
- o STATUS_INVALID_HANDLE
- o STATUS_ACCESS_VIOLATION —The Timeout pointer was invalid.
- o STATUS_DATATYPE_MISALIGNMENT —The Timeout pointer was not aligned on a 4 byte boundary.

3.8 Wait for Multiple Objects

A wait operation on multiple waitable objects (up to MAXIMUM_WAIT_OBJECTS) is accomplished with the **NtWaitForMultipleObjects** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtWaitForMultipleObjects(
    IN ULONG *Count*,
    IN HANDLE *Handles*[],
    IN WAIT_TYPE *WaitType*,
    IN BOOLEAN *Alertable*,
    IN PTIME *TimeOut* **OPTIONAL**
)

Parameters:

Count —A count of the number of objects that are to be waited on.

Handles —An array of object handles. An error status is returned if more than one of the handles refers to the same object. This can occur even if two handle values are different but both refer to the same object.

WaitType —The type of operation that is to be performed (WaitAny or WaitAll).

Alertable —A boolean value that specifies whether the wait is alertable.

TimeOut —An optional pointer to a time-out value that specifies the absolute or relative time over which the wait is to be completed.

Return Value:

Status code that indicates whether or not the operation was successful.

This function requires SYNCHRONIZE access to the passed handle.

This function may return one of the following success status codes that indicates how the wait was satisfied:

- A value of STATUS_TIME_OUT indicates that the wait was terminated due to the *TimeOut* conditions.

- A value from 0 to MAXIMUM_WAIT_OBJECTS - 1, indicates, in the case of wait for any object, the object number which satisfied the wait. In the case of wait for all objects, the value only indicates that the wait was completed successfully.

- A value from STATUS_ABANDONED to STATUS_ABANDONED + (MAXIMUM_WAIT_OBJECTS - 1), indicates, in the case of wait for any object, the object number which satisfied the event, and that the object which satisfied the event was abandoned. In the case of wait for all objects, the value indicates that the wait was completed successfully and at least one of the objects was abandoned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function may return one of the following error status codes if the wait was not satisfied:

- o STATUS_ALERTED
- o STATUS_USER_APC
- o STATUS_INVALID_PARAMETER
- o STATUS_HANDLE_NOT_WAITABLE
- o STATUS_ACCESS_DENIED
- o STATUS_INVALID_HANDLE
- o STATUS_QUOTA_EXCEEDED
- o STATUS_NO_MEMORY
- o STATUS_INVALID_PARAMETER_MIX —One or more of the handle values in the *Handles* array referenced the same object.
- o STATUS_ACCESS_VIOLATION —The *Handles* or Timeout pointer was invalid.
- o STATUS_DATATYPE_MISALIGNMENT —The *Handles* or Timeout pointer was not aligned on a 4 byte boundary.

3.9 Duplicate Handle

A duplicate handle can be created with the **NtDuplicateObject** function:

NTSTATUS
NtDuplicateObject(
    IN HANDLE *SourceProcessHandle*,
    IN HANDLE *SourceHandle*,
    IN HANDLE *TargetProcessHandle*,
    OUT PHANDLE *TargetHandle*,
    IN ACCESS_MASK *DesiredAccess*,
    IN ULONG *HandleAttributes*,
    IN ULONG *Options*
)

Parameters:

SourceProcessHandle —An open handle to a process object or NtCurrentProcess().

SourceHandle —An open handle valid in the context of the source process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

TargetProcessHandle —An open handle to a process object or NtCurrentProcess().

TargetHandle —A pointer to a variable which receives the new handle that points to the same object as *SourceHandle* does.

DesiredAccess —The access requested to for the new handle. This access must be equal to or a proper subset of the granted access associated with the *SourceHandle*. This parameter is ignored if the DUPLICATE_SAME_ACCESS option is specified.

HandleAttributes —The attributes to associated with the new handles. Only *OBJ_INHERIT* is relevant.

Options —Specifies optional behaviors for the caller.

Options Flags:

DUPLICATE_CLOSE_SOURCE —The *SourceHandle* will be closed by this server prior to returning to the caller. This occurs regardless of any error status returned.

DUPLICATE_SAME_ACCESS —The *DesiredAccess* parameter is ignored and instead the *GrantedAccess* associated with *SourceHandle* is used as the *DesiredAccess* when creating the *TargetHandle*.

Return Value:

Status code that indicates whether or not the operation was successful.

This is a generic function and operates on any type of object.

This function requires PROCESS_DUP_ACCESS to both the *SourceProcessHandle* and the *TargetProcessHandle*.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE
- STATUS_QUOTA_EXCEEDED
- STATUS_NO_MEMORY
- STATUS_ACCESS_VIOLATION —The *TargetHandle* pointer was invalid.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o STATUS_DATATYPE_MISALIGNMENT —The *TargetHandle* pointer was not aligned on a 4 byte boundary.

3.10 Close Handle

An open handle to any object can be closed with the **NtClose** function:

NTSTATUS
NtClose(
    IN HANDLE *Handle*
)

Parameters:

Handle —An open handle to an object.

Return Value:

Status code that indicates whether or not the operation was successful.

This is a generic function and operates on any type of object.

Closing an open handle to an object causes the handle to become invalid and the *HandleCount* of the associated object to be decremented and object retention checks to be performed.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_INVALID_HANDLE

3.11 Making an Object Temporary

An object can be made temporary with the **NtMakeTemporaryObject** function:

NTSTATUS
NtMakeTemporaryObject(
    IN HANDLE *Handle*
)

Parameters:

Handle —An open handle to an object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

Status code that indicates whether or not the operation was successful.

This is a generic function and operates on any type of object.

Making an object temporary causes the permanent flag of the associated object to be cleared. A temporary object has a name as long as its *HandleCount* is greater than zero. When the *HandleCount* becomes zero, the name is deleted and the *PointerCount* adjusted appropriately.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_ACCESS_DENIED
- o STATUS_INVALID_HANDLE

3.12 Query Object

Information about an opened object can be obtained with the **NtQueryObject** function:

NTSTATUS
NtQueryObject(
    IN HANDLE *Handle*,
    IN OBJECT_INFORMATION_CLASS *ObjectInformationClass*,
    OUT PVOID *ObjectInformation*,
    IN ULONG *Length*,
    OUT ULONG **ReturnLength* **OPTIONAL**
)

Parameters:

Handle —Specifies the object that information is being requested from.

ObjectInformationClass —Specifies the type of information to retrieve from the specified object.

ObjectInformationClass Values:

ObjectBasicInformation —Returns the basic information about the specified object.

ObjectNameInformation —Returns the complete path name of the object referred to by the *Object*.

ObjectTypeInformation —Returns the name of the object type associated with the object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ObjectInformation —A pointer to a buffer which receives the specified information. The format and content of the buffer depend on the specified object information class.

ObjectInformation Format by Information Class:

ObjectBasicInformation —Data type is POBJECT_BASIC_INFORMATION

OBJECT_BASIC_INFORMATION Structure:

ULONG *Attributes* —The attributes associated with this object. Only *OBJ_INHERIT*, *OBJ_PERMANENT* and *OBJ_EXCLUSIVE* are relevant after an object handle has been created.

ACCESS_MASK *GrantedAccess* —The access mask bits that were granted to the current process with the passed handle.

ULONG *PagedPoolCharge* —How much PagedPool is charged against a process when it creates a handle to this object.

ULONG *NonPagedPoolCharge* —How much NonPagedPool is charged against a process when it creates a handle to this object.

ULONG *NameInfoSize* —The size needed to store a copy of the name associated with this object. Zero if no name.

ULONG *TypeInfoSize* —The size needed to store a copy of the type name associated with this object.

ULONG *SecurityDescriptorSize* —The size needed to store a copy of the *SecurityDescriptor* associated with this object. See the **NtQuerySecurityObject** for a description of how to get the actual copy of the security descriptor.

ObjectNameInformation —Data type is POBJECT_NAME_INFORMATION

OBJECT_NAME_INFORMATION Structure:

STRING *Name* —The name associated with this object, if any.

ObjectTypeInformation —Data type is POBJECT_TYPE_INFORMATION

OBJECT_TYPE_INFORMATION Structure:

STRING *TypeName* —The name of the object type associated with this object.

Length —Specifies the length in bytes of the *ObjectInformation* buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ReturnLength —An optional parameter that receives the number of bytes placed in the *ObjectInformation* buffer.

Return Value:

Status code that indicates whether or not the operation was successful.

This function requires READ_CONTROL access to the passed handle.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_INVALID_INFO_CLASS —The *ObjectInformationClass* parameter did not specify a valid value.
- STATUS_INFO_LENGTH_MISMATCH —The value of the ObjectInformationLength parameter did not match the length required for the information class requested by the *ObjectInformationClass* parameter.
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE

3.13 Set Security Descriptor for an Object

The function **NtSetSecurityObject** takes a well formed Security Descriptor provided by the caller and assigns specified portions of it to an object. Based on the flags set in the Security Information parameter and the caller's access rights, this procedure will replace any or all of the security information associated with an object.

This is the only function available to users and applications for changing security information, including the owner ID, group ID, and the discretionary and system ACLs of an object. The caller must have WRITE_OWNER access to the object to change the owner or primary group of the object. The caller must have WRITE_DAC access to the object to change the discretionary ACL. The caller must have the "SeSecurityPrivilege" privilege to assign a system ACL to an object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtSetSecurityObject(
    IN HANDLE *Handle*,
    IN SECURITY_INFORMATION *SecurityInformation*,
    IN PSECURITY_DESCRIPTOR *SecurityDescriptor*
)

Parameters:

Handle —A handle to an existing object.

SecurityInformation —Indicates which security information is to be applied to the object. The value(s) to be assigned are passed in the *SecurityDescriptor* parameter.

The security information is specified using the following boolean flag fields:

SecurityInformation.Owner (Object's Owner SID) *SecurityInformation*.Group (Object's Group SID) *SecurityInformation*.Dacl (Object's Discretionary ACL) *SecurityInformation*.Sacl (Object's System ACL)

SecurityDescriptor —A pointer to a well formed Security Descriptor.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_PRIVILEGE_NOT_HELD
- STATUS_INVALID_HANDLE

3.14 Query Security Descriptor for an Object

The function **NtQuerySecurityObject** returns to the caller requested security information currently assigned to an object.

Based on the caller's access rights and privileges this procedure will return a security descriptor containing any or all of the object's owner ID, group ID, discretionary ACL or system ACL. To read the owner ID, group ID, or the discretionary ACL the caller must be granted READ_CONTROL access to the object. To read the system ACL the caller must have "SeSecurityPrivilege" privilege.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtQuerySecurityObject(
    IN HANDLE *Handle*,
    IN SECURITY_INFORMATION *SecurityInformation*,
    OUT PSECURITY_DESCRIPTOR *SecurityDescriptor*,
    IN ULONG *Length*,
    OUT PULONG *LengthNeeded*
)

Parameters:

Handle —A handle to an existing object.

SecurityInformation —Supplies a value describing which pieces of security information are being queried. The values that may be specified are the same as those defined in the **NtSetSecurityObject** API section.

SecurityDescriptor —A pointer to the buffer to receive a copy of the requested security information. This information is returned in the form of a security descriptor.

Length —The size, in bytes, of the Security Descriptor buffer.

LengthNeeded —A pointer to the variable to receive the number of bytes needed to store the complete security descriptor. If *LengthNeeded* is less than or equal to *Length* then the entire security descriptor is returned in the output buffer, otherwise none of the descriptor is returned.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_BUFFER_TOO_SMALL —The value of the *Length* parameter did not specify enough memory for the requested information. The *LengthNeeded* variable will be filled in with the amount of memory needed.
- STATUS_ACCESS_DENIED
- STATUS_PRIVILEGE_NOT_HELD
- STATUS_INVALID_HANDLE

x

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Opportunistic Locking Design Note

Authors: *Darryl E. Havens, Chuck Lenzmeier and Brian Andrew*

Revision 0.4, June 12, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction 1
2. Background 1
   2.1. What is an Oplock 1
   2.2. Current LAN Manager Product Features 1
   2.3. Future LAN Manager Product Features 3
3. NT OS/2 Overview 8
4. NT OS/2 Oplock Implementation 10
   4.1. Obtaining an Oplock 10
   4.2. Opening an Oplocked File 11
   4.3. Accessing an Oplocked File 12
   4.4. Releasing an Oplock 13
5. Design Issues 14
   5.1. Timeouts 14
   5.2. Batch Oplocks 14
6. Revision History 16

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This design note describes the current implementation of *oplocks* in the LAN Manager product for Microsoft, its future plans, and the design and implementation of the support for this feature in the **NT OS/2** product.

2. Background

This section describes what an oplock is, its purpose, and the types of oplocks that exist today. Also explained are the features that are being planned for future LAN Manager products.

2.1. What is an Oplock

An oplock is an opportunistic lock. It gives client machines the ability to assume certain information about files that it has open on remote machines for the purposes of buffering information on the client machine. Because information can be buffered on the local client, the amount of network traffic is reduced. That is, the client does not have to write information into a file on the remote server if it knows that no other process is accessing the file because, by definition, no one else needs to see the data.

Likewise, the client can buffer readahead data from the file because, by definition, no one else can change the data that has been read.

Once a file is no longer locked by the client, due to someone else opening the file, for example, readahead data must be flushed and any write data or locks must be applied to the file. This keeps the file in a consistent state. This is referred to as *breaking* the oplock.

2.2. Current LAN Manager Product Features

The current LAN Manager product provides two different types of oplocks:

- *Exclusive oplocks* —This type of locking allows a client to open a file for exclusive access.
- *Batch oplocks* —This type of locking allows a client to keep a file open on the server even though the local accessor on the client machine has closed the file.

Exclusive oplocks are used to buffer lock information, readahead data, and write data on a client machine because the client knows that it is the only accessor to a file on a remote node. The basic protocol is that the redirector on the client opens the file on the remote node requesting that an oplock be given to the client. If the file is open by anyone else, then the client is refused the oplock and no local buffering may be performed on the local client. Notice that this also means that no readahead may be performed to the file, unless the redirector knows that a particular range of the file is locked by the client. Today, no readahead buffering is performed on locked ranges either.

If the client is the only accessor of the file, then the server grants the client an oplock on the file. This informs the client redirector that it is the file's only accessor. This means that the client can perform

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

certain optimizations for the file such as buffering lock and read/write data. This potentially greatly reduces the amount of network traffic between the client and the server.

Batch oplocks are designed to be used where common programs on a client behave in such a way that causes the amount of network traffic on a wire to go beyond an acceptable level for the functionality provided by the program.

For example, the command processor today executes commands from within a command procedure by performing the following steps:

- Opening the command procedure.
- Seeking to the "next" line in the file.
- Reading the line from the file.
- Closing the file.
- Executing the command.

This process is repeated for each command that is to be executed from the command procedure file. As is obvious, this type of programming model causes an inordinate amount of processing of files, thereby creating a lot of network traffic that could otherwise be curtailed if the program were to simply open the file, read a line, execute the command, and then read the next line.

Batch oplocking is designed to curtail the amount of network traffic by opening the command procedure file with an oplock. By having an oplock on the file, the local client redirector can simply skip the extraneous open and close requests. This is done by keeping the file open once it has been opened. When the command processor then asks for the next line in the file, the redirector can either ask for the next line from the server, or it may have already read the data from the file as readahead data. In either case, the amount of network traffic from the client is greatly reduced.

Once the server receives either a rename or a delete request for the file that is oplocked, it must inform the client that the oplock is to be broken if the client redirector's caller actually believes that the file has been closed. This keeps the semantics of the view of the system consistent with what would normally happen where the client redirector had actually closed the file each time its caller closed it.

2.3. Future LAN Manager Product Features

Future LAN Manager products will support several different types of oplocking. The five different types that have been seriously proposed are described in this section. Of these five, the first four have been agreed upon as the set that will be implemented in the future. Whatever design the **NT OS/2** system uses, however, must take into account the desire to perhaps one day implement all of the following types of oplocks.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

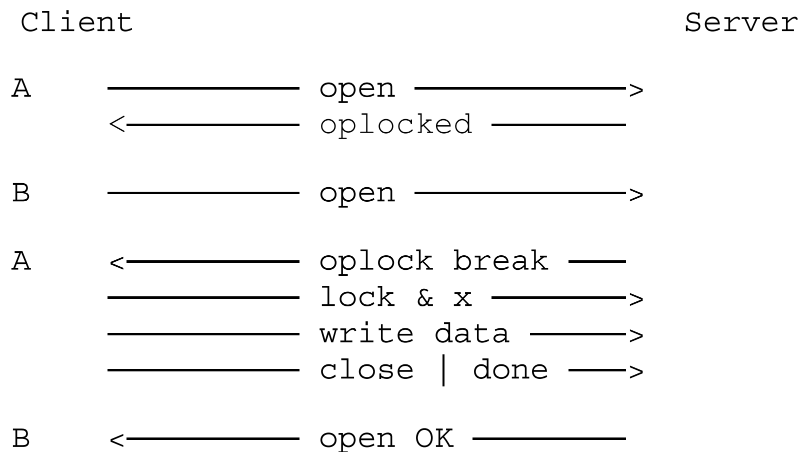
Each of the proposed oplocks is either the same, or builds on, those features currently in the LAN Manager product. These features are designed to further curtail the amount of network traffic on the wire for common situations.

The types of oplocks are:

Exclusive - The same exclusive oplocks that are part of LANMAN today.
Batch - The same batch-mode oplocks that are part of LANMAN today.
Level II - Level II oplocks allow multiple readers to a file.
Restoring - This feature allows broken oplocks to be restored.
Distributed - This feature allows distributed oplocks in the network.

2.3.1. Exclusive Oplocks

The exclusive oplocks proposed for future LAN Manager products is the same functionality that is in the current product. The protocol, in picture format, appears as follows:



As can be seen, when client A opens the file, it can request an oplock. Provided no one else has the file open on the server, then the oplock is granted to client A.

If, at some point in the future, another client, such as client B, requests an open to the same file, then the server must have client A break its oplock. Breaking the oplock involves client A sending the server any lock or write data that it has buffered, and then letting the server know that it has acknowledged that the oplock has been broken. This synchronization message informs the server that it is now permissible to allow client B to complete its open.

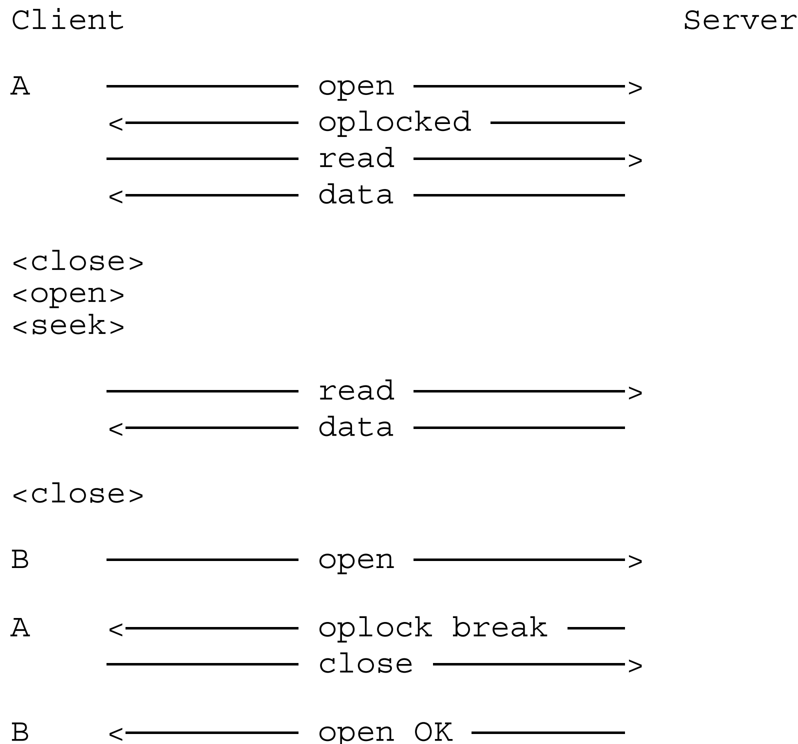
It should be noted that client A must also purge any readahead buffers that it has for the file. This is not shown in the above diagram since no network traffic is needed to do this. Future products may wish to continue to buffer any readahead data that client A knows is locked in the file for its caller.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

It is also possible for client A to complete the oplock break synchronization sequence with a close operation rather than a done. This simply short-circuits the logic in the server to allow it to optimize client B's open request and give it an oplock, provided that client B requested one.

2.3.2. Batch Oplocks

The batch oplock feature proposed for future LAN Manager products is very close to the functionality that is in the current product. The protocol, in picture format, appears as follows:



As can be seen, when client A opens the file, it can request an oplock. Provided no one else has the file open on the server, then the oplock is granted to client A.

Client A, in this case, keeps the file open for its caller across multiple open/close operations. Data may be read ahead for the caller and other optimizations, such as buffering locks, can also be performed.

When another client requests an open, rename, or delete operation to the server for the file, however, client A must cleanup its buffered data and synchronize with the server. Most of the time this involves actually closing the file, provided that client A's caller actually believes that he has closed the file. Once the file is actually closed, client B's open request can be completed.

The difference between this functionality and the feature as it is currently implemented is that in the future client A will be able to specify that it would like to have client B's open request fail. That is, client A would not drop the oplock to the file, so client B's operation should not be allowed to

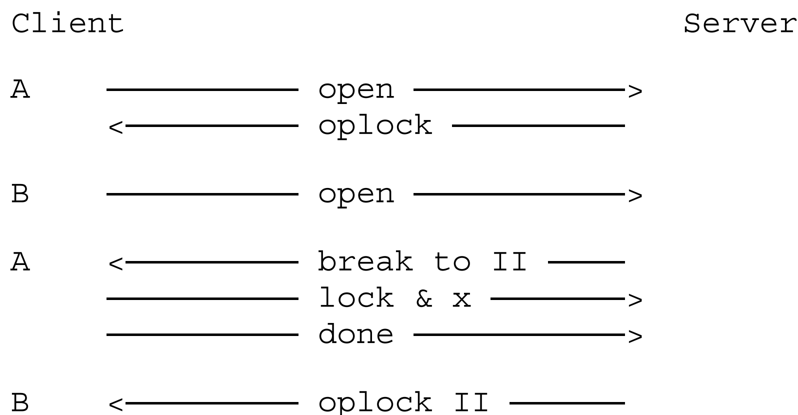
Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

continue. In some senses, this is exactly as if client A had opened the file exclusively or in some mode that is incompatible with client B's request.

2.3.3. Level II Oplocks

A level II oplock is a new feature being proposed for future LAN Manager products. This feature allows multiple clients to have the same file open, providing that no client is performing write operations to the file. This is important for many environments because most compatibility mode opens from down-level clients map to an open request for shared read/write access to the file. While it makes sense to do this, it also tends to break oplocks for other clients even though neither machine actually intends to write to the file.

The protocol, in picture format, appears as follows:



It should be noted that this sequence of events is very much like an exclusive oplock. The basic difference is that the server informs the client that it should break to a level II lock when no one has been writing the file. That is, client A, for example, may have opened the file for a desired access of READ, and a share access of READ/WRITE. This means, by definition, that client A has not performed any writes to the file.

When client B opens the file, the server must synchronize with client A in case client A has any buffered locks. Once it is synchronized, client B's open request may be completed. Client B, however, is informed that he has a level II oplock, rather than an exclusive oplock to the file.

In this case, no client that has the file open with a level II oplock may buffer any lock information on the local client machine. This allows the server to guarantee that if any write operation is performed, it need only notify the level II clients that the lock should be broken without having to synchronize all of the accessors of the file.

> \\ *It would seem that a truly correct implementation of level II oplocks would require the oplock to be broken whenever anyone took out a byte-range lock. This would prevent clients from satisfying reads from previously obtained readahead data that may currently*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

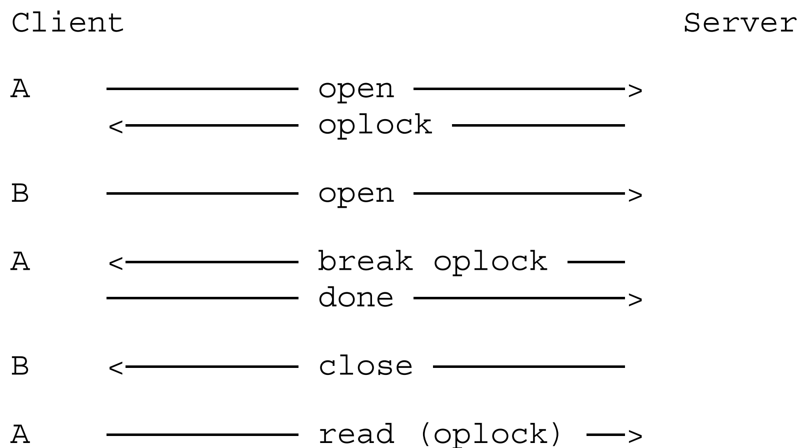
be locked. Perhaps the best approach here is heuristic that allows level II oplocks to be retained in the face of locks until the first write. \\

The level II oplock may be *broken to none*, meaning that some client that had the file opened has now performed a write operation to the file. Because no level II client may buffer lock information, the server is in a consistent state. The writing client, for example, could not have written to a locked range, by definition. Read ahead data may be buffered in the client machines, however, thereby cutting down on the amount of network traffic required to the file. Once the level II oplock is broken, however, the buffering client must flush its buffers and degrade to performing all operations on the file across the network.

2.3.4. Restoring Oplocks

Restoring oplocks to a file once they have been broken is a feature being proposed for future LAN Manager products. This feature allows an oplock to be reenabled to a file once it has been broken. To cut down on the amount of network traffic required, this request is piggy-backed on top of other requests that are normally being sent to the server. It will most likely be implemented in the SMB protocol as a simple flag in the SMB header.

The protocol, in picture format, appears as follows:



The protocol for requesting that an oplock be taken out for a file is exactly the same as it is for the previous cases. In the case of restoring oplocks, however, once the oplock has been broken, the client, in this case client A, can request that the oplock be restored. Once client B closes the file, then the oplock can actually be restored to client A by the server.

In the above example, then, should the server determine that the conditions are right to restore client A's oplock, it can simply grant the oplock by sending client A an "oplock gained" message, just as it did when client A opened the file.

Note, as well, that if client A closes the file, and client B attempts to take out an oplock on the file, then the server can choose to give the oplock to client B.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.3.5. Distributed Oplocks

Distributed oplocks refer to the ability to have multiple clients accessing the same file, but giving an oplock to the "active" client. For example, in an airline reservation system, many reservation terminals open the central database file first thing in the morning. However, once the file is opened, the terminals are rarely used. This means that each time a terminal operator wishes to make a transaction, he must assume that someone else is accessing the file as well.

> \\ *The idea of distributed oplocks has been examined by the various individuals in both the LAN and NT groups. To date, no plans have been agreed upon to implement this type of oplocking. It is included herein to give the reader a broad picture of all of the possible oplocks that have been considered.*\\

If a distributed oplocking feature were added to the system, then the oplock could be given to the "active" terminal, if no other terminals were active and hadn't been for some period of time. (Of course, selecting the right "inactive" time is an issue.) This means that if all of the terminals were idle, and one operator started a transaction, that terminal would be given an oplock. It would own this oplock until it went inactive or until some other terminal attempted to access the database at the same time.

This would reduce the network traffic and the processing involved on the oplocked client machine because it would appear to the client as if it was the only accessor of the database file.

3. NT OS/2 Overview

The **NT OS/2** product provides users of the system with the ability to perform oplocking on either a remote node or on the local machine. That is, there are no "back doors" or hidden hacks in the system that are needed to support the oplocking functionality. All users can oplock files. Implementing this functionality in this manner is consistent with the design of the entire system. This also implies that no special code need be added to allow either redirectors or network servers to provide this functionality.

Oplocking functionality in **NT OS/2** is provided through the use of the **NtFsControlFile** system service. This service allows a user to pass file system specific requests to the file system that is servicing the file represented by the user's handle to it. The requests used to oplock a file are standard, non-privileged requests.

Because the **NT OS/2** I/O system is asynchronous by nature, the ability to make a request and then have it completed at a later time makes it natural for implementing oplocks. Further, because synchronization is required by the file system to determine when the caller has completed its oplock update transfers, the file system can use this feature to block open requests to a file by queueing the I/O Request Packet **(IRP)** to its internal file control structure until the oplock owner lets it know that it is finished.

The user requests an oplock by submitting a request to the file system. If the return status from the system service is failure, then the oplock is not owned. If, on the other hand, the service return status is *STATUS_PENDING*, then the oplock is owned until the I/O operation is completed. At that time,

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

the oplock may have been broken completely, or just to another level. This is indicated by the contents of the *Information* field of the I/O status block.

If synchronization is required by the file system because an oplock was broken to a level which requires this, then the user must flush his buffers, locks, etc. to the file system and then submit another I/O request that specifies that the operation that caused the oplock to break may now be continued.

Given this simple, straightforward design, all of the oplock types can be implemented. Note that because the user actually asks for the oplock after the file is open, rather than at the time the file is opened, oplock restoring falls out.

Implementing a redirector using this design is also straightforward. The redirector always requests an oplock when it attempts to perform an open operation on a remote file. It must also remember whether or not it has gotten the oplock (so it can perform the appropriate local buffering, etc.) When the local user asks for an oplock to the file, the redirector simply completes the request accordingly.

Implementing a server is done in much the same way. If the remote redirector requests an oplock to the file, then the server opens the file and requests an oplock. It then relays whether or not it gained the oplock to the remote redirector.

When it receives a request to open a file that is oplocked, the file system blocks the open request and begins the process of breaking the oplock. Because opening a file in **NT OS/2** is a synchronous operation, this means that the opener's thread is blocked while the oplock is broken. Openers can avoid having their thread blocked by specifying an option on their call to **NtCreateFile** or **NtOpenFile**. If this option is specified, and the file is oplocked, the file system starts the oplock break, then immediately releases the opener's thread, specifying the status code for the open operation and the opener receives a handle to the file. A distinguished success code of ***STATUS_OPLOCK_BREAK_IN_PROGRESS*** is used in this case. When this handle is used to access the file, the operation will block or return ***STATUS_PENDING*** if the requested operation cannot complete immediately due to the state of the oplock.

4. NT OS/2 Oplock Implementation

The **NT OS/2** I/O system provides users with the ability to oplock files by using three FS control functions to the file system servicing the open file, in addition to a file open option.

4.1. Obtaining an Oplock

All oplocks are requested on the open file by invoking the **NtFsControlFile** service with the handle to the open file and one of the following request codes. For more information on the **NtFsControlFile** system service, see the *NT OS/2 I/O System Specification*.

- *Request level I oplock*. ––This function requests that an exclusive oplock to the file be granted. This type of request is consistent with the *exclusive* oplock discussed in previous sections. If the I/O request service status is an error, then the oplock was not granted. Otherwise, the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

request was granted and is held by the requestor until the file is closed or the I/O request completes later, indicating that the oplock has been broken to a level II oplock or no oplock.

The file system control code for this function is ***FSCTL_REQUEST_OPLOCK_LEVEL_1***. The input and output buffers are not used. If the oplock was granted, then when the I/O request completes, the *Information* field of the I/O status block indicates whether the oplock has been broken to level II (*FILE_OPLOCK_BROKEN_TO_LEVEL_2*) or to none (*FILE_OPLOCK_BROKEN_TO_NONE*).

- o *Request level II oplock*. —This function requests that a *Level II* oplock to the file be granted. Again, if the I/O request service status is an error, then the oplock was not granted. Otherwise, the request was granted and is held by the requestor until the file is closed or the I/O request completes at a later date. If the latter occurs, then the oplock has been broken to none.

  The file system control code for this function is ***FSCTL_REQUEST_OPLOCK_LEVEL_2***. The input and output buffers are not used.

- o *Request batch oplock*. —This function requests that a *Batch Oplock* to the file be granted. The semantics of a batch oplock are the same as for level I oplocks except that a subsequent open of the file will initiate the oplock break before the access sharing check is made. Unless specified, all references to level I oplocks will also refer to batch oplocks.

  The file system control code for this function is *FSCTRL_REQUEST_BATCH_OPLOCK*. If the oplock was granted, then when the I/O request completes, the *Information* field of the I/O status block indicates whether the oplock has been broken to level II (*FILE_OPLOCK_BROKEN_TO_LEVEL_2*) or to none (*FILE_OPLOCK_BROKEN_TO_NONE*).

If the **NtFsControlFile** service (not the I/O request) completes with a status other than *STATUS_PENDING*, then the oplock was not granted. If the status is *STATUS_PENDING*, then the caller owns an oplock on the file. In this case, the I/O request does not complete unless and until the oplock is broken.

If a level I oplock is requested, but the file is already oplocked (at any level), the request is rejected. If a level II oplock is requested, but the file is already oplocked at level I, the request is rejected. If a level II oplock is request on a file that is already oplocked at level II, the request is accepted.

If the owner of a level I oplock requests a level II oplock, the request is rejected.

If the owner of a level II oplock requests a level I oplock and no one else has the file open, then the level II oplock will be broken and the level I oplock will be granted. The level II oplock is broken by completing the Irp used in granting the level II oplock.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2. Opening an Oplocked File

When an oplocked file is opened again, the file system initiates an oplock break. This involves completing the FS control request(s) that created the oplock. The file system normally blocks the second open request (and subsequent opens) while the oplock is being broken.

For a level I oplock, the oplock is not considered broken until the owner of the lock issues an Accept Oplock Break FS control (see next section). This allows the owner to flush writebehind data and byte-range locks to the file before releasing the oplock.

For a level II oplock, the oplock is considered broken immediately; the owner has no writebehind data or locks to flush, so there is no need to wait for acknowledgement. The owner need not issue an Accept Oplock Break request, but the file system should not consider it an error if one is issued.

An opener can avoid having its thread blocked while waiting for a level I oplock to be broken by specifying the *FILE_COMPLETE_IF_OPLOCKED* option on the call to **NtCreateFile** or **NtOpenFile**. If this option is specified, and the file is oplocked at level I, the file system starts the oplock break, then immediately releases the opener's thread by completing the I/O request. The status of the I/O is *STATUS_OPLOCK_BREAK_IN_PROGRESS*, and the *Information* field of the I/O status block is determined by the result of the open, disregarding the oplock state of the file. The opener receives a handle to the file and may access the file using this handle. If the file is accessed and the operation cannot complete until the oplock break is completed, then the calling thread will block or ***STATUS_PENDING*** will be returned based on whether the operation is synchronous or asynchronous.

Batch oplocks present a problem in that a remote user may have opened a file with restricted share access (read-only). A batch oplock is obtained on this open file. The remote user may call to close the file, but the redirector holds the file open in anticipation of future open and read calls. When the remote user attempts to reopen the file with more liberal access (read-write), the open will fail unless the redirector acknowledges the oplock break and closes the file. The share access check for any subsequent opens of a file with a batch oplock will be blocked until the owner of the oplock has acknowledged the oplock break and closed the file (if it intends to do so). This subsequent open may elect not to block by specifying *FILE_COMPLETE_IF_OPLOCKED*, but this may cause the open to fail due to performing the share access check prematurely.

4.3. Accessing an Oplocked File

If an opener specifies that the open operation is not to block due to the oplock state of a file, then it is possible that the file may be accessed via a handle created in this way prior to the completion of the oplock break. This has no effect on files with a level 2 oplock, as they are broken immediately. Breaking a level 1 oplock requires an acknowledgement from the owner of the oplock after locally buffered data and lock requests are flushed. An operation requested on the file during the oplock break operation will be blocked pending the completion of the oplock break. If the requested operation is a synchronous operation, the thread will block pending the completion of the oplock break. Otherwise

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

STATUS_PENDING is returned and the IRP is completed when the oplock break acknowledgement is received.

The above conditions apply to the following operations:

- NtReadFile -- All read operations on the file will be blocked.
- NtLockFile -- All byte range lock requests on the file will be blocked.
- NtUnlockFile -- All byte range unlock requests on the file will be blocked.
- NtQueryInformationFile -- Any query operation involving the following file information classes will be blocked: ***FileBasicInformation, FileStandardInformation*** or ***FileAllInformation.***
- NtSetInformationFile -- Any set informtion operation involving the following file information classes will be blocked: ***FileBasicInformation, FileAllocationInformation*** or ***FileEndOfFileInformation.***
- NtWriteFile -- All write operations on the file will be blocked.

4.4. Releasing an Oplock

In response to the breaking of a level I oplock, the owner of the lock must flush any pending write behind data and lock requests. The owner then issues the following **NtFsControlFile** request:

- *Accept oplock break*. —This function is used to synchronize with the file system once an oplock has been broken. When this request is issued, the file system restarts any pending open requests and any operations blocked pending the completion of the oplock break. The file system control code for this function is ***FSCTL_OPLOCK_BREAK_ACKNOWLEDGE***.

  If the level 1 oplock is being broken to level 2, then the IRP used to acknowledge the oplock break is treated as a request for a level 2 oplock. If the level 2 oplock can be granted at this time, then ***STATUS_PENDING*** is returned. Any other return code indicates that the level 2 oplock is not granted. The IRP will be completed when the level 2 oplock is later broken.

- *Releasing a Batch Oplock*. —When a batch oplock break is initiated, the original Irp is is completed with a status indicating whether the oplock is being broken to none or to level II. The owner of the oplock will need to update the file with any locally buffered changes and then acknowledge the oplock break. The file system control code in this case is *FSCTL_OPLOCK_BREAK_ACKNOWLEDGE.*

  In many cases the oplock break acknowledgement is followed immediately by a close call. In this case, it is desirable to hold off any opens of the file until the close is performed. If the oplock break is acknowledged with the file system control code

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FSCTL_OPBATCH_ACK_CLOSE_PENDING, then the in-process open and any subsequent opens will be blocked until remote close operation has been completed.

An oplock break may be initiated by the owner of either a level 1 or level 2 oplock by calling **NtClose** on the handle used to request the oplock. In the case of a level 1 oplock, the close operation also serves as the acknowledgement of the oplock break. The IRPs for the oplocks are completed and any operations pending due to the oplock state are continued. NOTE -- This action occurs in the "Cleanup" operation in the file systems, not the "Close" operation.

5. Design Issues

This section presents the issues that need to be resolved before the oplock design herein can be fully implemented.

5.1. Timeouts

In the LAN Manager product today, timeouts are implemented on the server once it has broken an oplock for a client redirector. The redirector has 45 seconds (or so) to cleanup its buffered data, flush its locks, etc., and then submit the packet that specifies that it is okay to continue. If this doesn't occur within the specified timeout period, then the redirector's session is closed.

The issue here is two-fold:

1. Should the local machine attempt to timeout a user in the same manner, and if so, what should the timeout value be and how does the file system efficiently implement this?

2. If the local file system times out users, what should the action be? In the remote case the redirector's session is lost. This is unfortunate because this means that the file is left in an inconsistent state. Perhaps in the local case the original accessor should be given exclusive access to the file?

~~5.2. Batch Oplocks~~

~~It's not clear whether NT OS/2 needs to implement specific support for batch oplocks. The distinction is made in OS/2 because open files in that system cannot be deleted or renamed. In NT OS/2, the target file *must* be open in order to be deleted and renamed. The ability to delete or rename a file that someone else has open is controlled by sharing modes.~~

~~This implies that exclusive vs. batch oplocks can be implemented in the following way:~~

- ~~An application can obtain OS/2 semantics for exclusive oplocks by not specifying *FILE_SHARE_DELETE*. Attempts to open the file for delete or rename will fail, and the oplock will be retained.~~

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- ~~An application can obtain OS/2 semantics for batch oplocks by specifying *FILE_SHARE_DELETE*. An attempt to open the file for delete or rename will cause the oplock to be broken.~~

~~The NT OS/2 LAN Manager server could implement batch oplocks in this manner. The server normally opens files without *FILE_SHARE_DELETE*, in order to obtain the sharing semantics required by the SMB protocol. However, if a file is opened with a batch oplock requested, the server could allow sharing for delete and rename.~~

~~This leads to a potential problem:~~

~~If the server allows delete sharing, but is then unable to obtain the oplock, then we have a situation in which the normal sharing rules are not being obeyed — the file can be deleted or renamed out from under the client. Perhaps this is not a problem; perhaps it is a small price to pay. On the other hand, to maintain the sharing rules, perhaps the server should close the file and reopen it without delete sharing.~~

~~Now suppose that we do obtain the oplock, but it is subsequently broken. Again, we have the file open with delete allowed, and the questions above apply. This problem may not be important, however, because the response to a batch oplock break should be to close the file.~~

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6. Revision History

Original Draft Revision 0.1, April 2, 1990

Revision 0.2, August 15, 1990 -- Implementation details added

Revision 0.3, January 2, 1991 -- Implementation details of initial implementation added. Corrections to initial implementation details made.

Revision 0.4, June 12, 1991 -- Interface extended to support batch oplocks.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

OS/2 Emulation Subsystem Specification

Author: *Steven R. Wood*

Revision 1.0, January 19, 1990
Original Draft August 15, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview.. 1
1.1 OS/2 DLL State.. 1
1.2 OS/2 Server State... 1
1.3 OS/2 Kernel Extension State 2
1.4 Process Structure... 2
1.5 Name Processing.. 4
1.6 File Handle Processing 4
1.7 32 Bit OS/2 API Summary 5
1.8 Rationale for Not Implemented OS/2 API Calls.. 8

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This specification describes the design and implementation of the OS/2 Emulation Subsystem for NT OS/2. The subsystem consists of a dynamic link library (DLL) that resides in the OS/2 application's address space, a server process that maintains global state across all OS/2 applications and an NT OS/2 Kernel Extension that runs in Kernel Mode and implements the OS/2 Semaphore primitives.

The DLL exports as entry points all of the 32 bit Dos32 API's defined by OS/2 Version 2.0 (Cruiser). Some of the entry points, that only manipulate process private state are implemented entirely in the DLL. Others call the OS/2 Subsystem server to access and/or modify the global state it maintains. Finally, the 32 bit Dos Semaphore API's call the OS/2 Subsystem Kernel Extension.

For the remainder of this document, these three components will be referred to as: the OS/2 Server, the OS/2 DLL and the OS/2 Kernel Extension.

1.1 OS/2 DLL State

The OS/2 DLL maintains the following information for each OS/2 process:

- Current drive
- Current directory for each drive
- Environment variables
- Command line
- OS/2 File handle table
- Hard Error and Verify flags
- Thread Information Block (TIB) for each thread

1.2 OS/2 Server State

The OS/2 Server maintains the following state:

- Hierarchy of OS/2 processes
- List of threads for each process
- Exit list procedures for each process
- Shadow of file handle table for each process
- Queues, Pipes and Shared memory objects

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Keyboard buffer for Dos32Read calls to Standard Input

- Listen Thread that listens for connection requests from OS/2 applications.

- Keyboard Thread that is waiting on a Presentation Manager message queue for keyboard events. This message queue is associated with any character mode window. In the current OS/2 1.1 implementation, this thread runs in the task manager process.

- Request Threads. The number of request threads will vary dynamically based on the number of outstanding connections to OS/2 applications. The exact ratio will be determined during performance analysis.

- Exception Port Thread that is waiting for exceptions for OS/2 application threads that were not handled.

- Session Manager Thread that is dedicated to servicing requests from the NT OS/2 Session Manager

1.3 OS/2 Kernel Extension State

The OS/2 Kernel Extension maintains the following state:

- OS/2 Event Semaphore objects

- OS/2 Mutex Semaphore objects

- OS/2 MuxWait Semaphore objects

1.4 Process Structure

The OS/2 Server is responsible for creating all OS/2 processes and maintain a process tree structure that describes the relationship between OS/2 processes. For each process the following information is maintained:

- OS/2 PID value

- NT OS/2 Process Handle

- Parent process

- Sibling process list

- Child process list

- Thread Table

- File Handle Table

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Process creation is the result of one of several external events:

- o an OS/2 application calls Dos32ExecPgm
- o an OS/2 application calls Dos32StartSession
- o the NT OS/2 Session Manager calls the OS/2 Server to start an OS/2 application.
- o opens the image file
- o creates a process with that image file mapped
- o extracts the entry address and program type from the image header
- o allocates a stack and fills in the TEB with the stack bounds
- o creates a suspended thread with an initial context that points to the correct entry address and stack
- o Client Id
- o Process and Thread handles
- o Type of image file

If the type of the image is not OS/2, then the OS/2 Server will pass the information returned by SmCreateImageFileProcess back to the Session Manager and allow it to communicate the information to the appropriate subsystem (e.g. Posix). When this happens, a node is still created in the OS/2 process structure so that the foreign process has a valid process Id in the OS/2 world.

Finally, the OS/2 Server can be called by the Session Manager with an OS/2 process that was created by another subsystem calling the SmCreateImageFileProcess routine. In this case the OS/2 Server will add the process as a top level OS/2 process whose parent process is the dummy process at the root of the OS/2 process tree.

Threads within an OS/2 process are also created and managed by the OS/2 Server. The server will maintain a doubly linked list of all the threads created by the client calling the Dos32CreateThread API within a given OS/2 process. For each thread, the following information will be maintained:

- o Thread list pointers
- o Client Id
- o OS/2 TID value
- o NT OS/2 Thread Handle

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Address of OS/2 TIB in client's address space
- o Address of NT OS/2 TEB in client's address space

1.5 Name Processing

All file name parsing occurs in the OS/2 DLL. It maintains the following information in the address space of each OS/2 process:

- o Current Drive
- o Current Directory for each drive

```
\OS2\Drives\A: => \Device\Floppy1
\OS2\Drives\B: => \Device\Floppy2
\OS2\Drives\C: => \Device\SCSI0
\"LogonDirectory"\OS2\Drives\A: => \OS2\Drives\A:
\"LogonDirectory"\OS2\Drives\B: => \OS2\Drives\B:
\"LogonDirectory"\OS2\Drives\C: => \OS2\Drives\C:
\"LogonDirectory"\OS2\Drives\D: => \"LogonDirectory"\Net\Portasys
```

The double level of indirection is to allow separation of network connections between logon sessions. In order to map an OS/2 file name, into an NT OS/2 file name, the following logic will be performed by the OS/2 DLL:

- o If no drive letter, supply current drive from process state.
- o If first character after drive letter, colon is not a path separator, then supply current directory for the drive letter from process state.
- o Scan the remainder of the file name, removing any relative path specifiers (. and ..) by shifting file name characters left and removing path separators.
- o At the same time convert any forward slash (/) path separators to back slashes (\).
- o Finally, insert the \"LogonDirectory"\OS2\Drives\ string at the front of the file name.

When querying a name from NT OS/2, a reverse of some of the logic above needs to be performed. Since the only API calls that return path names are the FindFirst and FindNext, the FindFirst code can cache the user specified path name so that it and FindNext can use it to format the return buffer. This prevents the OS/2 DLL from having to decode the reverse symbolic link path that leads from \Device\SCSI0 to C:

1.6 File Handle Processing

The OS/2 Server will maintain the OS/2 File Handle table in its process state. The file handle table will be indexed by OS/2 File Handles, which are small integers, starting from 0 and going to some maximum amount. The OS/2 DLL will impose no limit on the number of file handles, other than available memory for the file handle table. The OS/2 DLL will allocate chunks of memory that hold

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

64 file handles. If more than 64 file handles are created, then two chunks will be allocated, one to hold the second group of file handles and another to act as a layer of indirection that leads to either the first or second chunks of file handles.

For each file handle, the following information is maintained:

- o NT OS/2 File Handle
- o Flags
- o Handler

The handler associated with each file handle will enable the API stubs to dispatch to the appropriate code based on the type of the file handle (NT OS/2 File Handle, OS/2 Pipe Handle, etc.).

1.7 32 Bit OS/2 API Summary

Below is a complete list of all the 32-Bit OS/2 API calls supported by OS/2 2.0 (aka Cruiser). For each call, it is identified whether the call is implemented in the OS/2 Server, the OS/2 DLL, the OS/2 Kernel Extension or not implemented. In the case of calls implemented in the OS/2 Server, there is also work done in the OS/2 DLL to prepare the parameters for the server and to process the results from the call to the server.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Dos32QuerySysInfo             DLL
Dos32Error                    DLL
Dos32CreateThread             Server
Dos32WaitChild                Server
Dos32WaitThread               Server
Dos32EnterCritSec             Server
Dos32ExitCritSec              Server
Dos32ExecPgm                  Server
Dos32Exit                     Server
Dos32ExitList                 Server
Dos32GetThreadInfo            DLL
Dos32SetPriority              DLL
Dos32KillProcess              Server
Dos32ResumeThread             Server
Dos32SuspendThread            Server
Dos32CreatePipe               Server
Dos32CallNPipe                Server
Dos32ConnectNPipe             Server
Dos32DisConnectNPipe          Server
Dos32CreateNPipe              Server
Dos32PeekNPipe                Server
Dos32QueryNPHState            Server
Dos32QueryNPipeInfo           Server
Dos32QueryNPipeSemState       Server
Dos32RawReadNPipe             Server
Dos32RawWriteNPipe            Server
Dos32SetNPHState              Server
Dos32SetNPipeSem              Server
Dos32TransactNPipe            Server
Dos32WaitNPipe                Server
Dos32CreateQueue              Server
Dos32OpenQueue                Server
Dos32CloseQueue               Server
Dos32PeekQueue                Server
Dos32PurgeQueue               Server
Dos32QueryQueue               Server
Dos32ReadQueue                Server
Dos32WriteQueue               Server
Dos32CreateEventSem           Kernel Extension
Dos32OpenEventSem             Kernel Extension
Dos32CloseEventSem            Kernel Extension
Dos32ResetEventSem            Kernel Extension
Dos32PostEventSem             Kernel Extension
Dos32WaitEventSem             Kernel Extension
Dos32QueryEventSem            Kernel Extension
Dos32CreateMutexSem           Kernel Extension
Dos32OpenMutexSem             Kernel Extension
Dos32CloseMutexSem            Kernel Extension
Dos32RequestMutexSem          Kernel Extension
Dos32ReleaseMutexSem          Kernel Extension
Dos32QueryMutexSem            Kernel Extension
Dos32CreateMuxWaitSem         Kernel Extension
Dos32OpenMuxWaitSem           Kernel Extension
Dos32CloseMuxWaitSem          Kernel Extension
Dos32WaitMuxWaitSem           Kernel Extension
Dos32AddMuxWaitSem            Kernel Extension
Dos32DeleteMuxWaitSem         Kernel Extension
Dos32QueryMuxWaitSem          Kernel Extension
Dos32GetDateTime              DLL
Dos32SetDateTime              DLL
Dos32Sleep                    DLL
Dos32AsyncTimer               DLL
Dos32StartTimer               DLL
Dos32StopTimer                DLL
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Dos32AliasMem                  not implemented
Dos32AllocMem                  DLL
Dos32AllocSharedMem            Server
Dos32GetNamedSharedMem         Server
Dos32GetSharedMem              Server
Dos32GiveSharedMem             Server
Dos32FreeMem                   DLL
Dos32SetMem                    DLL
Dos32QueryMemState             not implemented
Dos32QueryMem                  DLL
Dos32SubAlloc                  DLL
Dos32SubFree                   DLL
Dos32SubSet                    DLL
Dos32LoadModule                Server
Dos32FreeModule                Server
Dos32QueryProcAddr             Server
Dos32QueryModuleHandle         Server
Dos32QueryModuleName           Server
Dos32GetResource               Server
Dos32QueryAppType              Server
Dos32Beep                      DLL
Dos32DevConfig                 DLL
Dos32PhysicalDisk              not implemented
Dos32ScanEnv                   DLL
Dos32SearchPath                DLL
Dos32QueryVerify               DLL
Dos32SetVerify                 DLL
Dos32SetMaxFH                  DLL
Dos32Open                      Server
Dos32SetFHState                DLL
Dos32QueryFHState              DLL
Dos32QueryHType                DLL
Dos32QueryFileMode             DLL
Dos32SetFileMode               DLL
Dos32SetFileInfo               DLL
Dos32QueryFileInfo             DLL
Dos32ResetBuffer               DLL
Dos32SetFilePtr                DLL
Dos32Read                      DLL
Dos32Write                     DLL
Dos32Close                     Server
Dos32DevIOCtl                  not implemented
Dos32DupHandle                 Server
Dos32FileIO                    DLL
Dos32SetFileLocks              DLL
Dos32SetFileSize               DLL
Dos32FindFirst                 DLL
Dos32FindNext                  DLL
Dos32FindClose                 DLL
Dos32FindNotifyFirst           DLL
Dos32FindNotifyNext            DLL
Dos32FindNotifyClose           DLL
Dos32SetDefaultDisk            DLL
Dos32QueryCurrentDisk          DLL
Dos32SetCurrentDir             DLL
Dos32QueryCurrentDir           DLL
Dos32Delete                    DLL
Dos32EditName                  DLL
Dos32QueryPathInfo             DLL
Dos32SetPathInfo               DLL
Dos32SetCurrentDir             DLL
Dos32CreateDir                 DLL
Dos32DeleteDir                 DLL
Dos32Move                      DLL
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Dos32Copy                               DLL
Dos32FSAttach                           not implemented
Dos32FSCtl                              not implemented
Dos32QueryFSAttach                      not implemented
Dos32SetFSInfo                          not implemented
Dos32QueryFSInfo                        not implemented
Dos32GetMessage                         DLL
Dos32InsertMessage                      DLL
Dos32PutMessage                         DLL
Dos32SetProcessCp                       DLL
Dos32QueryCp                            DLL
Dos32QueryCtryInfo                      DLL
Dos32QueryDBCSEnv                       DLL
Dos32QueryCollate                       DLL
Dos32MapCase                            DLL
Dos32StartSession                       Server
Dos32SetSession                         Server
Dos32SelectSession                      Server
Dos32StopSession                        Server
Dos32SetExceptionHandler                DLL
Dos32UnsetExceptionHandler              DLL
Dos32RaiseException                     DLL
Dos32UnwindException                    DLL
Dos32SendException                      eliminated(D658)
Dos32FlagProcess                        eliminated(D658)
Dos32ErrClass                           DLL
```

1.8 Rationale for Not Implemented OS/2 API Calls

Dos32QueryMemState is an internal API added for Component Test and performance testing. It is not part of the OS/2 2.0 API, even though it appears in BSEDOS.H.

Dos32AliasMem is an internal API added to support the 32 to 16 bit thunk code. It is not part of the OS/2 2.0 API, even though it appears in BSEDOS.H.

The five Installable File System calls: Dos32FSAttach, Dos32FSCtl, Dos32QueryFSAttach, Dos32SetFSInfo and Dos32QueryFSInfo are not implemented because Portable OS/2 is not compatible with existing IFS implementations.

Dos32DevIOCtl is not implemented because Portable OS/2 is not compatible with existing OS/2 device drivers. In addition, the Dos32DevIOCtl API in OS/2 V2.0 is only specified to work with 16 bit device drivers.

Dos32PhysicalDisk is not implemented because it provides a means for accessing the physical media via Dos32DevIOCtl calls, which is not implemented for Portable OS/2. We made need to support the ability of the Dos32PhysicalDisk API to return partition information for a drive, but for now there is no plan to do so.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Prefix Table Specification

Author: *Gary D. Kimura*

Revision 1.2, August 2, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction..1

2. Initializing a Prefix Table..1

3. Adding a New Prefix...2

4. Removing a Prefix..2

5. Locating a Prefix..2

6. Enumerating a Prefix Table...3

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the **Windows NT** routines that implement a prefix table package. The **Windows NT** prefix table package is designed for storing and matching path name prefixes.

The prefix table package exports two opaque types, one is a prefix table used to denote a collection of prefixes, and the other is a prefix table entry used to denote a prefix. A user of this package first initializes a prefix table variable and then either inserts or deletes prefixes, or finds the longest matching prefix in the table.

To utilize this package, the caller needs to define a local structure to contain a prefix table entry. When inserting a new prefix, the caller then supplies a prefix table, prefix string, and a prefix table entry.

To look up a prefix the caller supplies a prefix table and a full path name. If a prefix match is found, the look up procedure returns a pointer to the prefix table entry corresponding to the located prefix. The programmer can then use the CONTAINING_RECORD macro to associate the prefix table entry with the local data structures.

Only prefixes that match whole logical parts of a path name are returned. For example, if a table contains the prefix "\Alpha\Beta" then a look up on "\Alpha\", "\Alpha\Bet" and "\Alpha\BetaGamma" will be unsuccessful, but a look up on "\Alpha\Beta" and "\Alpha\Beta\Gamma" will be successful.

The **APIs** that implement the prefix table package are the following:

> **PfxInitialize** - Initialize a prefix table.
> **PfxInsertPrefix** - Add a new prefix to a prefix table.
> **PfxRemovePrefix** - Remove an existing prefix from a prefix table.
> **PfxFindPrefix** - Search a prefix table for the longest matching prefix.
> **PfxNextPrefix** - Enumerate all of the prefixes stored in a prefix table.

2. Initializing a Prefix Table

A prefix table is initialized with the **PfxInitialize** procedure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
PfxInitialize (
    IN PPREFIX_TABLE *PrefixTable*
    );

Parameters:

PrefixTable - A pointer to the prefix table variable being initialized

A prefix variable cannot be used by the other procedures until it has been initialized.

3. Adding a New Prefix

A user adds a new prefix to a prefix table with the **PfxInsertPrefix** procedure. The prefix is only added if it is not already in the table.

BOOLEAN
PfxInsertPrefix (
    IN PPREFIX_TABLE *PrefixTable*,
    IN PSTRING *Prefix*,
    IN PPREFIX_TABLE_ENTRY *PrefixTableEntry*
    );

Parameters:

PrefixTable - A pointer to the prefix table being modified

Prefix - The string to add to the prefix table

PrefixTableEntry - A pointer to the prefix table entry to use to denote the prefix

This procedure has a return value of TRUE if the prefix was not already in the table, and FALSE otherwise.

The prefix table keeps a reference to the input *Prefix* string, so once a prefix is added the input string must not be changed by the user.

4. Removing a Prefix

A user removes an existing prefix from a prefix table with the **PfxRemovePrefix** procedure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
VOID
PfxRemovePrefix (
    IN PPREFIX_TABLE PrefixTable,
    IN PPREFIX_TABLE_ENTRY PrefixTableEntry
    );
```

Parameters:

PrefixTable - A pointer to the prefix table being modified

PrefixTableEntry - A pointer to the prefix table entry to remove from the prefix table

5. Locating a Prefix

A user searches a prefix table for the longest matching prefix with the **PfxFindPrefix** procedure.

```
PPREFIX_TABLE_ENTRY
PfxFindPrefix (
    IN PPREFIX_TABLE PrefixTable,
    IN PSTRING FullString,
    IN BOOLEAN CaseInsensitive
    );
```

Parameters:

PrefixTable - A pointer to the prefix table being queried

FullString - A pointer to the string to use as the key in searching the prefix table

CaseInsensitive - Indicates if the prefix look up should be preformed in a manner that ignores the case (*CaseInsensitive* is TRUE) or expects an exact, case sensitive, match (*CaseInsensitive* is FALSE)

This procedure returns a pointer to the prefix table entry corresponding to the longest prefix that matches the input string if one exists and NULL otherwise.

6. Enumerating a Prefix Table

A user can enumerate all of the elements of a prefix table with the **PfxNextPrefix** procedure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PPREFIX_TABLE_ENTRY
PfxNextPrefix (
  IN PPREFIX_TABLE *PrefixTable*,
  IN BOOLEAN *Restart*
  );

<u>Parameters:</u>

PrefixTable - A pointer to the prefix table being enumerated

Restart - Indicates if the enumeration should start over

This procedure returns a pointer to the next prefix stored in the prefix table, or NULL if there are no more entries. The following code fragment illustrates how a programmer uses this procedure to enumerate all of the elements of a prefix table.

```
for (PfxTableEntry = PfxNextPrefix( PrefixTable, TRUE );
     PfxTableEntry != NULL;
     PfxTableEntry = PfxNextPrefix( PrefixTable, FALSE )) {

    LocalRecord = CONTAINING_RECORD( PfxTableEntry,
                                     LOCAL_RECORD,
                                     PrefixTableEntry);
    ...
}
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, August 1, 1989

Revision 1.1, August 2, 1989

1. Added statement concerning how the prefix table keeps a pointer back to the input prefix string.

2. Changed **PfxAddPrefix** to **PfxInsertPrefix**.

3. Dropped *PrefixLength* from **PfxFindPrefix**, and added *CaseInsensitive* parameter.

4. Changed **PfxNextPrefix** prototype and added example of its use.

Revision 1.2, August 3, 1989

1. Changed **PfxFindPrefix** to return a pointer to the table entry if one is found instead of an OUT parameter.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Process Structure

Author: *Mark Lucovsky*

Revision 1.27, January 14, 1992

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview.. 1

2. Process Structure Objects 1

3. Process Object APIs 1
3.1 Access Type And Privilege Information.................................... 2
3.2 NtCreateProcess 4
3.3 NtTerminateProcess 5
3.4 NtCurrentProcess 6
3.5 NtCurrentPeb 7
3.6 NtOpenProcess 8
3.7 NtQueryInformationProcess 8
3.8 NtSetInformationProcess 12

4. Thread Object APIs 14
4.1 Access Type And Privilege Information.................................... 14
4.2 NtCreateThread 16
4.3 NtTerminateThread 19
4.4 NtCurrentThread 20
4.5 NtCurrentTeb 20
4.6 NtSuspendThread 21
4.7 NtResumeThread 22
4.8 NtGetContextThread 22
4.9 NtSetContextThread 23
4.10 NtOpenThread 24
4.11 NtQueryInformationThread 25
4.12 NtSetInformationThread 27
4.13 NtImpersonateThread 28
4.14 NtAlertThread 29
4.15 NtTestAlert 29
4.16 NtAlertResumeThread 30
4.17 NtRegisterThreadTerminationPort 30
4.18 NtImpersonateThread 32

5. System Information API 33
5.1 NtQuerySystemInformation 33

6. Executive APIs 35
6.1 PsCreateSystemProcess 36
6.2 PsCreateSystemThread 37
6.3 PsLookupProcessThreadByCid 37
6.4 PsChargePoolQuota 38
6.5 PsReturnPoolQuota 38
6.6 PsGetCurrentThread 39

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.7 PsGetCurrentProcess 39
6.8 KeGetPreviousMode 39
6.9 PsRevertToSelf 39
6.10 PsReferencePrimaryToken 40
6.11 PsDereferencePrimaryToken 40
6.12 PsReferenceImpersonationToken 41
6.13 PsDereferenceImpersonationToken 41
6.14 PsOpenTokenOfProcess 42
6.15 PsOpenTokenOfThread 43
6.16 PsImpersonateClient 44

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This specification describes the **Windows NT** process structure.

The **Windows NT** system is designed to support both an **OS/2** and a **POSIX** operating system environment. Rather than packaging all of the capabilities of these operating system environments into the **Windows NT** kernel and executive, the system has been designed so that robust, protected subsystems can be built to provide the necessary API emulation.

The **Windows NT** approach is very similar to the approach taken in Carnegie Mellon's MACH operating system. The MACH system design is based on a simple process structure, IPC mechanism, and virtual memory system. Using these primitives, MACH is able to implement both **POSIX** and **Unix 4.3BSD** operating system environments as protected subsystems.

Like MACH, the **Windows NT** process structure provides a very basic set of services. The system does not provide a hierarchical process tree structure, global process names (PIDs), process grouping, job control, complex process or thread termination semantics, or other more traditional process structures. It does provide a complete set of services that subsystems can use to provide the set of semantics that are required by a particular operating system environment.

Using this set of services, vendors and users can develop applications based on either the **OS/2** or **POSIX** APIs (implemented as protected subsystems by Microsoft). An alternative to this is to develop applications using the native **Windows NT** system services or to develop custom subsystems and have the applications use these subsystems.

2. Process Structure Objects

The process structure is based on two types of objects. A *process object* represents an address space, a set of objects (resources) visible to the process, and a set of threads that executes in the context of the process. A *thread object* represents the basic schedulable entity in the system. It contains its own set of machine registers, its own kernel stack, a thread environment block (TEB), and user stack in the address space of its process.

The **Windows NT** process structure works with the overall **Windows NT** security architecture. Each process is assigned an access token, called the *primary token* of the process. The primary token is used by default by the process's threads when referencing a **Windows NT** object.

In addition to the primary token, each thread may have an *impersonation token* associated with it. When this is done, the impersonation token, rather than the process's primary token, is used for access validation purposes. This is done to allow efficient impersonation of clients in a client-server model.

3. Process Object APIs

The following programming interfaces support the process object:

NtCreateProcess - Creates a process object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NtTerminateProcess - Terminates a process object.
NtCurrentProcess - Identifies the currently executing process.
NtCurrentPeb - Returns the address of the current processes Process Environment Block (PEB).
NtOpenProcess - Creates a handle to a process object.
NtQueryInformationProcess - Returns information about the process.
NtSetInformationProcess - Sets information about the process.

3.1 Access Type And Privilege Information

Object type-specific access types:

The object type-specific access types are defined below.

PROCESS_TERMINATE - Required to terminate a process.

PROCESS_CREATE_THREAD - Required to create a thread in a process.

PROCESS_VM_OPERATION - Required to manipulate the address space of a process. This does not include reading and writing the memory of a process.

PROCESS_VM_READ - Required to read the virtual memory of a process (through **Windows NT APIs**).

PROCESS_VM_WRITE - Required to write the virtual memory of a process (through **Windows NT APIs**).

PROCESS_DUP_HANDLE - Required to duplicate an object handle visible to a process.

PROCESS_CREATE_PROCESS - Required to create a process.

PROCESS_SET_QUOTA - Required to modify the quota limits of a process.

PROCESS_SET_INFORMATION - Required to modify certain attributes of a process.

PROCESS_QUERY_INFORMATION - Required to read certain attributes of a process. This access type is also needed to open the primary token of a process (using **NtOpenProcessToken()**).

PROCESS_SET_PORT - Required to set the debug or exception port of a process.

Generic Access Masks:

The object type-specific mapping of generic access types to non-generic access types for this object type are:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

GENERIC_READ	**STANDARD_READ | PROCESS_VM_READ | PROCESS_QUERY_INFORMATION**
GENERIC_WRITE	**STANDARD_WRITE | PROCESS_TERMINATE | PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_DUP_HANDLE | PROCESS_CREATE_PROCESS | PROCESS_SET_QUOTA | PROCESS_SET_INFORMATION | PROCESS_SET_PORT**
GENERIC_EXECUTE	**STANDARD_EXECUTE | SYNCHRONIZE**

Standard Access Types:

This object type supports the optional SYNCHRONIZE standard access type. All required access types are supported by the object manager.

The mask of all supported access types for this object is:

PROCESS_ALL_ACCESS	**STANDARD_RIGHTS_REQUIRED | SYNCHRONIZE | PROCESS_TERMINATE | PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_DUP_HANDLE | PROCESS_CREATE_PROCESS | PROCESS_SET_QUOTA | PROCESS_SET_INFORMATION | PROCESS_QUERY_INFORMATION | PROCESS_SET_PORT**

Privileges Defined Or Used:

This object type defines or uses the following privileges:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SeAssignPrimaryTokenPrivilege - This privilege is needed to assign a new primary token for a process.

3.2 NtCreateProcess

A process object can be created and a handle opened for access to the process with the **NtCreateProcess** function:

NTSTATUS
NtCreateProcess(
 OUT PHANDLE *ProcessHandle,*
 IN ACCESS_MASK *DesiredAccess,*
 IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
 IN HANDLE *ParentProcess,*
 IN BOOLEAN *InheritObjectTable,*
 IN HANDLE *SectionHandle* **OPTIONAL**,
 IN HANDLE *DebugPort* **OPTIONAL**,
 IN HANDLE *ExceptionPort* **OPTIONAL**
 );

Parameters:

ProcessHandle - A pointer to a variable that will receive the process object handle value.

DesiredAccess - The desired types of access to the created process.

ObjectAttributes - An optional pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT*, *OBJ_EXCLUSIVE*, *OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a process object.

ParentProcess - An open handle to a process object. The new process is created using some of the attributes of the specified parent process. *PROCESS_CREATE_PROCESS* access to this process is required.

InheritObjectTable - A flag which determines whether or not the new process will be created with an object table whose initial contents come from the specified parent process. A value of false causes the new process to be created with an empty object table. A value of true causes the new process to be created by cloning the parent process's object table. All objects in the parent process's object table marked with the *OBJ_INHERIT* attribute appear in the new process's object table with exactly the same handle values, attributes, and granted access.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SectionHandle - An optional open handle to a section object. If the value of the argument is not null, then it specifies a handle to a section object backed by an image file the process is being created to run. *SECTION_MAP_EXECUTE* access to the section object is required.

DebugPort - An optional open handle to a port object. If specified, the port is assigned as the process's debugger port; otherwise, the process is created without a debugger port. *PORT_WRITE* and *PORT_READ* access to the port object are required.

ExceptionPort - An optional open handle to a port object. If specified, the port is assigned as the process's exception port; otherwise, the process is created without an exception port. *PORT_WRITE* and *PORT_READ* access to the port object are required.

Creating a process object causes a new process to be created. The new process shares some of its initial attributes with the specified parent process.

- The new process is created with an object table. The table is either an empty table, or a clone of the parent process's object table. This is a function of the *InheritObjectTable* parameter.
- The access token of the new process is identical to the access token of the parent process.
- The quota limits of the new process are identical to the quota limits of the parent process.
- The base priority of the new process is identical to the base priority of the parent process.

The address space of the new process is defined by the specified section handle or the address space of the specified parent process. If the section handle is not null, the section object must be backed by an image file. The address space of the new process is created by mapping a view of the entire section object. Otherwise, the address space of the process is created by copying or sharing those pieces of the parent process's address space marked as **PAG_COPY/PAG_SHARE** into the address space of the new process.

The new process is created without any threads.

Each process is created with a Process Environment Block (PEB). The PEB is readable and writeable by the application, but can only be deleted by the system. The PEB is partially initialized by the system and is placed in the address space of the. If the process is created without a section handle, then the new processes PEB is shared "copy on write" with the parent process PEB.

The PEB contains process global context such as startup parameters, image base address, a Mutant object handle for process wide synchronization, and loader data structures.

The function **NtCurrentPeb** returns the address of the current processes PEB. Access to PEB locations must be made through this API.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The process object is a waitable object. A wait performed on a process object is satisfied when the process becomes signaled. A process becomes signaled when its last thread terminates, or if a process without a thread is terminated with **NtTerminateProcess**.

Both the debugger and exception ports are used by the exception handling system within **Windows NT**. The role that these ports play in exception handling is described in another document.

3.3 NtTerminateProcess

A process can be terminated with the **NtTerminateProcess** function:

NTSTATUS
NtTerminateProcess(
    IN HANDLE *ProcessHandle OPTIONAL,*
    IN NTSTATUS *ExitStatus*
    );

Parameters:

ProcessHandle - An optional parameter, that if specified, supplies an open handle with **PROCESS_TERMINATE** access to the process to terminate. If this parameter is not supplied, then **PROCESS_TERMINATE** access is required to the current process and the API terminates all threads in the process except for the calling thread.

ExitStatus - A value that specifies the exit status of the process to be terminated.

Terminating a process causes the specified process and all of its threads to terminate. Any threads in the process that are suspended are resumed by this service so that they can begin termination. The handles of the process's threads are not explicitly closed by this service. The handle to the process being terminated is also not closed by this service. If any thread in the process was suspended and resumed by this API and informational status code of *STATUS_THREAD_WAS_SUSPENDED* is returned.

In order to terminate a process, the calling thread must have *PROCESS_TERMINATE* access to the specified process.

After all of the process's threads are terminated (and set to the signaled state), the process's object table is processed by closing all open handles.

The process object is signaled upon termination, and its exit status is updated to reflect the value of the exit status argument. Once a process object becomes signaled, no more threads can be created in the process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The process's address space remains valid until the process object itself is deleted (the last handle to the process object is closed).

3.4 NtCurrentProcess

An object handle to the current process can be fabricated with the **NtCurrentProcess** function:

HANDLE
NtCurrentProcess();

The **NtCurrentProcess** function returns a pseudo handle to the currently executing process. The handle can be used whenever a handle to a process object is required (e.g. **NtTerminateProcess**).

When the system is asked to translate an object handle into an object pointer, the object type is a process object, and the object handle is the pseudo handle returned by **NtCurrentProcess**, the following occurs.

- The **SECURITY_DESCRIPTOR** of the current process is checked against the desired access specified in the object translation call. If access is denied a failure status is returned to the caller.
- If access is allowed, the appropriate reference count in the current process object is adjusted and a pointer to the current process object is returned.

This function is designed mainly for the use of native applications so that they can refer to their own process in process termination calls, thread creation calls, and address space modification calls without having to explicitly open their process by name or otherwise obtain a handle to their own process. A similar function exists to reference the currently executing thread.

3.5 NtCurrentPeb

The address of the current processes **PEB** can be located with the **NtCurrentPeb** function:

PPEB
NtCurrentPeb()

The **NtCurrentPeb** function returns the address of the current processes **PEB**. The **PEB** consists of a single page in the address space of the process. The page is allocated and deallocated by the system at process creation/process termination. Only the system may delete a processes **PEB**. The **PEB** contains the following:

<u>Peb Structure</u>

BOOLEAN *InheritedAddressSpace* - A flag set by the system to indicate that the processes initial address space was from inheritance rather than from a mapping a section.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

HANDLE *Mutant* - Contains a handle to a mutant object. Various portions of the system use this mutant to synchronize within the process. The functions **RtlAcquirePebLock** and **RtlReleasePebLock** may be used to access this field.

PCOFF_HEADERS *ImageBaseAddress* - Contains the address of the image header of the processes initial image.

PPEB_LDR_DATA *Ldr* - Contains the address of the loaders per-process data. The value of this pointer is null until the first thread of a process initialzes the loader.

PEB_SM_DATA *Sm* - Contains Session Manager specific information.

PRTL_USER_PROCESS_PARAMETERS *ProcessParameters* - Contains the address of the processes startup parameters.

PVOID *SubsystemData* - Contains the address of subsystem specific data.

PPEB_FREE_BLOCK *FreeList* - Contains the address of a dynamic area in the PEB. Calls to **RtlAllocateFromPeb** and **RtlFreeToPeb** are satisfied from this area.

3.6 NtOpenProcess

A handle to a process object can be created with the **NtOpenProcess** function:

NTSTATUS
NtOpenProcess(
    OUT PHANDLE *ProcessHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes,*
    IN PCLIENT_ID *ClientId* **OPTIONAL**
    );

Parameters:

ProcessHandle - A pointer to a variable that will receive the process object handle value.

DesiredAccess - The desired types of access to the opened process. For a complete description of desired access flags, refer to the **NtCreateProcess API** description.

ObjectAttributes - An pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a process object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ClientId - An optional parameter that if specified, supplies the client ID of a thread whose process is to be opened. It is an error to specify this parameter along with the an *ObjectAttributes* variable that contains a process name.

Opening a process object causes a new handle to be created. The access that the new handle has to the process object is a function of the desired access and any *SECURITY_DESCRIPTOR* on the process object

3.7 NtQueryInformationProcess

Selected information about a process can be retrieved using the **NtQueryInformationProcess** function.

NTSTATUS
NtQueryInformationProcess(
    IN HANDLE *ProcessHandle,*
    IN PROCESSINFOCLASS *ProcessInformationClass*
    OUT PVOID *ProcessInformation,*
    IN ULONG *ProcessInformationLength,*
    OUT PULONG *ReturnLength* **OPTIONAL**
    );

Parameters:

ProcessHandle - A variable that specifies the handle to a process from which to retrieve information.

ProcessInformationClass - A variable that specifies the type of information to retrieve from the specified process object.

ProcessInformationClass Values

ProcessBasicInformation - Returns the basic information about the specified process. This information class value requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessQuotaLimits - Returns the quota limits of the specified process. This information class requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessIoCounters - Returns the input/output counters of the specified process. This information class requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessVmCounters - Returns the virtual memory counters of the specified process. This information class requires *PROCESS_QUERY_INFORMATION* access to the process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ProcessTimes - Returns the cpu time usage of the specified process. This information class requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessLdtInformation - Returns the contents of the Ldt for the process. Requires *PROCESS_VM_READ* access to the process. Returns *STATUS_NOT_SUPPORTED* on non i386 (and compatible) processors.

ProcessInformation - A pointer to a buffer that will receive information about the specified process. The format and contents of the buffer depend on the specified information class being queried.

ProcessInformation Format by Information Class

ProcessBasicInformation - Data type is *PPROCESS_BASIC_INFORMATION*.

PROCESS_BASIC_INFORMATION Structure

NTSTATUS *ExitStatus* - Specifies the exit status of the process. This field only contains meaningful information if the process is in the signaled state; otherwise, it contains a value of "exit status pending".

PPEB *PebBaseAddress* - Specifies the base address of the processes PEB.

KPRIORITY *BasePriority* - Specifies the base priority of the process.

KAFFINITY *AffinityMask* - Specifies the default affinity mask assigned to each thread in the process during thread creation.

ProcessQuotaLimits - Data type is *PQUOTA_LIMITS*.

QUOTA_LIMITS Structure

ULONG *PagedPoolLimit* - Specifies the maximum amount of paged pool (in bytes) that can be used by the process.

ULONG *NonPagedPoolLimit* - Specifies the maximum amount of nonpaged pool (in bytes) that can be used by the process.

ULONG *MinimumWorkingSetSize* - Specifies the minimum working set size (in bytes) for the process.

ULONG *MaximumWorkingSetSize* - Specifies the maximum working set size (in bytes) for the process.

ULONG *PagefileLimit* - Specifies the maximum amount of pagefile space (in bytes) that can be used by the process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

TIME *TimeLimit* - Specifies the maximum number of 100ns units that the process can execute for.

ProcessIoCounters - Data type is *PIO_COUNTERS*.

IO_COUNTERS Structure

ULONG *ReadOperationCount* - Specifies the number of read I/O operations performed by the process.

ULONG *WriteOperationCount* - Specifies the number of write I/O operations performed by the process.

ULONG *OtherOperationCount* - Specifies the number of other I/O operations (not read or write) performed by the process.

LARGE_INTEGER *ReadTransferCount* - Specifies the number of bytes transferred through read I/O operations.

LARGE_INTEGER *WriteTransferCount* - Specifies the number of bytes transferred through write I/O operations.

LARGE_INTEGER *OtherTransferCount* - Specifies the number of bytes transferred through other I/O operations.

ProcessVmCounters - Data type is *PVM_COUNTERS*.

VM_COUNTERS Structure

ULONG *PeakVirtualSize* - Specifies the largest virtual address space size (in bytes) that the process has reached.

ULONG *VirtualSize* - Specifies the current virtual address space size (in bytes) of the process.

ULONG *PageFaultCount* - Specifies the number of pagefaults incurred by the process.

ULONG *PeakWorkingSetSize* - Specifies the largest working set size (in bytes) that the process has reached.

ULONG *WorkingSetSize* - Specifies the current working set size (in bytes) of the process.

ULONG *QuotaPeakPagedPoolSize* - Specifies the largest amount of paged pool (in bytes) that the process has used and has been charged quota for.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG *QuotaPagedPoolSize* - Specifies the current amount of paged pool (in bytes) in use by the process and being charged to the process.

ULONG *QuotaNonPeakPagedPoolSize* - Specifies the largest amount of nonpaged pool (in bytes) that the process has used and has been charged quota for.

ULONG *QuotaNonPagedPoolSize* - Specifies the current amount of nonpaged pool (in bytes) in use by the process and being charged to the process.

ULONG *PagefileUsage* - Specifies the current amount of pagefile space (in bytes) in use by the process.

ProcessTimes - Data type is *PKERNEL_USER_TIMES*.

KERNEL_USER_TIMES Structure

TIME *UserTime* - Specifies the number of 100ns units that the process has spent executing in user mode.

TIME *KernelTime* - Specifies the number of 100ns units that the process has spent executing in kernel mode.

TIME *CreateTime* - Specifies the time that the process was created.

TIME *ExitTime* - Specifies the time that the process terminated.

ProcessLdtInformation - Data type is *PPROCESS_LDT_INFORMATION*.

PROCESS_LDT_INFORMATION Structure

ULONG *Start* - Specifies the starting offset in the LDT to return descriptors from. It must be 0 mod 8. If this value is larger than the current size of the LDT, no information will be put into the LdtEntries field.

ULONG *Length* - Supplies the length of the section of the LDT to return. Must be 0 mod 8. Returns the length of the Ldt. Will always be set.

LDT_ENTRY *LdtEntries[1]* - Variable size array of LDT_ENTRYs, is the actual Ldt data in hardware format.

ProcessInformationLength - Specifies the length in bytes of the process information buffer (i.e. size of the information structure).

ReturnLength - An optional parameter that if specified, receives the number of bytes placed in process information buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.8 NtSetInformationProcess

Selected information can be set in a process using the **NtSetInformationProcess** function.

NTSTATUS
NtSetInformationProcess(
  IN HANDLE *ProcessHandle,*
  IN PROCESSINFOCLASS *ProcessInformationClass,*
  IN PVOID *ProcessInformation,*
  IN ULONG *ProcessInformationLength*
  );

Parameters:

ProcessHandle - A variable that specifies the handle to a process to set information into.

ProcessInformationClass - A variable that specifies the type of information to set into the specified process object.

ProcessInformationClass Values

ProcessBasePriority - Sets the base priority of the specified process. This information class value requires *PROCESS_SET_INFORMATION* access to the process.

ProcessQuotaLimits - Sets the quota limits associated with the process. This information class value requires *PROCESS_SET_QUOTA* access to the process. If an attempt is made to increase quota, a privilege check is done to ensure that the calling process has ***TBD*** privilege.

ProcessAccessToken - Sets the primary access token of the specified process. This information class requires *PROCESS_SET_INFORMATION* access to the process. Furthermore, the caller must have **SeAssignPrimaryTokenPrivilege** privilege.

Since the process access token is inherited during process creation, this operation only needs to be performed when a process is being created for a new user or for a privileged application.

ProcessDebugPort - Sets the debug port of the specified process. If the process already has a debug port either through process creation, or a previous call to **NtSetInformationProcess** then an error is returned. This information class requires *PROCESS_SET_PORT* access to the process.

ProcessExceptionPort - Sets the exception port of the specified process. If the process already has an exception port either through process creation, or a previous call to **NtSetInformationProcess** then an error is returned. This information class requires *PROCESS_SET_PORT* access to the process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ProcessLdtInformation - Returns the contents of the Ldt for the process. Requires *PROCESS_VM_WRITE* access to the process. Returns *STATUS_NOT_SUPPORTED* on non i386 (and compatible) processors.

ProcessLdtSize - Returns the size of the Ldt for the process. *PROCESS_VM_WRITE* access required. Returns *STATUS_NOT_SUPPORTED* on non i386 (and compatible) processors.

ProcessInformation - A pointer to a buffer that contains the information to set in the specified process. The format and contents of the buffer depend on the specified information class being queried.

ProcessInformation Format by Information Class

ProcessBasePriority - Data type is *KPRIORITY*.

KPRIORITY *BasePriority* - Specifies the base priority of the process.

ProcessQuotaLimits - Data type is *PQUOTA_LIMITS*.

QUOTA_LIMITS Structure

ULONG *PagedPoolLimit* - Specifies the maximum amount of paged pool (in bytes) that can be used by the process.

ULONG *NonPagedPoolLimit* - Specifies the maximum amount of nonpaged pool (in bytes) that can be used by the process.

ULONG *MinimumWorkingSetSize* - Specifies the minimum working set size (in bytes) for the process.

ULONG *MaximumWorkingSetSize* - Specifies the maximum working set size (in bytes) for the process.

ULONG *PagefileLimit* - Specifies the maximum amount of pagefile space (in bytes) that can be used by the process.

TIME *TimeLimit* - Specifies the maximum number of 100ns units that the process can execute for.

ProcessAccessToken - Data type is *PHANDLE*. The handle is expected to be to a Token object. The handle must have been opened to provide **TOKEN_ASSIGN_PRIMARY** access.

ProcessDebugPort - Data type is *PHANDLE*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ProcessExceptionPort - Data type is *PHANDLE*.

ProcessLdtInformation - Data type is *PPROCESS_LDT_INFORMATION*.

PROCESS_LDT_INFORMATION Structure

ULONG *Start* - Offset in Ldt of first entry to set. Must be 0 mod 8.

ULONG *Length* - Length of section of Ldt to set. Must be 0 mod 8.

LDT_ENTRY *LdtEntries[1]* - Variable size array of LDT_ENTRYs, is the actual Ldt data in hardware format.

ProcessLdtSize - Data type is *PPROCESS_LDT_SIZE*.

PROCESS_LDT_SIZE Structure

ULONG *Length* - Size to set Ldt to. Setting 0 sets a null Ldt. Can be used to truncate the Ldt. Must be 0 mod 8.

ProcessInformationLength - Specifies the length in bytes of the process information buffer.

4. Thread Object APIs

The following programming interfaces support the thread object:

NtCreateThread - Creates a thread object.
NtTerminateThread - Terminates a thread object.
NtCurrentThread - Identifies the currently executing thread.
NtCurrentTeb - Returns the address of the current thread's Thread Environment Block (TEB).
NtSuspendThread - Suspends user-mode execution of a thread.
NtResumeThread - Resumes user-mode execution of a thread.
NtGetContextThread - Returns the user-mode context of a thread.
NtSetContextThread - Sets the user-mode context of a thread.
NtOpenThread - Returns a handle to a thread object.
NtQueryInformationThread - Returns information about the thread.
NtSetInformationThread - Sets information about the thread.
NtImpersonateThread - Set one thread to be impersonating another thread.
NtAlertThread - Alerts the specified thread.
NtTestAlert - Tests for an alert condition.
NtAlertResumeThread - Alerts and resumes the specified thread.
NtRegisterThreadTerminationPort - Adds a port notification descriptor to the specified thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.1 Access Type And Privilege Information

Object type-specific access types:

The object type-specific access types are defined below.

THREAD_TERMINATE - Required to terminate a thread.

THREAD_SUSPEND_RESUME - Required to suspend or resume a thread.

THREAD_ALERT - Required to alert a thread using either **NtAlertThread** or **NtAlertResumeThread**.

THREAD_GET_CONTEXT - Required to read a thread's context (using **NtGetContextThread**).

THREAD_SET_CONTEXT - Required to modify a thread's context (using **NtSetContextThread**).

THREAD_SET_INFORMATION - Required to modify certain attributes of a thread.

THREAD_QUERY_INFORMATION - Required to read certain attributes of a thread. This access type is also needed to open the impersonation token of a thread (using **NtOpenThreadToken()**).

THREAD_SET_THREAD_TOKEN - Required to explicitly assign an impersonation token to the thread. In some cases, impersonation will happen automatically (e.g., as a result of a call from a client via LPC). However, to explicitly assign an impersonation token (via a handle) to a thread (also via a handle), requires this access to the thread.

THREAD_IMPERSONATE - Required to directly impersonate a thread. In some instances this access is not required to impersonate a thread. In particular, when a thread calls a server using an communication session layer that supports security quality of service([1]), then the server does not need to directly access the thread to impersonate. However, in some cases it is desireable to allow a server to impersonate a thread without using a communication session layer to impersonate a client. In that case, the target client thread may be opened for this access, and then a call made to **NtImpersonateThread()**.

Generic Access Masks:

The object type-specific mapping of generic access types to non-generic access types for this object type are:

---

[1] See the Windows NT Local Security Specification for more on security quality of service.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

GENERIC_READ	**STANDARD_READ | THREAD_GET_CONTEXT | THREAD_QUERY_INFORMATION**
GENERIC_WRITE	**STANDARD_WRITE | THREAD_TERMINATE | THREAD_SUSPEND_RESUME | THREAD_THREAD_ALERT | THREAD_SET_CONTEXT | THREAD_SET_INFORMATION**
GENERIC_EXECUTE	**STANDARD_EXECUTE | THREAD_SET_THREAD_TOKEN | SYNCHRONIZE**

Standard Access Types:

This object type supports the optional SYNCHRONIZE standard access type. All required access types are supported by the object manager.

The mask of all supported access types for this object is:

THREAD_ALL_ACCESS	**STANDARD_RIGHTS_REQUIRED | SYNCHRONIZE | THREAD_GET_CONTEXT | THREAD_QUERY_INFORMATION | THREAD_TERMINATE | THREAD_SUSPEND_RESUME | THREAD_THREAD_ALERT | THREAD_SET_CONTEXT | THREAD_SET_INFORMATION | THREAD_SET_THREAD_TOKEN | THREAD_IMPERSONATE | THREAD_DIRECT_IMPERSONATION**

4.2 NtCreateThread

A thread object can be created and a handle opened for access to the thread with the **NtCreateThread** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtCreateThread(
OUT PHANDLE *ThreadHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL***,*
IN HANDLE *ProcessHandle,*
OUT PCLIENT_ID *ClientId,*
IN PCONTEXT *ThreadContext,*
IN PINITIAL_TEB *InitialTeb,*
IN BOOLEAN *CreateSuspended*
);

Parameters:

ThreadHandle - A pointer to a variable that will receive the thread object handle value.

DesiredAccess - The desired types of access to the created thread.

ObjectAttributes - An optional pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a thread object.

ProcessHandle - An open handle to the process object that the thread is to run in. The subject thread must have *PROCESS_CREATE_THREAD* access to this process. The value of this argument may be the value returned by **NtCurrentProcess** to specify that the new thread is to be created in the context of the current process.

ClientId - A pointer to a structure that will receive the client identifier of the new thread. Each thread in the system is assigned a client identifier value. A client identifier remains valid from the time the thread is created until it is terminated. The value of the client identifier is unique for each thread in the system. The client identifier contains two fields. One field is unique for each process in the system, and one field is unique for each thread in the system.

ClientId Structure

ULONG *UniqueProcess* - Unique value for each process in the system.

ULONG *UniqueThread* - Unique value for each thread in the system.

ThreadContext - A pointer to the structure that contains the new thread's initial user mode context.

InitialTeb - A pointer to a structure that specifies initial values for portions of the thread's TEB.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>InitialTeb Structure</u>

PVOID *StackBase* - Contains the base address of the thread's stack.

PVOID *StackLimit* - Contains the stack limit for the thread.

PVOID *EnvironmentPointer* - Unspecified.

CreateSuspended - A parameter that specifies whether or not the thread is to be created suspended. If the value of this parameter is **TRUE**, then the thread is created in a suspended state. The thread will not begin executing until it is explicitly resumed using **NtResumeThread**. If the value of this parameter is **FALSE**, then the thread begins execution in user-mode using the specified context.

Creating a thread object causes a new thread to be created. The new thread is assigned some of its initial attributes from the process object it is being created to run in.

- The new thread's priority is the same as its process's base priority.
- The new thread's processor affinity mask is the same as its process's default processor affinity mask.
- The new thread's access token is the same as its process's.

All threads begin execution with a user-mode APC to system code that is part of each processes address space. This code optionally initializes the loaders data structures and resolves dynamic link library references. When the APC routine returns, the thread's context is restored. Normally, this context is the same as that specified during thread creation.

The thread object is a waitable object. A wait performed on a thread object is satisfied when the thread becomes signaled. A thread becomes signaled when it terminates.

Each thread is created with a Thread Environment Block (TEB). The TEB is readable and writeable by the application, but can only be deleted by the system. The TEB is partially initialized by the system and is placed in the address space of the specified process.

The TEB contains thread local context such as stack base and bounds, environment pointer (used by subsystems/dll's), thread local storage descriptors, and the thread's client id. The thread's creator is responsible for initializing the TEB's stack base and bounds since it is also responsible for creating the thread's stack.

The function **NtCurrentTeb** returns the address of the current thread's TEB. Access to TEB locations must be made through this API. The TEB of each thread is located at a different address. The system will guarantee that TEB access of the form:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
foo = NtCurrentTeb()->StackBase;
NtCurrentTeb()->EnvironmentPointer = &PsxEnvironment;
```

will cause locations in the current thread's TEB to be referenced.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.3 NtTerminateThread

A thread can be terminated with the **NtTerminateThread** function:

NTSTATUS
NtTerminateThread(
    IN HANDLE *ThreadHandle* **OPTIONAL**,
    IN NTSTATUS *ExitStatus*
    );

Parameters:

ThreadHandle - An optional parameter, that if specified, supplies an open handle with **THREAD_TERMINATE** access to the thread to terminate. If this parameter is not supplied, then **THREAD_TERMINATE** access is required to the current thread and the API terminates the current thread in the process except for the case where the current thread is the last thread in the current process. In this case, a status code of *STATUS_CANT_TERMINATE_SELF* is returned.

ExitStatus - A value that specifies the exit status of the thread to be terminated.

Terminating a thread causes the specified thread to terminate its execution. If the target thread is currently suspended, it will be resumed so that it can begin termination. Once termination begins, the thread will no longer execute in either user mode or kernel mode. The handle to the thread being terminated is not closed by this service. If the thread was suspended and resumed by this API an informational status code of *STATUS_THREAD_WAS_SUSPENDED* is returned.

In order to terminate a thread, the calling thread must have *THREAD_TERMINATE* access to the specified thread.

Once a thread has become the target thread in a valid call to **NtTerminateThread** (i.e. the calling thread has *THREAD_TERMINATE* access to the target thread), the target thread will terminate without executing another instruction in user-mode. This is accomplished by queueing a special kernel-mode APC to the thread which queues a user-mode APC to the target thread and user-mode alerts the thread. The kernel routine associated with the user-mode APC will cause the thread to terminate itself. To guarantee the delivery of the user-mode APC (i.e. to bypass the alert mechanism), the user APC pending bit in the target thread is set during the execution of the special kernel-mode APC.

During thread termination, the terminating thread's port notification list is processed. For each entry in the list, a thread termination datagram is sent to the port. The system blindly ignores any errors sending this datagram (e.g. port disconnect...).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

After the thread is terminated (and set to the signaled state), the thread's TEB is deallocated from the address space of the thread's process and its exit status is updated to reflect the value of the exit status argument. The system does not delete the thread's user-mode stack.

Once terminated, the thread's client identifier is available for re-use.

If the terminating thread is the last thread in its process, its process is terminated (via an internal call to **NtTerminateProcess(NtCurrentProcess(), ExitStatus);**). There is no mechanism that a subsystem can use to prevent this from happening.

4.4 NtCurrentThread

An object handle to the current thread can be fabricated with the **NtCurrentThread** function:

HANDLE
NtCurrentThread();

The **NtCurrentThread** function returns a pseudo handle to the currently executing thread. The handle can be used whenever a handle to a thread object is required (e.g. **NtTerminateThread**).

When the system is asked to translate an object handle into an object pointer, the object type is a thread object, and the object handle is the pseudo handle returned by **NtCurrentThread**, the following occurs.

- The **SECURITY_DESCRIPTOR** of the current thread is checked against the desired access specified in the object translation call. If access is denied, a failure status is returned to the caller.

- If access is allowed, the appropriate reference count in the current thread object is adjusted and a pointer to the current thread object is returned.

This function is designed mainly for the use of native applications so that they can refer to their own thread in thread termination calls, thread creation calls, and thread control calls without having to explicitly open their thread by name or otherwise obtain a handle to their own thread.

4.5 NtCurrentTeb

The address of the current thread's **TEB** can be located with the **NtCurrentTeb** function:

PTEB
NtCurrentTeb()

The **NtCurrentTeb** function returns the address of the current thread's **TEB**. The **TEB** consists of a single page in the address space of the thread's process. The page is allocated and deallocated by the system at thread creation/thread termination. Only the system may delete a thread's **TEB**. The **TEB** contains the following:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>Teb Structure</u>

PEXCEPTION_REGISTRATION_RECORD *ExceptionRegistrationRecord* - Contains the base address of the thread's exception handler chain. This field is only used on implementations that require this sort of exception handler registration.

PVOID *StackBase* - Contains the base address of the thread's stack.

PVOID *StackLimit* - Contains the stack limit for the thread.

PVOID *EnvironmentPointer* - Unspecified.

ULONG *Version* - Unspecified.

PVOID *ArbitraryUserPointer* - Unspecified.

CLIENT_ID *ClientId* - Contains the client identifier of the thread.

PVOID *ActiveRpcHandle* - Reserved for use by the *Microsoft Remote Procedure Call Runtime Package*.

PVOID *ThreadLocalStoragePointer* - Reserved for runtime support.

PPEB *ProcessEnvironmentBlock* - Contains the base address of the thread's PEB.

PVOID *UserReserved[USER_RESERVED_TEB]* - TEB locations reserved for applications.

PVOID *SystemReserved[SYSTEM_RESERVED_TEB]* - TEB locations reserved for Microsoft system software.

4.6 NtSuspendThread

A thread can be suspended with the **NtSuspendThread** function:

NTSTATUS
NtSuspendThread(
    IN HANDLE *ThreadHandle,*
    OUT PULONG *PreviousSuspendCount* **OPTIONAL**
    );

<u>Parameters:</u>

ThreadHandle - A handle to the thread to be suspended.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PreviousSuspendCount - A pointer to the variable that receives the thread's previous suspend count.

Suspending a thread causes the thread to stop executing in user-mode. If the thread is resumed without altering its context and its previous suspend count is one, then the thread resumes execution at the point that it was suspended. If the specified thread is either terminated or is currently terminating, an error status of *STATUS_THREAD_IS_TERMINATING* is returned.

The suspension of a thread is controlled by a suspend count. This count has a maximum value. If an attempt is made to suspend a thread whose suspend count is at its maximum, an error is returned When an attempt is made to suspend a thread, the thread's suspend count is incremented. If the previous value of the suspend count was zero, then a kernel mode APC is queued to the thread. When the APC executes, it causes the thread to wait on its built-in suspend semaphore (the wait is not alertable). The previous value of the thread's suspend count is returned to the caller. A non-zero value indicates that the thread was previously suspended. The value plus 1 specifies the number of calls to **NtResumeThread** that must be made in order to bring the thread out of the suspend state.

This service requires *THREAD_SUSPEND_RESUME* access to the specified thread.

4.7 NtResumeThread

A thread can be resumed with the **NtResumeThread** function:

NTSTATUS
NtResumeThread(
    IN HANDLE *ThreadHandle,*
    OUT PULONG *PreviousSuspendCount* **OPTIONAL**
    );

Parameters:

ThreadHandle - A handle to the thread to be resumed.

PreviousSuspendCount - A pointer to the variable that receives the thread's previous suspend count.

Resuming a thread reverses the effects of a previous call to **NtSuspendThread**.

When an attempt is made to resume a thread, the thread's suspend count is examined. If the count is zero, then the service returns the suspend count. Otherwise, the count is decremented and if the count reaches zero, the thread resumes. In either case, the previous value of the thread's suspend count is returned. A non-zero value indicates that the thread was previously suspended. The value minus 1 specifies the number of calls to **NtResumeThread** that must be made in order to bring the thread out of the suspend state.

This service requires *THREAD_SUSPEND_RESUME* access to the specified thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.8 NtGetContextThread

A thread's user-mode machine context can be read using the **NtGetContextThread** function:

NTSTATUS
NtGetContextThread(
    IN HANDLE *ThreadHandle,*
    IN OUT PCONTEXT *ThreadContext*
    );

<u>Parameters:</u>

ThreadHandle - An open handle to the thread object from which to retrieve context information.

ThreadContext - A pointer to the structure that will receive the user mode context of the specified thread. The initial value of the context flags field indicates the type and amount of context returned by this function.

The **NtGetContextThread** function is designed to facilitate the implementation of debuggers, and to allow subsystems to control the execution flow of their threads (e.g.; emulate signal delivery or APC delivery).

The **NtGetContextThread** function is absolutely NOT PORTABLE! The layout, contents, and length of the *PCONTEXT* structure depend on the processor and system architecture of the system servicing the **NtGetContextThread** function.

This service requires *THREAD_GET_CONTEXT* access to the specified thread.

The **NtGetContextThread** function is implemented by:

- Validating its arguments and translating the thread handle.
- Assuming everything is valid, it allocates a buffer for the thread's user-mode context. It then queues a special kernel-mode APC to the thread, and waits on an event located in the allocated buffer.
- When the APC executes, the thread dumps its user-mode context into the buffer and sets an event (located in the allocated buffer) indicating that the context dump is complete.

    *\ The APC is actually a special kernel mode APC, so that it can work even on a thread that is stuck in a suspend. *

- The target thread returns to whatever it was doing, and the thread calling **NtGetContextThread** copies the user-mode context from the allocated buffer into the thread context buffer passed in the system service. The allocated buffer is freed and the **NtGetContextThread** service completes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The specified thread does not need to be in a suspend state in order to call **NtGetContextThread** (subsystems and debuggers must explicitly do this if that is what is required). There is nothing to prevent a thread from calling **NtGetContextThread** on itself.

4.9 NtSetContextThread

A thread's user-mode machine context can be altered using the **NtSetContextThread** function:

NTSTATUS
NtSetContextThread(
    IN HANDLE *ThreadHandle,*
    IN OUT PCONTEXT *ThreadContext*
    );

Parameters:

ThreadHandle - An open handle to the thread whose context is to be set.

ThreadContext - A pointer to the structure that contains the user-mode context to be restored into the specified thread. The initial value of the context flags field indicates the type and amount of context that will be restored by this function.

The **NtSetContextThread** function is designed to facilitate the implementation of debuggers, and to allow subsystems to control the execution flow of their threads (e.g.; emulate signal delivery).

The **NtSetContextThread** function is absolutely NOT PORTABLE! The layout, contents, and length of the *PCONTEXT* structure depend on the processor and system architecture of the system servicing the **NtSetContextThread** function. Some fields of the *PCONTEXT* structure contain registers that contain both user-mode and kernel-mode context. Setting kernel-mode portions of these registers is not an error, but is ignored.

This service requires *THREAD_SET_CONTEXT* access to the specified thread.

The **NtSetContextThread** function is implemented by:

- Validating its arguments and translating the thread handle.
- Any kernel-mode only portions of fields in the *PCONTEXT* structure are set to a benign value.
- Assuming everything is valid, it allocates a buffer for the thread's user-mode context and copies the contents of the *ThreadContext* parameter into this buffer. It then queues a kernel-mode APC to the thread, and waits on an event located in the allocated buffer.
- When the APC executes, it writes the thread's user-mode context using the contents of the buffer and sets an event (located in the allocated buffer) indicating it is done with the buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

*\ The APC is actually a special kernel mode APC, so that it can work even on a thread that is stuck in a suspend. *

- o The target thread returns to whatever it was doing. When the target thread transitions into user-mode, its user-mode context will be restored using the context passed in during the call.
- o The thread calling **NtSetContextThread** frees the allocated buffer and completes the service.

The specified thread does not need to be in a suspend state in order to call **NtSetContextThread** (subsystems and debuggers must explicitly do this if that is what is required). There is also nothing that prevents the thread making the call to **NtSetContextThread** from being the target thread in the call.

4.10 NtOpenThread

A handle to a thread object can be created with the **NtOpenThread** function:

```
NTSTATUS
NtOpenThread(
    OUT PHANDLE ThreadHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN PCLIENT_ID ClientId OPTIONAL
    );
```

Parameters:

ThreadHandle - A pointer to a variable that will receive the thread object handle value.

DesiredAccess - The desired types of access to the opened thread. For a complete description of desired access flags, refer to the **NtCreateThread API** description.

ObjectAttributes - An pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a thread object.

ClientId - An optional parameter that if specified, supplies the client identifier of the thread to be opened. It is an error to specify this parameter along with an *ObjectAttributes* variable that contains a thread name.

Opening a thread object causes a new handle to be created. The access that the new handle has to the thread object is a function of the desired access and any *SECURITY_DESCRIPTOR* on the thread object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.11 NtQueryInformationThread

Selected information about a thread can be retrieved using the **NtQueryInformationThread** function.

NTSTATUS
NtQueryInformationThread(
IN HANDLE *ThreadHandle,*
IN THREADINFOCLASS *ThreadInformationClass*
OUT PVOID *ThreadInformation,*
IN ULONG *ThreadInformationLength,*
OUT PULONG *ReturnLength* **OPTIONAL**
);

Parameters:

ThreadHandle - An open handle to the thread object from which to retrieve information.

ThreadInformationClass - A variable that specifies the type of information to retrieve from the specified thread object.

ThreadInformationClass Values

ThreadBasicInformation - Returns the basic information about the specified thread. This information class value requires *THREAD_QUERY_INFORMATION* access to the thread.

ThreadTimes - Returns the cpu time usage of the specified thread. This information class requires *THREAD_QUERY_INFORMATION* access to the thread.

ThreadDescriptorTableEntry - Returns a descriptor from appropriate descriptor table for the thread. This information class will return a descriptor from either the Ldt, or the Gdt for the thread. This information class is only available on x86 processors, and returns STATUS_NOT_IMPLEMENTED on other processors. This information class requires *THREAD_QUERY_INFORMATION* access to the thread.

ThreadInformation - A pointer to a buffer that will receive information about the specified thread. The format and contents of the buffer depend on the specified information class being queried.

ThreadInformation Format by Information Class

ThreadBasicInformation - Data type is *PTHREAD_BASIC_INFORMATION.*

THREAD_BASIC_INFORMATION Structure

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG *ExitStatus* - Specifies the exit status of the thread. This field only contains meaningful information if the thread is in the signaled state; otherwise, it contains a value of "exit status pending".

PTEB *TebBaseAddress* - Specifies the virtual address of the thread's **TEB**.

CLIENT_ID *ClientId* - Specifies the thread's client identifier.

KPRIORITY *Priority* - Specifies the current priority of the thread.

KAFFINITY *AffinityMask* - Specifies the current processor affinity mask of the thread.

ThreadTimes - Data type is *PKERNEL_USER_TIMES*.

KERNEL_USER_TIMES Structure

TIME *UserTime* - Specifies the number of 100ns units that the thread has spent executing in user mode.

TIME *KernelTime* - Specifies the number of 100ns units that the thread has spent executing in kernel mode.

TIME *CreateTime* - Specifies the time that the thread was created.

TIME *ExitTime* - Specifies the time that the thread terminated.

ThreadDescriptorTableEntry - Data type is *PDESCRIPTOR_TABLE_ENTRY*

DESCRIPTOR_TABLE_ENTRY Structure

ULONG *Selector* - Specifies the number of the descriptor to return.

LDT_ENTRY *Descriptor* - Returns the descriptor contents.

ThreadInformationLength - Specifies the length in bytes of the thread information buffer (i.e.: the size of the information structure).

ReturnLength - An optional parameter that if specified, receives the number of bytes placed in thread information buffer.

4.12 NtSetInformationThread

Selected information can be set in a thread using the **NtSetInformationThread** function.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtSetInformationThread(
    IN HANDLE *ThreadHandle,*
    IN THREADINFOCLASS *ThreadInformationClass,*
    IN PVOID *ThreadInformation,*
    IN ULONG *ThreadInformationLength*
    );

Parameters:

ThreadHandle - A variable that specifies the handle to the thread to set information into.

ThreadInformationClass - A variable that specifies the type of information to set into the specified thread object.

ThreadInformationClass Values

ThreadPriority - Sets the priority of the specified thread. This information class value requires *THREAD_SET_INFORMATION* access to the thread.

ThreadAffinityMask - Sets the processor affinity mask of the specified thread. This information class requires *THREAD_SET_INFORMATION* access to the thread.

ThreadImpersonationToken - A handle to an impersonation token to be assigned as the impersonation token of the thread. This requires **THREAD_SET_THREAD_TOKEN** to the thread object and **TOKEN_IMPERSONATE** access to the token object. If the handle value is null, then any impersonation already in progress is discontinued.

ThreadInformation - A pointer to a buffer that contains the information to set in the specified thread. The format and contents of the buffer depend on the specified information class being queried.

ThreadInformation Format by Information Class

ThreadPriority - Data type is *PKPRIORITY*.

KPRIORITY *Priority* - Specifies the priority of the thread.

ThreadAffinityMask - Data type is *PKAFFINITY*.

KAFFINITY *AffinityMask* - Specifies the affinity mask assigned to the specified thread. The specified mask is anded with the process' default affinity mask and with the system wide affinity mask (which specifies the entire set of active processors in the system). The net effect is to limit a threads allowable affinity mask such that it is a subset of the maximum affinity mask in the current

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

configuration, and is also a subset of the affinity allowed to the process. Attempting to set an affinity that specifies no processors is an error condition.

ThreadImpersonationToken - Data type is *PHANDLE*. The handle value is that of an impersonation token, or may be null to indicate impersonation is to be discontinued.

ThreadInformationLength - Specifies the length in bytes of the thread information buffer.

4.13 NtImpersonateThread

Sets a server thread to be impersonating a client thread.

NTSTATUS
NtImpersonateThread(
    IN HANDLE *ServerThread*,
    IN HANDLE *ClientThread*
    );

Parameters:

ServerThread - A handle to the thread which is to be set to impersonate the client thread. This handle must be open for **THREAD_SET_THREAD_TOKEN** access.

ClientThread - A handle to the thread to be impersonated. This handle must be open for **THREAD_IMPERSONATE** access.

This service causes the thread specified by the *ServerThread* argument to impersonate the thread specified by the *ClientThread* argument. The impersonation will have the following security quality of service parameters:

- Delegation Level.
- Dynamic Tracking.
- Not EffectiveOnly.

4.14 NtAlertThread

A thread can be alerted with the **NtAlertThread** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtAlertThread(
    IN HANDLE *ThreadHandle*
    );

<u>Parameters:</u>

ThreadHandle - A handle to the thread to be alerted.

This function provides a mechanism that can be used to interrupt thread execution in the caller's previous mode (if this service is called from user mode the alert mode is user; otherwise, the alert mode is kernel) at well defined points.

Each thread has an alerted flag for each of the processor modes user and kernel. These flags are set by calling the **NtAlertThread** function.

If **NtAlertThread** is called and the target thread is in a wait state, then several additional tests are performed to determine the correct action to take.

If the mode of the wait is user (e.g. **NtWait** was called from user mode), and the alert mode is user, then a thread specific user mode **APC** is queued to the thread and the thread's wait will complete with a status of "alerted". When the **APC** executes it will raise the "alerted" condition.

If the mode of the wait is user or kernel, and the wait is alertable, then the thread's wait will complete with a status of "alerted".

If the target thread is not in a wait state, then the appropriate alerted bit in the target thread is set. Executing an **NtTestAlert**, or an alertable **NtWait** will clear the bit, return a status, and possibly queue a user mode **APC**.

This service requires *THREAD_ALERT* access to the specified thread.

4.15 NtTestAlert

A thread can test its alerted flag using the **NtTestAlert** function.

NTSTATUS
NtTestAlert();

The **NtTestAlert** function tests the calling thread's alerted flag for the thread's previous processor mode (i.e. if this function is called from user mode, the user mode alerted flag is tested; otherwise, the kernel mode alerted flag is tested). If the appropriate alerted flag is set, then the status value "alerted" is returned and the alerted flag is cleared; otherwise, a "normal" status value is returned. If the alerted flag was set and the previous mode is user, then a user **APC** is queued to the thread. When the **APC** executes, it will raise the "alerted" condition.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

In addition, **NtTestAlert** tests whether a user **APC** should be delivered. If the previous mode is user and the user **APC** queue contains an entry, then **APC** pending is set in the thread (this will cause an **APC** to be delivered to the thread on a transition from kernel mode into user mode).

4.16 NtAlertResumeThread

A thread can be alerted and resumed with the **NtAlertResumeThread** function:

NTSTATUS
NtAlertResumeThread(
    IN HANDLE *ThreadHandle,*
    OUT PULONG *PreviousSuspendCount* **OPTIONAL**
    );

Parameters:

ThreadHandle - A handle to the thread to be alerted and resumed.

PreviousSuspendCount - A pointer to the variable that receives the thread's previous suspend count.

Resuming and alerting a thread reverses the effects of a previous call to **NtSuspendThread** and causes the thread to be interrupted out of an alertable kernel mode wait with a status of "alerted". This function is provided to allow a subsystem to resume a thread and interrupt it out of an interruptible system service.

When an attempt is made to resume and alert a thread, the thread is alerted with a kernel mode alert, and its suspend count is examined. If the count is zero, then the service returns the suspend count. Otherwise, the count is decremented and if the count reaches zero, the thread resumes. In either case, the previous value of the thread's suspend count is returned. A non-zero value indicates that the thread was previously suspended. The value minus 1 specifies the number of calls to **NtResumeThread** that must be made in order to bring the thread out of the suspend state.

If the thread was waiting in a kernel mode alertable wait, its wait completes with a status of alerted.

This service requires *THREAD_SUSPEND_RESUME* and *THREAD_ALERT* access to the specified thread.

4.17 NtRegisterThreadTerminationPort

A thread can arrange for a port to be notified when it terminates using **NtRegisterThreadTerminationPort**.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtRegisterThreadTerminationPort(
    IN HANDLE *PortHandle*
    );

<u>Parameters:</u>

ULONG *PortHandle* - A handle to the port object that is to be notified when the subject thread terminates.

The **NtRegisterThreadTerminationPort** function is designed to allow a thread to specify a port object that is to be send a thread termination datagram when the subject thread terminates. Multiple calls to this service cause multiple ports to be notified when the thread terminates.

Each thread has a list of ports that are to be notified via a thread termination datagram when the thread terminates. When a thread terminates, the list is scanned and for each entry in the list, a thread termination datagram specifying the thread's client identifier and exit status is sent to the port. If during the send operation any errors occur (e.g. the port's connection was broken...) the system skips to the next entry in the list.

> *\ There is no need to provide this type of service at the process level since all of the process's port objects are closed during process termination. When a port object is closed (for the last time) its connections are broken, and the port that it was connected to is notified. *

The service is useful for subsystems that maintain per thread state (e.g. The Presentation Manager (**PM**) Subsystem). During the subsystem initialization that occurs in the client thread (e.g. calling **WinInitialize**), a call can be made to **NtRegisterThreadTerminationPort** specifying the port to the subsystem. When the thread terminates, the subsystem will receive a thread termination datagram. This datagram can be used as a signal to the subsystem that allows it to free up any thread specific resources.

Another use of this service is to allow a process to be notified when one of its own threads terminates. In order to do this, a multithreaded process creates a port to itself. A monitor thread monitors this port for thread termination datagrams. Each thread (in its startup routine) calls **NtRegisterThreadTerminationPort** specifying the port. Whenever a thread in the process terminates, the monitor thread is notified via the termination datagram. The monitor thread can use this event to perform appropriate actions.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.18 NtImpersonateThread

NTSTATUS
NtImpersonateThread(
  IN HANDLE *ServerThreadHandle*,
  IN HANDLE *ClientThreadHandle*,
  IN PSECURITY_QUALITY_OF_SERVICE *SecurityQos*
)

Arguments:

ServerThreadHandle - Is a handle to the server thread (the impersonator, or doing the impersonation). This handle must be open for THREAD_IMPERSONATE access.

ClientThreadHandle - Is a handle to the Client thread (the impersonatee, or one being impersonated). This handle must be open for THREAD_DIRECT_IMPERSONATION access.

SecurityQos - A pointer to security quality of service information indicating what form of impersonation is to be performed.

Return Value:

STATUS_SUCCESS - Indicates the call completed successfully.

Routine Description:

This routine is used to cause the server thread to impersonate the client thread. The impersonation is done according to the specified quality of service parameters.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. System Information API

The following programming interface provide support for querying information about the system:

NtQuerySystemInformation - Returns information about the system.

5.1 NtQuerySystemInformation

Information about the system can be retreived using the NtQuerySystemInformation system service.

NTSTATUS
NtQuerySystemInformation(
IN SYSTEM_INFORMATION_CLASS *SystemInformationClass,*
OUT PVOID *SystemInformation,*
IN ULONG *SystemInformationLength,*
OUT PULONG *ReturnLength* **OPTIONAL**
)

Parameters:

SystemInformationClass - The system information class about which to retrieve information.

SystemInformation - A pointer to a buffer which receives the specified information. The format and content of the buffer depend on the specified system information class.

SystemInformation Format by Information Class:

SystemBasicInformation - Data type is PSYSTEM_BASIC_INFORMATION

SYSTEM_BASIC_INFORMATION Structure

ULONG *OemMachineId* - An OEM specific bit pattern that identifies the machine configuration.

ULONG *TimerResolutionInMicroSeconds* - The resolution of the hardware time. All time values in Windows NT are specified as 64-bit LARGE_INTEGER values in units of 100 nanoseconds. This field allows an application to understand how many of the low order bits of a system time value are insignificant.

ULONG *PageSize* - The physical page size for virtual memory objects. Physical memory is committed in PageSize chunks.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG *AllocationGranularity* - The logical page size for virtual memory objects. Allocating 1 byte of virtual memory will actually allocate AllocationGranularity bytes of virtual memory. Storing into that byte will commit the first physical page of the virtual memory.

ULONG *MinimumUserModeAddress* - The smallest valid user mode address. The first AllocationGranullarity bytes of the virtual address space are reserved. This forces access violations for code the dereferences a zero pointer.

ULONG *MaximumUserModeAddress* - The largest valid used mode address. The next AllocationGranullarity bytes of the virtual address space are reserved. This allows system service routines to validate user mode pointer parameters quickly.

KAFFINITY *ActiveProcessorsAffinityMask* - The system wide affinity mask that specifies the set of processors configured into the system. This set represents the maximum allowable affinity of any thread within the system.

CCHAR *NumberOfProcessors* - The number of processors in the current hardware configuration.

SystemProcessorInformation - Data type is SYSTEM_PROCESSOR_INFORMATION

SYSTEM_PROCESSOR_INFORMATION Structure

ULONG *ProcessorType* - The processor type.

ProcessorType Values:

PROCESSOR_INTEL_386

PROCESSOR_INTEL_486

PROCESSOR_INTEL_860

PROCESSOR_MIPS_R2000

PROCESSOR_MIPS_R3000

PROCESSOR_MIPS_R4000

ULONG *ProcessorStepping* - The processor stepping. The high order 16 bits specify the stepping letter (0==A, 1==B, etc.) and the low order 16 bits specify the stepping level (e.g. 0, 1, 2, etc.).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ULONG *ProcessorOptions* - Flags that specify processor options that may or may not be present. The flags are processor specific.

ProcessOptions flags for PROCESSOR_INTEL_386:

PROCESSOR_OPTION_387 - A 387 co-processor chip is present.

PROCESSOR_OPTION_WEITEK - A Weitek floating pointer co-processor chip is present.

SystemInformationLength - Specifies the length in bytes of the system information buffer.

ReturnLength - An optional pointer which, if specified, receives the number of bytes placed in the system information buffer.

Return Value:

NTSTATUS - *STATUS_SUCCESS* if the operation is successful and an appropriate error value otherwise.

The following status values may be returned by the function:

- *STATUS_SUCCESS* - successful completion.
- *STATUS_INVALID_INFO_CLASS* - The *SystemInformationClass* parameter did not specify a valid value.
- *STATUS_INFO_LENGTH_MISMATCH* - The value of the *SystemInformationLength* parameter did not match the length required for the information class requested by the *SystemInformationClass* parameter.
- *STATUS_ACCESS_VIOLATION* - Either the *SystemInformation* buffer pointer or the *ReturnLength* pointer value specified an invalid address.

6. Executive APIs

The following programming interfaces are available from within the **Windows NT** executive:

PsCreateSystemProcess - Creates a system process.
PsCreateSystemThread - Creates a system thread.
PsLookupProcessThreadByCid - Locates the process and thread using the specified **CID**.
PsChargePoolQuota - Charges pool quota to the specified process.
PsReturnPoolQuota - Returns pool quota to the specified process.
PsGetCurrentThread - Returns the address of the currently executing thread's thread object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PsGetCurrentProcess - Returns the address of the process object that the currently executing thread is attached to.
ExGetPreviousMode - Returns the processor mode that the thread was executing in prior to the last trap.
PsRevertToSelf - Reverts the calling thread's access token to its original value.
PsReferencePrimaryToken - This function returns a pointer to the primary token of a process. The reference count of that primary token is incremented to protect the pointer returned.
PsDereferencePrimaryToken - This function releases a pointer to a primary token obtained using **PsReferencePrimaryToken()**.
PsReferenceImpersonationToken - This function returns a pointer to the impersonation token of a thread. The reference count of that impersonation token is incremented to protect the pointer returned.
PsDereferenceImpersonationToken - This function releases a pointer to a primary token obtained using **PsReferenceImpersonationToken()**.
PsOpenTokenOfProcess - This function does the thread specific processing of an **NtOpenThreadToken()** service.
PsOpenTokenOfThread - This function does the thread specific processing of an **NtOpenThreadToken()** service.
PsImpersonateClient -This routine sets up the specified thread so that it is impersonating the specified client.

6.1 PsCreateSystemProcess

A system process can be created using **PsCreateSystemProcess**.

NTSTATUS
PsCreateSystemProcess(
    OUT HANDLE *ProcessHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL***,*
    );

Parameters:

ProcessHandle - A pointer to a variable that will receive the process object handle value.

DesiredAccess - The desired types of access to the created process. For a complete description of desired access flags, refer to the **NtCreateProcess API** description.

ObjectAttributes - An pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a process object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Creating a system process creates a process object whose address space is initialized so that the "user" portion of the address space is empty, and the "system" portion of the address space maps the system. This option is not available from user-mode via **NtCreateProcess**. The process inherits its access token and quotas from the initial system process. It is created with an empty handle table. The process's debug and exception ports are **NULL**.

The system does not treat a process created through this API any differently than any other process. Any **Windows NT** API that requires a handle to a process object may specify a process created through this API.

6.2 PsCreateSystemThread

A system thread that executes in kernel mode can be created and a handle opened for access to the thread with the **PsCreateSystemThread** function:

NTSTATUS
PsCreateSystemThread(
    OUT PHANDLE *ThreadHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL***,*
    IN HANDLE *ProcessHandle* **OPTIONAL***,*
    OUT PCLIENT_ID *ClientId* **OPTIONAL***,*
    IN PKSTART_ROUTINE *StartRoutine,*
    IN PVOID *StartContext*
    );

Parameters:

ThreadHandle - A pointer to a variable that will receive the thread object handle value.

DesiredAccess - The desired types of access to the created thread. For a complete description of desired access flags, refer to the **NtCreateThread API** description.

ObjectAttributes - An pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a thread object.

ProcessHandle - An open handle to the process object that the thread is to run in. The subject thread must have *PROCESS_CREATE_THREAD* access to this process. If this parameter is not supplied, then the thread will be created in the initial system process.

ClientId - A pointer to a structure that will receive the client identifier of the new thread.

StartRoutine - Supplies the address of a function in system space that the thread begins execution at. A return from this function causes the thread to terminate.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

StartContext - Supplies a single argument passed to the thread when it begins execution.

Creating a system thread begins a separate thread of execution within the system. System threads may only execute in kernel-mode. A system thread has no **TEB**, or user-mode context. It is not possible to terminate a system thread using **NtTerminateThread** unless the thread is terminating itself.

6.3 PsLookupProcessThreadByCid

A process and thread can be located by client id using the **PsLookupProcessThreadByCid** function:

NTSTATUS
PsLookupProcessThreadByCid(
    IN PCID *Cid,*
    OUT PEPROCESS *Process* **OPTIONAL,**
    OUT PETHREAD *Thread*
    );

Parameters:

Cid - A pointer to the client id whose thread and process are to be located.

Process - An optional parameter that if specified receives a referenced pointer to the process object associated with the specified client id.

Thread - A parameter that receives a referenced pointer to the thread object associated with the specified client id.

6.4 PsChargePoolQuota

Pool quota can be charged to the specified process using the **PsChargePoolQuota** function:

VOID
PsChargePoolQuota(
    IN PEPROCESS *Process,*
    IN POOL_TYPE *PoolType*,
    IN ULONG *Amount*
    );

Parameters:

Process - Supplies the address of a process to charge pool quota to.

PoolType - Supplies the pool type to charge the quota for.

Amount - Supplies the amount of quota to charge to the process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The **PsChargePoolQuota** function is designed to charge pool quota to a process subject to the quota limits of that process. If the quota charge would cause the process to exceed its quota limit for the specified pool type, then an *STATUS_QUOTA_EXCEEDED* exception is raised and the charge is not made. Otherwise, the quota pool usage of the specified process is adjusted (incremented) to account for the quota being charged to the process.

6.5 PsReturnPoolQuota

Pool quota can be returned to the specified process using the **PsReturnPoolQuota** function:

VOID
PsReturnPoolQuota(
    IN PEPROCESS *Process,*
    IN POOL_TYPE *PoolType*,
    IN ULONG *Amount*
    );

Parameters:

Process - Supplies the address of a process to return pool quota to.

PoolType - Supplies the pool type to return the quota for.

Amount - Supplies the amount of quota to return to the process.

The **PsReturnPoolQuota** function is designed to return pool quota to a process to reverse the effects of a previous call to **PsChargePoolQuota**. The system will catch attemps to return more quota to the process than the process has been charged for and bug check. Otherwise, the quota pool usage of the specified process is adjusted (decremented) to account for the quota being returned to the process.

6.6 PsGetCurrentThread

The address of the thread object of the currently executing thread is returned using the **GetCurrentThread** function:

PETHREAD
PsGetCurrentThread();

6.7 PsGetCurrentProcess

The address of the process object that the currently executing thread is attached to is returned using the **PsGetCurrentProcess** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PEPROCESS
PsGetCurrentProcess();

6.8 KeGetPreviousMode

The processor mode that the current thread was running in prior to the last trap or interrupt can be determined using the **KeGetPreviousMode** function:

KPROCESSOR_MODE
KeGetPreviousMode();

The **KeGetPreviousMode** function is used mainly inside **Windows NT** system services to determine the processor mode that the thread was executing in prior to the system service.

6.9 PsRevertToSelf

The current can switch to its original access token using the **PsRevertToSelf** function:

VOID
PsRevertToSelf();

The **PsRevertToSelf** function switches the access token used by the calling thread back to its original value. This is the same token that would have been in effect if the thread had never called **PsImpersonateThread**.

6.10 PsReferencePrimaryToken

PACCESS_TOKEN
PsReferencePrimaryToken(
  IN PEPROCESS *Process*
)

Arguments:

Process - Supplies the address of the process whose primary token is to be referenced.

Return Value:

A pointer to the specified process's primary token.

Routine Description:

This function returns a pointer to the primary token of a process. The reference count of that primary token is incremented to protect the pointer returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

When the pointer is no longer needed, it should be freed using **PsDereferencePrimaryToken()**.

6.11 PsDereferencePrimaryToken

VOID
PsDereferencePrimaryToken(
  IN PACCESS_TOKEN *PrimaryToken*
)

Arguments:

PrimaryToken - Pointer to a token obtained using **PsReferencePrimaryToken()**.

Return Value:

None.

Routine Description:

This function causes the referenced primary token to be dereferenced. This token is expected to have been referenced using **PsReferencePrimaryToken()**.

6.12 PsReferenceImpersonationToken

PACCESS_TOKEN
PsReferenceImpersonationToken(
  IN PETHREAD *Thread*,
  OUT PBOOLEAN *CopyOnOpen*,
  OUT PBOOLEAN *EffectiveOnly*,
  OUT PSECURITY_IMPERSONATION_LEVEL *ImpersonationLevel*,
)

Arguments:

Thread - Supplies the address of the thread whose impersonation token is to be referenced.

CopyOnOpen - The current value of the Thread->CopyOnOpen field.

EffectiveOnly - The current value of the Thread->EffectiveOnly field.

ImpersonationLevel - The current value of the Thread->ImpersonationLevel field.

Return Value:

A pointer to the specified thread's impersonation token.

If the thread is not currently impersonating a client, then NULL is returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Routine Description:

This function returns a pointer to the impersonation token of a thread. The reference count of that impersonation token is incremented to protect the pointer returned.

If the thread is not currently impersonating a client, then a null pointer is returned.

If the thread is impersonating a client, then information about the means of impersonation are also returned (ImpersonationLevel).

If a non-null value is returned, then **PsDereferenceImpersonationToken()** must be called to decrement the token's reference count when the pointer is no longer needed.

6.13 PsDereferenceImpersonationToken

VOID
PsDereferenceImpersonationToken(
   IN PACCESS_TOKEN *ImpersonationToken*
)

Arguments:

ImpersonationToken - Pointer to a token obtained using **PsReferenceImpersonationToken**().

Return Value:

None.

Routine Description:

This function causes the referenced impersonation token to be dereferenced. This token is expected to have been referenced using **PsReferenceImpersonationToken()**.

6.14 PsOpenTokenOfProcess

NTSTATUS
PsOpenTokenOfProcess(
   IN HANDLE *ProcessHandle*,
   OUT PACCESS_TOKEN **Token*
)

Arguments:

ProcessHandle - Supplies a handle to a process object whose primary token is to be opened.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Token - If successful, receives a pointer to the process's token object.

Return Value:

STATUS_SUCCESS - Indicates the call completed successfully.

Status may also be any value returned by an attemp the reference the process object for PROCESS_QUERY_INFORMATION access.

Routine Description:

This function does the process specific processing of an NtOpenProcessToken() service.

The service validates that the handle has appropriate access to referenced process. If so, it goes on to reference the primary token object to prevent it from going away while the rest of the NtOpenProcessToken() request is processed.

NOTE: If this call completes successfully, the caller is responsible for decrementing the reference count of the target token. This must be done using the PsDereferencePrimaryToken() API.

6.15 PsOpenTokenOfThread

NTSTATUS
PsOpenTokenOfThread(
  IN HANDLE *ThreadHandle*,
  OUT PACCESS_TOKEN *Token*,
  OUT PBOOLEAN CopyOnOpen,
  OUT PBOOLEAN *EffectiveOnly*,
  OUT PSECURITY_IMPERSONATION_LEVEL *ImpersonationLevel*
)

Arguments:

ThreadHandle - Supplies a handle to a thread object.

Token - If successful, receives a pointer to the thread's token object.

CopyOnOpen - The current value of the Thread->CopyOnOpen field.

EffectiveOnly - The current value of the Thread->EffectiveOnly field.

ImpersonationLevel - The current value of the Thread->ImpersonationLevel field.

Return Value:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

STATUS_SUCCESS - Indicates the call completed successfully.

STATUS_NO_TOKEN - Indicates the referenced thread is not currently impersonating a client.

STATUS_CANT_OPEN_ANONYMOUS - Indicates the client requested anonymous impersonation level. An anonymous token can not be openned.

status may also be any value returned by an attemp the reference the thread object for THREAD_QUERY_INFORMATION access.

Routine Description:

This function does the thread specific processing of an **NtOpenThreadToken()** service.

The service validates that the handle has appropriate access to reference the thread. If so, it goes on to increment the reference count of the token object to prevent it from going away while the rest of the NtOpenThreadToken() request is processed.

NOTE: If this call completes successfully, the caller is responsible for decrementing the reference count of the target token. This must be done using PsDereferenceImpersonationToken().

6.16 PsImpersonateClient

VOID
PsImpersonateClient(
  IN PETHREAD *Thread*,
  IN BOOLEAN *CopyOnOpen*,
  IN BOOLEAN *EffectiveOnly*,
  IN SECURITY_IMPERSONATION_LEVEL *ImpersonationLevel*
)

Arguments:

Thread - points to the thread which is going to impersonate a client.

CopyOnOpen - If TRUE, indicates the token is considered to be private by the assigner and should be copied if opened. For example, a session layer may be using a token to represent a client's context. If the session is trying to synchronize the context of the client, then user mode code should not be given direct access to the session layer's token.

This field is ANDed with the DirectAccess field of the ClientContext to establish the CopyOnOpen value actually assigned to the impersonation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

CopyOnOpen - If TRUE, indicates the token is considered to be private by the assigner and should be copied if opened. For example, a session layer may be using a token to represent a client's context. If the session is trying to synchronize the context of the client, then user mode code should not be given direct access to the session layer's token.

Basically, session layers should always specify TRUE for this, while tokens assigned by the server itself (handle based) should specify FALSE.

EffectiveOnly - Is a boolean value to be assigned as the Thread->EffectiveOnly field value for the impersonation. A value of FALSE indicates the server is allowed to enable currently disabled groups and privileges.

ImpersonationLevel - Is the impersonation level that the server is allowed to access the token with.

Return Value:

STATUS_SUCCESS - Indicates the call completed successfully.

Routine Description:

This routine sets up the specified thread so that it is impersonating the specified client. This will result in the reference count of the token representing the client being incremented to reflect the new reference.

If the thread is currently impersonating a client, that token will be dereferenced.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Revision 1.2

1. Simplify Create Thread.
2. Remove create if, permanent, and other object options that are only there for orthoganality.
3. Add port notification handlers.
4. Add 32 bit exit status for process and thread termination.
5. Add NtAlertThread/NtAlertResumeThread.
6. Add get thread info.
7. Add debugger port and subsystem port to process creation.
8. Add process get/set info place holders.

Revision 1.3

1. Complicate create thread
2. Reorganize considerations

Revision 1.15, August 20, 1990, Jim Kelly

1. Eliminated previous token query information levels. This is done using NtOpenProcessToken() and NtOpenThreadToken().
2. Added information level allowing the setting of a primary token.
3. Added PsReferenceImpersonationToken() and PsDereferenceImpersonationToken().
4. Added PsReferencePrimaryToken() and PsDereferencePrimaryToken().
5. Added PsImpersonateClient().
6. Added PsOpenTokenOfProcess() and PsOpenTokenOfThread().
7. Eliminated PsLockToken(), PsUnlockToken(), and PsImpersonateThread().
8. Minor grammatical and spelling corrections.
9. Removed *TokenLength* field from THREAD_BASIC_INFORMATION structure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision 1.22, February 7, 1991, Jim Kelly.

1. Changed THREAD_IMPERSONATE_CLIENT access type to be THREAD_SET_THREAD_TOKEN.

2. Added the ability to directly impersonate a thread. This resulted in a new API (**NtImpersonateThread()**) and a new access type (**THREAD_IMPERSONATE**).

3. Corrected minor typos.

Revision 1.24, February 28, 1991, Mark Lucovsky.

1) ???

Revision 1.24, April 21, 1991, Jim Kelly (JimK).

1. Added NtImpersonateThread() service.

Revision 1.25, May 2, 1991, Bryan Willman (bryanwi).

1. Added ProcessLdtInformation and ProcessLdtSize to set of data types for NtQueryInformationProcess and NtSetInformationProcess.

Revision 1.26, May 24, 1991, Dave Hastings (daveh).

1. Added ThreadDescriptorTableEntry to NtQueryInformationThread.

2. Allowed querying of specific regions of the LDT for ProcessLdtInformation.

Revision 1.27, January 14, 1992, Jim Kelly (JimK).

1. Eliminated PROCESS_SET_ACCESS_TOKEN as an access type. Changing the primary token of a process will be protected by PROCESS_SET_INFORMATION followed by a privilege test at the time the change is requested (rather than at open time).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Shared Resource Specification

Author: *Gary D. Kimura*

Revision 1.5, May 15, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction 1
2. Initializing a Resource Variable 1
3. Acquiring a Resource for Shared Access 2
4. Acquiring a Resource for Exclusive Access 2
5. Releasing a Resource 2
6. Changing from Shared Access to Exclusive Access 3
7. Changing from Shared Access to Exclusive Access 3
8. Deleting a Resource Variable 4

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the **Windows NT** routines that implement multiple-readers, single-writer access to a shared resource. Access is controlled via a shared resource variable and a set of routines to acquire the resource for shared access (also commonly known as read access) or to acquire the resource for exclusive access (also called write access).

A resource is logically in one of three states:

- o Acquired for shared access
- o Acquired for exclusive access
- o Released (i.e., not acquired for shared or exclusive access)

Initially a resource is in the released state, and can be acquired for either shared or exclusive access by a user.

A resource that is acquired for shared access can be acquired by other users for shared access. The resource stays in the acquired for shared access state until all users that have acquired it have released the resource, and then it becomes released. Each resource, internally, maintains a count of the number of users granted shared access.

A resource that is acquired for exclusive access cannot be acquired by other users until the single user that has acquired the resource for exclusive access releases the resource. However, a thread can recursively acquire exclusive access to the same resource without blocking.

The routines described in this specification do not return to the caller until the resource has been acquired. or the user has the option of having the procedure return immediately to the caller if the resource cannot be acquired with blocking. The procedure's return value then indicates if the resource has been acquired.

To help avoid starvation of a user requesting exclusive access to a resource, the procedures do not allow additional users shared access to a resource if there is a user waiting for exclusive access to the resource.

Also, when a user releases exclusive access to a resource, priority is given to those waiting for exclusive access over those waiting for shared access.

The **APIs** that implement a shared resource are the following:

ExInitializeResource - Initialize a resource and set its state to released

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ExAcquireResourceShared - Acquire shared access to a resource
ExAcquireResourceExclusive - Acquire exclusive access to a resource
ExReleaseResource - Release a resource (shared or exclusive)
ExConvertSharedToExclusive - Convert from shared to exclusive access
ExConvertExclusiveToShared - Convert from exclusive to shared access
ExDeleteResource - Deletes (i.e., uninitializes) a resource

2. Initializing a Resource Variable

A resource variable can be initialized and its state set to released with the **ExInitializeResource** procedure.

VOID
ExInitializeResource (
    IN PERESOURCE *Resource*
    );

Parameters:

Resource – A pointer to the resource variable being initialized

A resource variable cannot be used by the other procedures until it has been initialized.

3. Acquiring a Resource for Shared Access

A user can acquire shared access to a resource with the **ExAcquireResourceShared** procedure.

BOOLEAN
ExAcquireResourceShared (
    IN PERESOURCE *Resource*,
    IN BOOLEAN *Wait*
    );

Parameters:

Resource - A pointer to the resource variable to be acquired for shared access

Wait - Indicates if the call is allowed to block in order to acquire the resource. A value of TRUE indicates that the call is allowed to wait for the resource and FALSE indicates that control is to return immediately even if the resource cannot be acquired.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Return Value:

BOOLEAN - Returns TRUE if the access to the resource has been acquired and FALSE otherwise. If the Wait input parameter is TRUE then this function will always return a value of TRUE. If the Wait input parameter is FALSE then this function will return TRUE if the resource was acquired without blocking and FALSE if the resource cannot be acquired without blocking.

If the Wait parameter is TRUE, then this procedure does not return to the caller until the resource has been acquired for shared access. When the user acquires the resource, the count of the number of shared access users is incremented by one.

If the caller's thread has previously acquired exclusive access to the resource then the call to **ExAcquireResourceShared** will automatically succeed.

4. Acquiring a Resource for Exclusive Access

A user can acquire exclusive access to a resource with the **ExAcquireResourceExclusive** procedure.

BOOLEAN
ExAcquireResourceExclusive (
  IN PERESOURCE *Resource*,
  IN BOOLEAN *Wait*
  );

Parameters:

Resource - A pointer to the resource variable to be acquired for exclusive access

Wait - Indicates if the call is allowed to block in order to acquire the resource. A value of TRUE indicates that the call is allowed to wait for the resource and FALSE indicates that control is to return immediately even if the resource cannot be acquired.

Return Value:

BOOLEAN - Returns TRUE if the access to the resource has been acquired and FALSE otherwise. If the Wait input parameter is TRUE then this function will always return a value of TRUE. If the Wait input parameter is FALSE then this function will return TRUE if the resource was acquired without blocking and FALSE if the resource cannot be acquired without blocking.

If the Wait parameter is TRUE, then this procedure does not return to the caller until the resource has been acquired for exclusive access.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the caller's thread has previously acquired exclusive access to the resource then subsequent calls to **ExAcquireResourceExclusive** will automatically succeed.

5. Releasing a Resource

A user can release access to a resource with the **ExReleaseResource** procedure. This procedure is for releasing either exclusive access or shared access to a resource.

```
VOID
ExReleaseResource (
    IN PERESOURCE Resource
    );
```

Parameters:

Resource - A pointer to the resource variable to be released

If the resource is acquired for shared access, the number of users with shared access to the resource is decremented by one. If the count is now zero, the resource is released and next user waiting for exclusive access to the resource acquires it.

If the resource is acquired for exclusive access then the count of the number of times it has been recursively acquired is decremented by one. If the count is now zero, the resource is released and the next user waiting for access to the resource acquires it.

6. Changing from Shared Access to Exclusive Access

A user that has shared access to a resource can change to exclusive access with the **ExConvertSharedToExclusive** procedure.

```
VOID
ExConvertSharedToExclusive (
    IN PERESOURCE Resource
    );
```

Parameters:

Resource - A pointer to the resource variable to be acquired for exclusive access

This procedure does not return to the caller until the resource has been acquired for exclusive access. This procedure is similar in function to releasing a shared resource and then acquiring it for exclusive access; however, in the case where only

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

one user has the resource acquired with shared access, the conversion to exclusive access with **ExConvertSharedToExclusive** is more efficient.

It is an error to try to covert a resource to exclusive access that is not currently acquired for either shared or exclusive access.

7. Changing from Shared Access to Exclusive Access

A user that has exclusive access to a resource can change to shared access with the **ExConvertExclusiveToShared** procedure.

VOID
ExConvertExclusiveToShared (
    IN PERESOURCE *Resource*
    );

<u>Parameters:</u>

Resource - A pointer to the resource variable to be converted to shared access

This procedure does not return to the caller until the resource has been acquired for shared access. This procedure is similar in function to releasing an exclusive resource and then acquiring it for shared access; however the user calling **ExConvertExclusiveToShared** does not relinquish access to the resource as the two step operation does.

It is an error to try to convert a resource to shared access that is not currently acquired for exclusive access, or has been acquired recusively.

8. Deleting a Resource Variable

A resource variable can be deleted (i.e., uninitialized) with the **ExDeleteResource** procedure.

VOID
ExDeleteResource (
    IN PERESOURCE *Resource*
    );

<u>Parameters:</u>

Resource – A pointer to the resource variable being deleted

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The programmer is responsible for ensuring that no one is actively using the resource. After calling this procedure the resource cannot be used again unless it is reinitialized.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, June 23, 1989

Revision 1.1, June 26, 1989

1. Added explanation regarding who gets priority when releasing a resource

2. Changes **PEXRESOURCE** to **PERESOURCE**

Revision 1.2, June 27, 1989

1. Changed **ExChangeSharedToExclusive** to **ExConvertSharedToExclusive**.

2. Added **ExConvertExclusiveToShared**.

Revision 1.3, July 10, 1989

1. Added **ExDeleteResource**

Revision 1.4, March 21, 1990

1. Changed **ExAcquireResourceShared** and **ExAcquireResourceExclusive** to take an additional Wait input parameter and to return a BOOLEAN function value.

Revision 1.5, May 15, 1990

1. Added recusive exclusive resource acquisition.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Session Management and Control

Author: *Mark Lucovsky*

Revision 1.9, January 7, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction 2
  1.1 NT Sessions 2
  1.3 Windows NT System Structure 4

2. General Sm Services 8
  2.1 SmConnectToSm 8
  2.2 SmGetLogonObjectDirectory 8

3. Logon Process Support 9
  3.1 Logon Process Philosophy 9
  3.2 SmRegisterLogonProcess 12
  3.3 SmExecLogonShell 13

4. System Subsystems Support 15
  4.1 Session Control Services 15
    4.1.1 SmCreateForeignSession 15
    4.1.2 SmSessionComplete 16
    4.1.3 SmTerminateForeignSession 17
  4.2 Piper 18
    4.2.1 PiperCreatePipe 18
    4.2.2 PiperJoinPipe 19
    4.2.3 PiperLeavePipe 20
    4.2.4 PiperReadPipe 20
    4.2.5 PiperWritePipe 20

5. Emulation Subsystems 22
  5.1 PSX++ 22
  5.2 OS/2++ 22
  5.3 NT++ 23
  5.4 Emulation Subsystem APIs 23
    5.4.1 SbCreateSession 24
    5.4.2 SbTerminateSession 25
    5.4.3 SbForeignSessionComplete 26

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

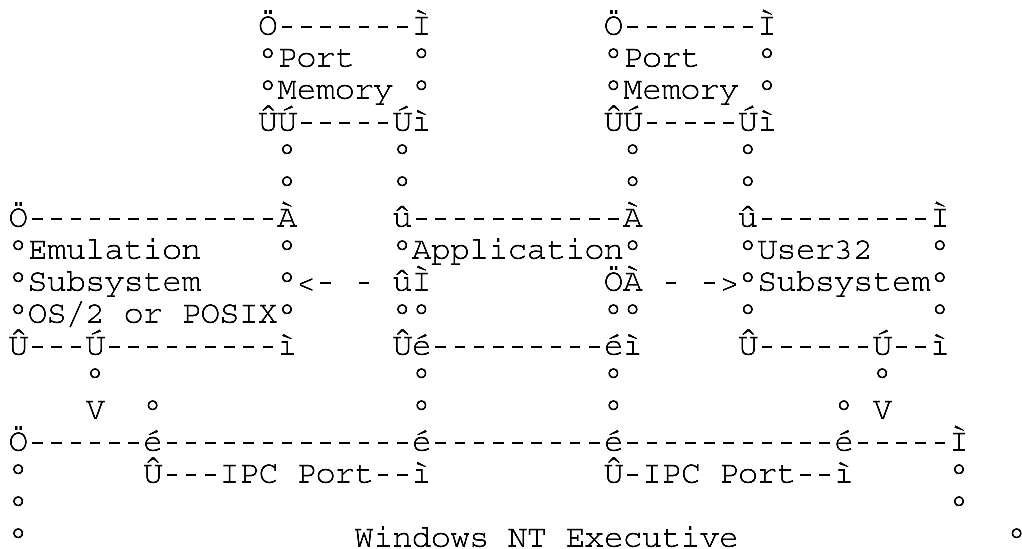
1. Introduction

The **Windows NT** operating system is designed to support multiple concurrent application execution environments. The initial application execution environments that will be supported under **Windows NT** include **POSIX** (*IEEE Std 1003.1-1988*), and **32-Bit Cruiser OS/2**.

Users will see **Windows NT** as a system that lets them execute both **POSIX** and **OS/2** applications concurrently. There is no need to reboot the system to gain access to a particular execution environment.

Multiple concurrent application execution environments are made possible by implementing these environments as *Emulation Subsystems*. An *Emulation Subsystem* implements the APIs of a given operating system as a protected subsystem. Each application program image file header contains a description of the operating system environment that it has been designed to run in (e.g., **cmd.exe** is marked as an **OS/2** application and **ed** is marked as a **POSIX** application). During the process initialization of an application, an *LPC* connection is made between the application and the *Emulation Subsystem* that it has been designed to run with. Each system service API call that the application makes is translated into a *Local Procedure Call* (*LPC*) to the *Emulation Subsystem*. The subsystem implements the respective APIs using native **Windows NT** services.

The structure of an application program with respect to an *Emulation Subsystem* and the Native **Windows NT** System Services is depicted below.



Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.1 NT Sessions

Windows NT provides a mechanism that allows an application in one environment to execute an application designed to run in another environment. For example, the **OS/2** command line interpreter **cmd.exe** can start the **POSIX** editor **ed** as follows.

- **cmd.exe**, an **OS/2** application calls **DosExecPgm** passing it the program name **ed**.- The **OS/2** subsystem creates a process ready to execute the **ed** program.

- After creating the process, the image type is examined.

- Since the image type indicates that it is not an **OS/2** application, the **OS/2** subsystem issues an *LPC* to *Sm* asking it to forward the process off to an appropriate *Emulation Subsystem*. *Sm* exports an API named **SmCreateForeignSession** that performs this function.

- *Sm* examines the image type passed as part of the **SmCreateForeignSession** call. The image type indicates that **ed** is a **POSIX** application.

- *Sm* issues an *LPC* to the **POSIX** subsystem passing it the process (originally created by the **OS/2** subsystem). Each *Emulation Subsystem* exports an API named **SbCreateSession** that performs this function.

- When the **ed** application terminates, the **POSIX** subsystem issues an *LPC* to *Sm* indicating that the process has completed with the specified termination status. *Sm* exports an API named **SmSessionComplete** that performs this function.

- Upon receipt of the call, *Sm* issues an *LPC* to the **OS/2** subsystem indicating that **ed** has terminated with the specified termination status. Each *Emulation Subsystem* exports an API named **SbForeignSessionComplete** that performs this function.

In addition to starting an application in a different environment, **Windows NT** allows an application in one environment to pass information through a pipe stream to a process in another environment. The *Pipe Stream Subsystem* (*Piper*) exports a set of APIs used by *Emulation Subsystems* that make this possible.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.2 NT Logon Sessions

To tie all related NT sessions together, a *logon session* is used. A logon session serves as a parent to all sessions related to a single logon.

Associated with a logon session, and all the sessions related to it, is an object directory refered to as the *Logon Object Directory.* This object directory may be used to house objects related to processes related to all sessions of the logon session. The name of the logon object directory may be obtained using the **SmGetLogonObjectDirectory()** service.

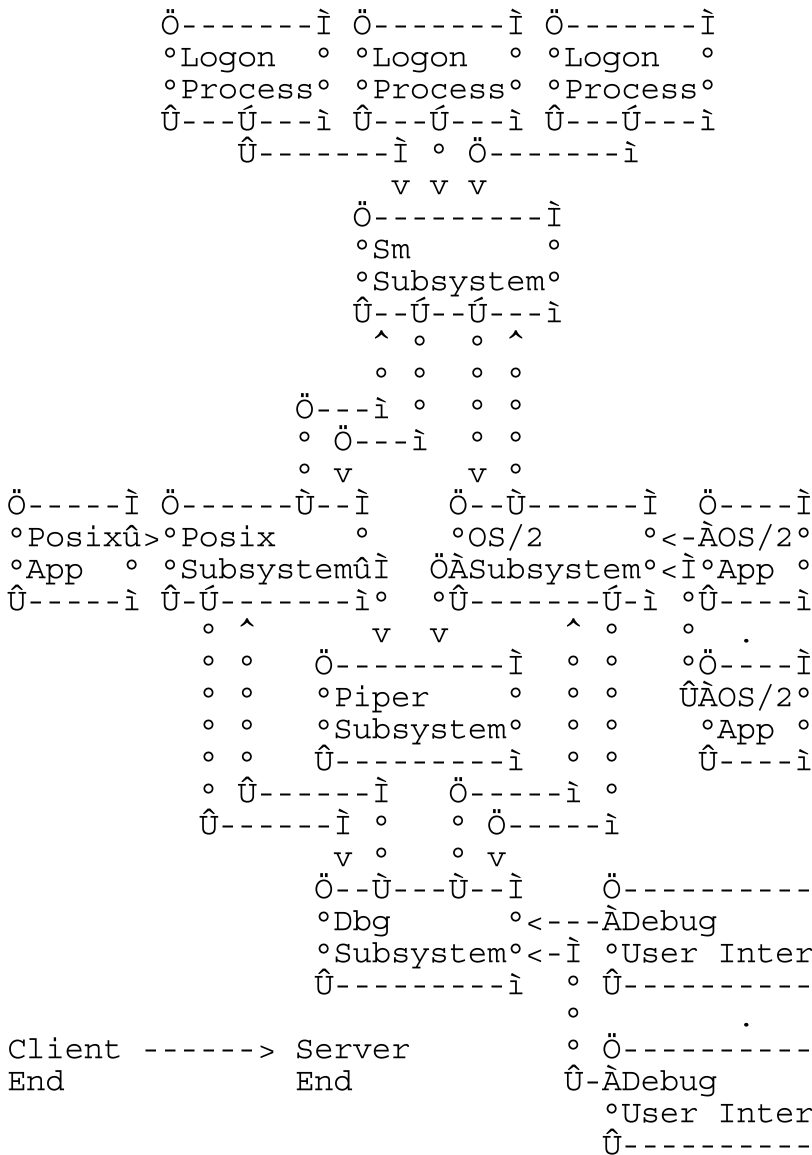
Throughout this document, the term *session* typically referes to an NT session. When a higher level *logon session* is being refered to, it will explicitly be called out as a logon session.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.3 Windows NT System Structure

Before going any further, the following diagram is presented to show the overall structure of the subsystems and system processes that implement the session management and control portion of the **Windows NT** operating system.

Windows NT System Structure



Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The above diagram shows the structure of a **Windows NT** system. Most of the structure is static and is created at system boot time. The purpose of each component is described below.

Logon Processes - A logon process is created for each class of devices that can accept and process logon requests. Each logon process exists as a client process served by *Sm*. The *LPC* connection between a logon process and *Sm* is trusted and relatively static (created when each logon process initializes). A logon process is responsible for detecting logon requests from the devices it manages, authenticating the user (using the Local Security Authority), and calling *Sm* to activate the logon shell for the newly logged on user.

Sm Subsystem - The *Sm* subsystem is created during system initialization as the initial user mode process. It is responsible for building the structure presented in the above diagram. After the structure is built, *Sm* acts as the system session manager. In this role it is responsible for activating new logon shell programs and for fielding process creation requests from the various *Emulation Subsystem* and forwarding them on to the appropriate *Emulation Subsystem*.

This occurs when a subsystem is instructed to execute a program image, and the image file header describes an image designed to run in a different environment. *Sm* acts as a server to both logon processes and *Emulation Subsystems*.

As a server, *Sm* exports the following APIs over a trusted *LPC* connection between an *Emulation Subsystem* and itself:

o - **SmConnectToSm** - Called by an *Emulation Subsystem* to create an *LPC* connection to *Sm*.

o - **SmCreateForeignSession** - Called by an *Emulation Subsystem* when it detects an image file designed to execute in a different environment.

o - **SmTerminateForeignSession** - Called by an *Emulation Subsystem* when it wants to terminate a session that it has asked *Sm* to create.

o - **SmSessionComplete** - Called by an *Emulation Subsystem* when a session it has been asked to create completes.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o - **SmGetLogonObjectDirectory** - Called by an *Emulation Subsystem* to determine the logon object directory associated with a session.

As a server, *Sm* exports the following APIs over a trusted *LPC* connection between a *Logon Process* and itself:

- o - **SmConnectToSm** - Called by an *Logon Process* to create an *LPC* connection to *Sm*.

- o - **SmRegisterLogonProcess** - Called by a *Logon Process* to identify itself as a logon process. This is called after connecting to *Sm* using **SmConnectToSm**.

- o - **SmExecLogonShell** - Called by a *Logon Process* to activate a user interface shell program for a new interactive logon session. This is used after the user has been authenticated, and a token obtained from the Local Security Authority.

Sm acts as a client of the *Emulation Subsystems*. As a client, *Sm* makes the following API calls over trusted *LPC* connections between an *Emulation Subsystem* and itself:

- o - **SbCreateSession** - *Sm* calls this API to implement a portion of **SmCreateForeignSession**. After examining the image type, *Sm* directs this call to the appropriate *Emulation Subsystem*.

- o - **SbTerminateSession** - *Sm* calls this API to implement a portion of **SmTerminateForeignSession**. After locating the *Emulation Subsystem* responsible for the specified session ID, *Sm* makes this call to the *Emulation Subsystem*.

- o - **SbForeignSessionComplete** - *Sm* calls this API to implement a portion of **SmSessionComplete**. After locating the *Emulation Subsystem* responsible for the specified session ID, *Sm* makes this call to the *Emulation Subsystem*.

Emulation Subsystems - *Emulation Subsystems* implement the operating system service APIs for a given operating system environment. In this role, *Emulation Subsystems* act as "system service servers" exporting system service APIs between themselves and the applications that run in a particular environment. The *LPC* connections between an application and

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

its *Emulation Subsystem* are not trusted. When an *Emulation Subsystem* is called it can determine if it created the calling thread and can fail the call if appropriate.

Emulation Subsystems maintain connections to other subsystems as well. These connections are static connections created at system initialization time and are trusted. Each *Emulation Subsystem* maintains the following static connections:

o - A pair of connections is maintained between each *Emulation Subsystem* and *Sm*. One connection is used when the *Emulation Subsystem* is acting as a server to export the **Sb...** APIs to *Sm*. The other connection is used when the *Emulation Subsystem* is acting as a client calling the **Sm...** APIs.

o - A single connection is maintained between each *Emulation Subsystem* and *Piper*. This connection allows the subsystem to pass pipe stream input and output between itself and another *Emulation Subsystem*. The *Emulation Subsystem* is responsible for determining when I/O needs to be serviced using APIs available over this connection. The **Windows NT** I/O system is not involved in this decision.

o - A pair of connections is maintained between each *Emulation Subsystem* and the *Debugger Subsystem* (*Dbg*). One connection is used when the *Emulation Subsystem* is acting as a server to export the **SbDebugSupport** API to *Dbg*. This API lets *Dbg* read and write the memory and context associated with the specified thread, and to control the execution (start, stop, terminate) of the specified thread. The other connection is used by the *Emulation Subsystem* to notify *Dbg* of significant events occuring in a "debugged" thread or process (e.g., encountering an exception, process or thread creation, process or thread termination).

o - A pair of implicit connections are maintained between each *Emulation Subsystem* and the **Windows NT** executive. These connections can act as the "*DebugPort* and *ExceptionPort*" values specified in a call to **NtCreateProcess**. Upon receipt of an exception, the **Windows NT** executive examines the process of the thread in which the exception occured. If the process

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

was created with either a *DebugPort* or an *ExceptionPort*, then the *Emulation Subsystem* is notified of the exception over this connection.

Piper Subsystem - *Piper* is implemented as a server subsystem that views *Emulation Subsystems* as its clients. *Piper* only maintains trusted *LPC* connections between itself and the *Emulation Subsystems*. *Piper* is responsible for maintaining read/write data streams. *Piper* exports the following APIs:

- o - **PiperCreatePipe** - This API causes the *Piper* to create a pipe stream accessible to processes in the specified sessions. The data in the stream is only available by having the process' *Emulation Subsystem* call *Piper*.

- o - **PiperJoinPipe** - This API causes the *Piper* to bind to a pipe stream so that data can flow over the pipe.

- o - **PiperLeavePipe** - This API causes the *Piper* to close one end of a pipe stream. Once both ends of a pipe stream are closed, the pipe and any remaining data become inaccesible.

- o - **PiperReadPipe** - This API causes the *Piper* to return data stored in the pipe stream making room for new data.

- o - **PiperWritePipe** - This API causes the *Piper* to store data in the specified pipe stream.

Dbg Subsystem - The *Dbg Subsystem* implements the machine dependent facilities needed to debug an application thread. For more information on the *Dbg Subsystem*, refer to the **Windows NT Debug Architecture** document.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2. General Sm Services

The *Sm* has several classes of client, and provides services tailored to each class. The services that are used by more than one class of client are:

SmConnectToSm
SmGetLogonObjectDirectory

These services are described in the following subsections.

2.1 SmConnectToSm

NTSTATUS
SmConnectToSm(
    IN PSTRING *SbApiPortName* **OPTIONAL***,*
    IN HANDLE *SbApiPort* **OPTIONAL***,*
    OUT PHANDLE *SmApiPort*
    );

Parameters:

SbApiPortName - An optional string that if supplied specifies the name of a connection port that *Sm* will use to connect back to the *Emulation Subsystem*. This parameter is only used by *Emulation Subsystems* that are known to *Sm*.

SbApiPort - A optional handle that if supplied specifies a handle to a port named by the *SbApiPortName* parameter. This parameter is only used by *Emulation Subsystems* that are known to *Sm*.

SmApiPort - An output variable that returns a handle to a communication port connected to *Sm*, and over which the **Sm...** APIs may be made.

The **SmConnectToSm** API is provided so that *Emulation Subsystem's* and *Logon Processes* can connect to *Sm*. For *Emulation Subsystem's*, the *SbApiPortName,* and *SbApiPort* parameters must be supplied. This is because in addition to creating a connection to *Sm* (over which the **Sm...** APIs are exported), a connection is made to the *Emulation Subsystem* over which the **Sb...** APIs are exported.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.2 SmGetLogonObjectDirectory

NTSTATUS
SmGetLogonObjectDirectory(
    IN ULONG *SessionId* **OPTIONAL**,
    OUT PSTRING *LogonObjectDirectoryName*
    );

Parameters:

SessionId - An optioanl variable that supplies the session id whose associated logon object directory name is to be found. If this optional parameter is not provided, then the caller's logon object directory name is returned.

LogonObjectDirectoryName - A variable that returns the name of the session's associated logon object directory.

The name of the logon object directory associated with a session can be determined using the **SmGetLogonObjectDirectory** function.

3. Logon Process Support

Before a user can make use of the **Windows NT** system, that user must first "logon" to the system. Device-specific logon processes are responsible for collecting information about the user and authenticating the user. The authentication is performed using services of the Local Security Authority. Following authentication, a logon process may decide to activate a user interface shell program to interact with the user.(1) This is done using *Sm* services.

The *Sm* services provided to support logon processes are:

SmRegisterLogonProcess
SmExecLogonShell

These services are described in following subsections. Before these API descriptions, some background/philosophy informtion is provided on logon processes.

[1] Note that this is not always the case. The LAN Manager logon process, for instance, authenticates users as part of session setup, but no shell process is activated.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.1 Logon Process Philosophy

The general philosophy and logic of logon processes, from the perspective of the *Sm* is:

- Some set of logon process are activated by configuration control or other means. For the standard Windows NT devices (windows, terminals, LAN Manager), the logon processes will be started as part of device/network initialization. Other logon processes, such as automated teller device, or cash-register device logon processes may be started either via configuration control, or other mechanisms, such as operator actions.

  Note that there is nothing special about a logon process except that it has the **SeTcbPrivilege** privilege. Note also that a logon process does not have to be an independent process running nothing but logon process code. For example, the windows server (User32 server) could include logon processing code within it.

- Each logon process connects to the session manager using **SmConnectToSm()**. The *SbApiPortName* is left null in this call to indicate that something other than an emulation subsystem is connecting. At this time, the session manager doesn't yet know that the connected client is a logon process.

- The logon process then identifies itself as a logon process. This is done using the **SmRegisterLogonProcess()** API. This allows the session manager to authenticate the caller as having the **SeTcbPrivilege**.

  As part of **SmRegisterLogonProcess()** processing, the session manager opens the client process for **PROCESS_DUP_HANDLE** access. Note that all calls from this logon process must originate from this same process. That is, the port object handles used to communicate with the session manager can not be shared with a third process who will also act as a logon process.

- When a user attempts to log on, the logon process collects identification and authentication information and calls the Local Security Authority (LSA) directly to authenticate the user. If the authentication is successful, the logon process will be given a handle to a primary token representing the new logon session.

- Once a user has been successfully authenticated, the logon process may activate a root process for the user by calling **SmExecLogonShell()**. This call takes as parameters:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- The name of the shell (image) to activate,
- A handle to the primary token to assign to the new process,
- Memory quota information for the new process,
- A GUID representing the new logon session (which the session manager will use to create a logon object directory),
- (optional) environment variables that are to be passed to the new logon shell process.

o The session manager attempts to create a new process running the logon shell image. The session manager sets the process's primary token to be that supplied by the logon process. The initial thread of this process is created, but left in a suspended state. It is the logon process's responsibility to resume the thread when desired.

If the process creation is successful, then handles to the newly created shell process and thread are returned to the logon process. The process handle will be open for **SYNCHRONIZE** access. The thread handle will be open for **THREAD_SUSPEND_RESUME** access. Logon processes are expected to close these handles when no longer needed.

This allows logon processes to:

1) Specify UI shell initialization parameters (via environment variables). For example, the User32 logon process will specify the name of the window station the user has logged on from using environment variables.

2) Wait on the newly logged on process to exit unexpectedly. For example, a windows32 logon shell is expected to open a desktop object in the window station the user logged on from. If the shell process exits before openning a desktop, then the User32 logon process assumes something has gone wrong and treats the condition as a logoff, making the window station available for another logon.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.2 SmRegisterLogonProcess

```
NTSTATUS
SmRegisterLogonProcess(
        IN HANDLE SmApiPort,
        IN PSTRING LogonProcessName
        );
```

Parameters:

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

LogonProcessName - A name string that identifies the logon process. This should be a printable name suitable for display to administrators. For example, "User32LogonProcess" might be used for the windows logon process name. No check is made to determine whether the name is already in use.

Return Value:

STATUS_SUCCESS - The call completed successfully.

STATUS_PRIVILEGE_NOT_HELD - Indicates the caller does not have the privilege necessary to act as a logon process. **SeTcbPrivilege** is needed.

Before being able to use the **SmExecLogonShell()** service, a logon process must identify itself as a logon process. This is done using the **SmRegisterLogonProcess()** service.

This service verifies that the caller is a legitimate logon process. This is done by ensuring the caller has **SeTcbPrivilege**. It also opens the caller's process for **PROCESS_DUP_HANDLE**. This information is cached for future use.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3.3 SmExecLogonShell

NTSTATUS
SmExecLogonShell(
    IN HANDLE *SmApiPort,*
    IN GUID *LogonGuid*,
    IN PSTRING *ShellImageName*,
    IN HANDLE *PrimaryToken*,
    IN QUOTA_LIMITS *Quotas*,
    IN RTL_USER_PROCESS_PARAMETERS *ProcessParameters*,
    OUT PHANDLE *Process*,
    OUT PHANDLE *Thread*
    );

Parameters:[2]

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

LogonGuid - A GUID uniquelly assigned to represent this logon session.

ShellImageName - Provides the path name of the shell program to execute.

PrimaryToken - Provides a handle to the primary token to assign to the new process. This handle must be open for **TOKEN_ASSIGN_PRIMARY** access.

Quotas - Provides quota values to be assigned to the new process.

ProcessParameters - Provides parameters to be passed to the new process.

Process - Receives a handle to the new process. The handle will be open for **SYNCHRONIZE** access.

Thread - Receives a handle to the initial thread of the process. The handle will be open for **THREAD_SUSPEND_RESUME** access. The thread will not yet have been activated.

---

[2] Loup, DaveC, DarrylH: Do we need a *CaptiveAccount* parameter too?

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>Return Value:</u>

STATUS_SUCCESS - The call completed successfully.

STATUS_NOT_LOGON_PROCESS - The caller has not registered as a logon process.

STATUS_LOGON_SESSION_EXISTS - Indicates the GUID assigned to this logon session is already in use.

In addition to these, the following general classes of errors may be returned:

- Errors related to creation of a process or thread, including attempts to access the image file.
- Attempts to duplicate and assign the primary token.

This service is used by logon processes to activate a user interface shell program for a newly logged on interactive user. The logon process may pass information to the new shell program via environment variables.

The session manager:

1) Creates a new logon session to run the logon shell program in,
2) Creates a logon object directory for the new logon session,
3) creates the logon program and the initial thread in that program (but leaves the thread in a suspended state).

Handles to the new process and its initial thread are passed back to the requesting logon process. The process handle will be open for **SYNCHRONIZE** access. The thread handle will be open for **THREAD_SUSPEND_RESUME** access. The logon process is expected to close these handles when no longer needed.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. System Subsystems Support

System subsystems are logical extensions of the operating system. They provide privileged and protected operating system support, but are implemented as separated processes that execute in user mode.

4.1 Session Control Services

The *Sm* subsystem is responsible for coordinating the creation and management of sessions. It is responsible for coordinating the creation of sessions when *Emulation Subsystems* encounter an image file designed to operate in a different API environment.

Sm tends to act as an intermediary between *Emulation Subsystems*. It is responsible for allocating session ID's, and for associating a session ID with its controlling *Emulation Subsystem*.

Sm is also responsible for associating an image file with the *Emulation Subsystem* it is designed to run with.

Sm exports the following APIs to support *Emulation subsystem* operations:

SmCreateForeignSession
SmSessionComplete
SmTerminateForeignSession

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.1.1 SmCreateForeignSession

A request to create a foreign session can be made using the **SmCreateForeignSession** function.

NTSTATUS
SmCreateForeignSession(
    IN HANDLE *SmApiPort,*
    OUT PULONG *ForeignSessionId,*
    IN ULONG *SourceSessionId,*
    IN PRTL_USER_PROCESS_INFORMATION *ProcessInformation,*
    IN PCID *DebugUiClientId* **OPTIONAL**,
    IN HANDLE *StandardInput* **OPTIONAL**,
    IN HANDLE *StandardOutput* **OPTIONAL**,
    IN HANDLE *StandardError* **OPTIONAL**
    );

Parameters:

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm.*

ForeignSessionId - A variable whose return value specifies the session ID of the created session. The session ID is assigned by the session manager. The session ID is used in the session control APIs to identify the target foreign session.

SourceSessionId - A variable that specifies the session ID of the application that is creating (through its *Emulation Subsystem*) the foreign session. This session ID is used by *Sm* to determine a user profile for the new session.

ProcessInformation - A structure that describes the process to be run as a foreign session. This data structure contains a complete description of the process including handles to the process and its initial thread. Using **NtDupObject**, *Sm* makes these handles available to the *Emulation Subsystem* responsible for the process. Regardless of the outcome of this call, the calling process looses its handles to the process and thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

DebugUiClientId - An optional parameter that specifies the client ID of the debugger user interface that is debugging the session. If this parameter is specified, then the session is a "debug session".

StandardInput - An optional handle that specifies the standard input stream associated with the session. Using **NtDupObject**, *Sm* makes this handle available to the *Emulation Subsystem* responsible for the process. Regardless of the outcome of this call, the calling process' version of this handle is closed.

StandardOutput - An optional handle that specifies the standard output stream associated with the session. Using **NtDupObject**, *Sm* makes this handle available to the *Emulation Subsystem* responsible for the process. Regardless of the outcome of this call, the calling process' version of this handle is closed.

StandardError - An optional handle that specifies the standard error output stream associated with the session. Using **NtDupObject**, *Sm* makes this handle available to the *Emulation Subsystem* responsible for the process. Regardless of the outcome of this call, the calling process' version of this handle is closed.

Emulation Subsystems use this service whenever they are instructed to execute an image whose type is not supported by the subsystem (e.g. an **OS/2** application executes a **DosExecPgm** specifying an image file that is a **POSIX** application).

Sm implements this API by associating the image file type with an appropriate *Emulation Subsystem*, allocating a new session ID, transfering the handles (*Thread, Process, StandardInput, StandardOutput,* and *StandardError*) into the appropriate *Emulation Subsystem's* handle table, and calling the *Emulation Subsystem* at its **SbCreateSession** entry point. Assuming that the call to **SbCreateSession** succeeds, the session ID of the new session is returned to the caller.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.1.2 SmSessionComplete

Sm is notified that a session has completed through the **SmSessionComplete** function.

NTSTATUS
SmSessionComplete(
    IN HANDLE *SmApiPort,*
    IN ULONG *SessionId,*
    IN NTSTATUS *CompletionStatus*
    );

Parameters:

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

SessionId - A parameter that specifies the session ID of the foreign session that has completed.

CompletionStatus - A parameter that specifies the completion status of the session.

The **SmSessionComplete** API is provided so that an *Emulation Subsystem* can notify *Sm* that one of its sessions has completed.

Once *Sm* receives this call, it locates the *Emulation Subsystem* that created the foreign session using the specified session ID, and calls the subsystem at its **SbForeignSessionComplete** entry point.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.1.3 SmTerminateForeignSession

A request that a foreign session be terminated can be made through the **SmTerminateForeignSession** function.

NTSTATUS
SmTerminateForeignSession(
    IN HANDLE *SmApiPort,*
    IN ULONG *ForeignSessionId,*
    IN NTSTATUS *TerminationStatus*
    );

Parameters:

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

ForeignSessionId - A parameter that specifies the session ID of the foreign session being terminated.

TerminationStatus - A parameter that specifies the reason that the foreign session should be terminated.

The **SmTerminateForeignSession** API is provided so that an *Emulation Subsystem* can request the termination of a foreign session that it created.

Sm implements this call by locating the appropriate *Emulation Subsystem* using the specified foreign session ID, and then calling the subsystem at its **SbTerminateSession** entry point.

The **SmTerminateForeignSession** call returns before the session is actually terminated. When the session terminates *Sm* will be notified.

4.2 Piper

The *Piper* subsystem is responsible for providing pipe stream input and output between threads in different sessions (under the supervision of *Emulation Subsystems*).

This capability is provided to support transfering information between applications that are of a different class (e.g *foo* | *bar* where *foo* is a **POSIX** application and *bar* is and **OS/2** application).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Piper requires coordination between the *Emulation Subsystem* involved in the data piping, and the application runtime libraries that provide stream input and output through the *STDIN, STDOUT,* and *STDERR* I/O streams. All application input and output through these streams must be handled by the application's *Emulation Subsystem.* Only the subsystem knows the session that the application is part of, and the "file names" of its input, output, and error streams.

Piper exports the following APIs:

PiperCreatePipe
PiperJoinPipe
PiperLeavePipe
PiperReadPipe
PiperWritePipe

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2.1 PiperCreatePipe

An *Emulation Subsystem* creates the potential for pipe stream communication between the application threads in one of its sessions, and application threads in a "foreign" session that it asked *Sm* to create using the **PiperCreatePipe** function.

NTSTATUS

PiperCreatePipe(

> **IN ULONG** *ForeignSessionId*,
> **IN ULONG** *SourceSessionId*
>);

Parameters:

> *ForeignSessionId* - Specifies the session ID of the foreign session that makes up the other end of the pipe.
>
> *SourceSessionId* - Specifies the session ID of the local session that is creating the pipe.

Creating a pipe causes the potential for pipe stream communication to occur between the two specified sessions. Pipes provide a full duplex byte stream communication path between application threads in the specified sessions.

Data written by application threads within the local session is made available (to satisfy pipe reads) to threads within the foreign session. Reads to the pipe by application threads within the local session are satisfied by corresponding pipe writes made by threads within the foreign session.

After this call completes, application threads within the local session may attempt to read data from the pipe, and write data to the pipe. Until the foreign session joins the pipe using the **PiperJoinPipe** API, data that they write will remain in the pipe, and their pipe reads will block.

There is no need to synchronize this call with a corresponding **PiperJoinPipe** call specifying the foreign session. These calls may be issued in either order.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2.2 PiperJoinPipe

An *Emulation Subsystem* joins a pipe so that it can participate in pipe stream communication between the application threads in one of its sessions, and application threads in the session that created it using the **PiperJoinPipe** function.

NTSTATUS
PiperJoinPipe(

> **IN ULONG** *SessionId*
>);

Parameters:

> *SessionId* - Specifies the session ID of the local session that is joining a pipe.

Joining a pipe allows the threads within the specified local session to begin pipe stream communication over a pipe created in a corresponding call to **PiperCreatePipe**.

After this call completes, application threads within the local session may read data from the pipe, and write data to the pipe.

This call completes when a corresponding call to **PiperCreatePipe** is issued specifying the local session in its *ForeignSessionId* parameter.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2.3 PiperLeavePipe

An *Emulation Subsystem* leaves a pipe which informs *Piper* that it no longer wants to participate in pipe stream communication using the **PiperLeavePipe** function.

NTSTATUS

PiperLeavePipe(

IN ULONG *SessionId*
);

Parameters:

SessionId - Specifies the session ID of the local session that is leaving the pipe.

Leaving a pipe causes application threads within the local session to disassociate themselves with the pipe. All data destined for the local session is flushed, and further pipe writes to the local session fail.

When both sessions that make up a pipe leave the pipe, the pipe is deleted. All data within or destined for the pipe is deleted.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2.4 PiperReadPipe

An *Emulation Subsystem* can read data from a pipe stream that it has either joined or created using the **PiperReadPipe** function.

NTSTATUS

PiperReadPipe(

> **IN ULONG** *SessionId*,
> **OUT PUCHAR** *DataReadBuffer*,
> **IN ULONG** *DataReadLength*,
> **OUT PULONG** *DataActuallyReadLength*
>);

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2.5 PiperWritePipe

An *Emulation Subsystem* can write data to a pipe stream that it has either joined or created using the **PiperWritePipe** function.

NTSTATUS

PiperWritePipe(

> **IN ULONG** *SessionId*,
> **IN PUCHAR** *DataWriteBuffer*,
> **IN ULONG** *DataWriteLength*,
> **OUT PULONG** *DataActuallyWrittenLength*
>);

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Emulation Subsystems

The primary role of an *Emulation Subsystem* is to emulate a set of system services using native **Windows NT** system services. Applications written to a particular API use the appropriate *Emulation Subsystem* to implement that particular system API.

Each application contains in its image file header, a description of the *Emulation Subsystem* that the application requires (e.g. OS/2 applications like cmd.exe describe the **OS/2++** subsystem). In addition to providing operating system API emulation, the subsystem is responsible for managing the session to which the application belongs. The subsystem also acts as an intermediary between the **Dbg** protected subsystem and the application when the application is being "debugged".

Each *Emulation Subsystem* exports three **Windows NT** connection ports. An *LPC* connection to an *Emulation Subsystem* is established by specifying one of these ports in a call to **NtConnectPort**. Each connection port is associated with a class of services implemented by the *Emulation Subsystem*. The three classes of services are:

- *Sm* to *Emulation Subsystem* APIs. The connection port associated with this class of services is protected such that only the *Sm* subsystem can access the port. Once a connection has been established, the *Emulation Subsystem* does not respond to connection requests arriving on this port. The connection between *Sm* and each *Emulation Subsystem* is a trusted connection.

- *Dbg* to *Emulation Subsystem* APIs. The connection port associated with this class of services is protected such that only the *Dbg* subsystem can access the port. Once a connection has been established, the *Emulation Subsystem* does not respond to connection requests arriving on this port. The connection between *Dbg* and each *Emulation Subsystem* is a trusted connection.

- Operating System APIs emulated by the subsystem. The connection port associated with this class of services does not have to be protected. Each application that is using the APIs exported over this connection establishes a connection during its process initialization sequence (*crt0* equivalent). *Emulation Subsystem* must authenticate each call (associate the caller's CID with a CID created by the subsystem) to ensure that the thread making the call is one of its threads. The connection

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

between and application and its *Emulation Subsystem* is not a trusted connection.

5.1 PSX++

The **PSX++** protected subsystem implements the APIs described in the *IEEE P1003.1/Draft 13 August 22, 1988* specification. It is responsible for managing all applications written to this API.

5.2 OS/2++

The **OS/2++** protected subsystem implements the *Cruiser OS/2 V2.0* APIs. It is responsible for managing all applications written to this API.

5.3 NT++

The **NT++** protected subsystem implements a very small set of APIs. Its primary purpose is to implement the set of APIs needed to manage and control sessions, and to provide a *DosExecPgm* like API that a native debugger user interface or application can use to create and manage a session or to execute an image designed to run with one of the other *Emulation Subsystems*.

5.4 Emulation Subsystem APIs

Each *Emulation Subsystem* exports a set of APIs designed to manage and control sessions. These APIs are called by the *Sm*, *Dbg*, or by the **Windows NT** executive.

Emulation Subsystems export the following APIs:

SbCreateSession
SbTerminateSession
SbForeignSessionComplete

Emulation Subsystems see the APIs in their raw form. The subsystems must provide their own "API Loops" that receive and reply using *LPC* messages. The subsystem APIs are all called (by their own API loops) with a pointer to a subsystem API message (*SBAPIMSG*) the format of the message is given below:

SbApiMsg Structure

PORTMSG *h* - This field contains a standard *LPC* port message. The *ClientId* of the sender, message type, and length information are all placed in this area by the system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SBAPINUMBER *ApiNumber* - This field specifies the API number of the call. Values are:

ApiNumber Enumeration

SbCreateSessionApi - The message specifies the **SbCreateSession** API.

SbTerminateSessionApi - The message specifies the **SbTerminateSession** API.

SbForeignSessionCompleteApi - The message specifies the **SbForeignSessionComplete** API.

NTSTATUS *ReturnedStatus* - This field is used to pass the return status of the **Sb...** API back to the caller of the API. This field is designed to be modified by the "API loop".

union *u* - This union contains one field for each of the API types.

u Union

SBCREATESESSION *CreateSession* - This field contains information specific to the **SbCreateSession** API.

SBTERMINATESESSION *TerminateSession* - This field contains information specific to the **SbTerminateSession** API.

SBFOREIGNSESSIONCOMPLETE *ForeignSessionComplete* - This field contains information specific to the **SbForeignSessionComplete** API.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.4.1 SbCreateSession

A session is created and placed under the control of an *Emulation Subsystem* through the **SbCreateSession** function.

NTSTATUS
SbCreateSession(
    IN OUT PSBAPIMSG *SbApiMsg*
    );

Parameters:

SbApiMsg - A variable that supplies an *LPC* message that contains information necessary to allow the subsystem to create a session capable of running the process described in the message.

The *ApiNumber* associated with this call is *SbCreateSessionApi*. The *CreateSession* field of the API message contains the following:

CreateSession Structure

ULONG *SessionId* - A variable that specifies the session ID to be associated with the session being created. The session ID is assigned by the session manager. The session ID is used in the session control APIs to identify the target session.

RTL_USER_PROCESS_INFORMATION *ProcessInformation* - A structure that describes the process to be run as a new session. This data structure contains a complete description of the process including handles to the process and its initial thread. The subsystem is responsible for the process and thread even if it fails the create session request. It must terminate and close the process and thread at the appropriate time (even if it fails the session creation).

CID *DebugUiClientId* - An optional parameter that specifies the client ID of the debugger user interface that is debugging the session. If this parameter is specified, then the session is created as a "debug session". Debug sessions are created in a suspended state (i.e., the initial thread of the process is left suspended). In addition, the subsystem servicing this call must call into the *Dbg* subsystem to report the new debug session

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

and the CID of the debugger user interface that is debugging the session.

// All Windows NT threads are created in a suspended state. Most Emulation Subsystems create an application thread by creating a Windows NT thread and then resuming the thread. This parameter instructs the Emulation Subsystem to not resume the application thread. The Emulation Subsystem will be instructed to resume the thread through a DebugUi -> Dbg -> Emulation Subsystem transaction. //

The value of this parameter originates in the system. When a *DebugUi* issues a call to an API that creates a "debug process" the CID of the *DebugUi* is captured by the *DebugUi's Emulation Subsystem* from the message header of the message associated with the process creation call. If the process is foreign to the *DebugUi's* subsystem, the CID passes from the *DebugUi's Emulation Subsystem* to *Sm*, and then from *Sm* to the *Emulation Subsystem* that should run the process.

HANDLE *StandardInput* - An optional handle that specifies the standard input stream associated with the session. Regardless of the outcome of this call, the subsystem is responsible for closing the handle at the appropriate time (even if it fails the session creation).

HANDLE *StandardOutput* - An optional handle that specifies the standard output stream associated with the session. Regardless of the outcome of this call, the subsystem is responsible for closing the handle at the appropriate time (even if it fails the session creation).

HANDLE *StandardError* - An optional handle that specifies the standard error output stream associated with the session. Regardless of the outcome of this call, the subsystem is responsible for closing the handle at the appropriate time (even if it fails the session creation).

The *Sm* subsystem uses the **SbCreateSession** API to create a session to run the specified process. This call is made as part of the logon sequence (part of **SmLogonUser**), or when *Sm* is asked (by another subsystem, termed the "source subsystem") to create a session to run an image whose format is not understood by the source subsystem.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.4.2 SbTerminateSession

A session can be terminated through the **SbTerminateSession** function.

NTSTATUS
SbTerminateSession(
    IN OUT PSBAPIMSG *SbApiMsg*
    );

Parameters:

SbApiMsg - A variable that supplies an *LPC* message that contains information necessary to allow the subsystem to terminate the specified session.

The *ApiNumber* associated with this call is *SbTerminateSessionApi*. The *TerminateSession* field of the API message contains the following:

TerminateSession Structure

ULONG *SessionId* - A value that specifies the session ID of the session being terminated.

NTSTATUS *TerminationStatus* - A that specifies the reason that the session should be terminated.

The **SbTerminateSession** API is provided so that a session can be terminated. This call is made by the *Sm* subsystem in response to a request by the *Emulation Subsystem* that indirectly created the session.

The **SbTerminateSession** call returns before the session is actually terminated. When the session terminates, the *Sm* subsystem will be notified through an RPC from the session's controlling *Emulation Subsystem* to *Sm* at its **SmSessionComplete** entry point.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5.4.3 SbForeignSessionComplete

An *Emulation Subsystem* is notified that a foreign session has completed through the **SbForeignSessionComplete** function.

NTSTATUS
SbForeignSessionComplete(
    IN OUT PSBAPIMSG *SbApiMsg*
    );

Parameters:

SbApiMsg - A variable that supplies an *LPC* message that contains information which notifies the subsystem that a foreign session that it started has completed.

The *ApiNumber* associated with this call is *SbForeignSessionCompleteApi*. The *ForeignSessionComplete* field of the API message contains the following:

ForeignSessionComplete Structure

ULONG *SessionId* - A value that specifies the session ID of the session that has completed.

NTSTATUS *CompletionStatus* - A value that specifies the completion status of the session.

The **SbForeignSessionComplete** API is provided so that a subsystem can be notified that a foreign session that it created has completed. The subsystem that services this call is the subsystem that originally requested that the foreign session be created.

Once this call returns, the session ID is available for re-use.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Revision 1.9, January 7, 1990. Jim Kelly (JimK)

1) Eliminated all references to Presentation Manager.

2) Changed logon so that logon processes authenticate directly with the Local Security authority (LSA) and then interact with the NT Session Manager to activate the logon shell process. This obsoleted the SmLogonUser() API and caused the introduction of the SmRegisterLogonProcess() and SmExecLogonShell() APIs.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Event - Semaphore Specification

Author: *Lou Perazzoli*

Original Draft 1.0, January 5, 1989
Revision 1.3, May 11, 1989
Revision 1.4, August 8, 1989
Revision 1.5, October 23, 1989
Revision 1.6, December 1, 1989
Revision 1.7, January 3, 1990
Revision 1.8, January 23, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction.. 1

2. Event Objects 1
  2.1 Create Event Object 1
  2.2 Open Event Object 2
  2.3 Set Event 3
  2.4 Reset Event... 4
  2.5 Pulse Event.. 4
  2.6 Query Event... 4

3. Semaphore Objects.. 5
  3.1 Create Semaphore Object.. 5
  3.2 Open Semaphore Object.. 6
  3.3 Release Semaphore Object 7
  3.4 Query Semaphore 8

4.0 Delay Execution.. 9

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the **Windows NT** event and semaphore objects and the wait services. A definition and an explanation of operation is given for each API. No attempt has been made, however, to fully explain all error conditions and their consequences.

The APIs described include:

NtCreateEvent - create event and open handle
NtOpenEvent - open handle to existing event
NtSetEvent - set event to Signal state
NtResetEvent - set event to Not-Signaled state
NtPulseEvent - set / reset event state atomically
NtQueryEvent - get information about event
NtCreateSemaphore - create semaphore and open handle
NtOpenSemaphore - open handle to existing semaphore
NtReleaseSemaphore - release semaphore
NtQuerySemaphore - get information about semaphore
tDelayExecution - delay execution for the specified time
NtClose - close an object handle

2. Event Objects

There are two types of event objects, *notification events* and *synchronization events*. Notification event objects provide a mechanism for notification. Notification events are either *Signaled* (TRUE) or *Not-Signaled* (FALSE). An event may be set multiple times, yet the state remains Signaled. Notification events provides no ownership capability. If multiple threads are waiting on a notification event, then when the event becomes Signaled, **all** threads waiting for the event are made "runnable". A notification event becomes Not-Signaled only when explicitly reset.

Synchronization event objects have the property that when the event is set, the event attains a state of Signaled, which releases a single thread currently waiting on the event, and then the event immediately attains a state of Not-Signaled. If there are no threads waiting on the event, the state of the event remains Signaled. This allows threads to "synchronize" on the signaling of the event. Like notification events, synchronization events provide no ownership capability.

A synchronization event attains a state of Not-Signaled when explicitly reset or when the first wait operation is satisfied on the event. Note that any time an event attains a state of Not-Signaled, the event count for the state of the event is set to zero.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1 Create Event Object

An event object is created and a handle opened for access to the object with the **NtCreateEvent** function:

NTSTATUS
NtCreateEvent (
    OUT PHANDLE *EventHandle,*
    IN ULONG *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
    IN EVENT_TYPE *EventType,*
    IN BOOLEAN *InitialState*
    );

Parameters:

EventHandle - A pointer to a variable that receives the event object handle value.

DesiredAccess - The desired types of access for the event. The following object type specific access flags can be specified in addition to the *STANDARD_RIGHTS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

EVENT_QUERY_STATE - Query access to the event is desired.

EVENT_MODIFY_STATE - Modify state access (set and reset) to the event is desired.

SYNCHRONIZE - Synchronization access (wait) to the event is desired.

ObjectAttributes - An optional pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details.

EventType - The type of event object to be created. One of *NotificationEvent* or *SynchronizationEvent.*

InitialState - The initial state of the event object, one of *TRUE* or *FALSE.* If the *InitialState* is specified as TRUE, the event's current state value is set to one, otherwise it is set to zero.

The **NtCreateEvent** function creates an event object with the specified initial state. If an event is in the Signaled state (TRUE), a wait operation on the event does not

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

block. If the event is in the Not-Signaled state (FALSE), a wait operation on the event blocks until the specified event attains a state of Signaled, the timeout value is exceeded, or an alert is delivered.

2.2 Open Event Object

A handle can be opened to an existing event object with the **NtOpenEvent** function:

NTSTATUS
NtOpenEvent *(*
    OUT PHANDLE *EventHandle,*
    IN ULONG *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes,*
    );

Parameters:

EventHandle - A pointer to a variable that receives the value of the event object handle value.

DesiredAccess - The desired types of access to the event. The following object type specific access flags can be specified in addition to the *STANDARD_RIGHTS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

EVENT_QUERY_STATE - Query access to the event is desired.

EVENT_MODIFY_STATE - Modify state access (set and reset) to the event is desired.

SYNCHRONIZE - Synchronization access (wait) to the event is desired.

ObjectAttributes - A pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details.

2.3 Set Event

An event can be set to the signaled state (*TRUE*) with the **NtSetEvent** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtSetEvent *(*
    IN HANDLE *EventHandle,*
    OUT PLONG *PreviousState* **OPTIONAL**
    );

Parameters:

EventHandle - An open handle to an event object.

PreviousState - An optional pointer to a variable that receives the previous state of the event. Zero is Not-Signaled, non-zero is Signaled. The value indicates the number of times the event has been set since the last reset.

Setting the event causes the event to attain a state of Signaled, which releases all threads currently waiting on the event. Any threads which issue a wait operation on the event do not block and continue to execute. It also increments the event count for the state of the event.

2.4 Reset Event

The state of an event is set to the Not-Signaled state (*FALSE*) using the **NtResetEvent** function:

NTSTATUS
NtResetEvent *(*
    IN HANDLE *EventHandle,*
    OUT PLONG *PreviousState* **OPTIONAL**
    );

Parameters:

EventHandle - An open handle to an event object.

PreviousState - An optional pointer to a variable that receives the previous state of the event. Zero is Not-Signaled, non-zero is Signaled. The value indicates the number of times the event has been set since the last reset.

Once the event attains a state of Not-Signaled, any threads which wait on the event block, awaiting the event to become Signaled. The reset event service sets the event count to zero for the state of the event.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.5 Pulse Event

An event can be set to the Signaled state and reset to the Not-Signaled state atomically with the **NtPulseEvent** function:

NTSTATUS
NtPulseEvent *(*
    IN HANDLE *EventHandle,*
    OUT PLONG *PreviousState* **OPTIONAL**
    );

Parameters:

EventHandle - An open handle to an event object.

PreviousState - An optional pointer to a variable that receives the previous state of the event. Zero is Not-Signaled, non-zero is Signaled. The value indicates the number of times the event has been set since the last reset.

Pulsing the event causes the event to attain a state of Signaled, which releases all threads currently waiting on the event, and then attain a state of Not-Signaled. The pulse event service sets the event count to zero for the state of the event.

2.6 Query Event

The state of an event can be queried with the **NtQueryEvent** function:

NTSTATUS
NtQueryEvent *(*
    IN HANDLE *EventHandle,*
    IN EVENT_INFORMATION_CLASS *EventInformationClass,*
    OUT PVOID *EventInformation,*
    IN ULONG *EventInformationLength,*
    OUT PULONG *ReturnLength* **OPTIONAL**
    );

Parameters:

EventHandle - An open handle to an event object.

EventInformationClass - The event information class about which to retrieve information.

EventInformation - A pointer to a buffer that receives the specified information. The format and content of the buffer depend on the specified event class.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>EventInformation Format by Information Class:</u>

EventBasicInformation - Data type is *EVENT_BASIC_INFORMATION*.

<u>EVENT_BASIC_INFORMATION Structure</u>

EVENT_TYPE *EventType* - The type of the event.

LONG *EventState* - The current state of the event.

EventInformationLength - Specifies the length in bytes of the event information buffer.

ReturnLength - An optional pointer which, if specified, receives the number of bytes placed in the event information buffer.

This function provides the capability to determine the state and granted access of an event object.

3. Semaphore Objects

Semaphore objects provide a mechanism for resource gates. When a semaphore is created, it is provided an initial count and maximum count. When a thread waits on a semaphore, if the current count is greater than zero, then the current count is decremented and the thread continues to execute. If the current count is zero, the thread blocks until the count becomes greater than zero. When a thread releases a semaphore, the current count is augmented. Semaphores do not provide ownership; multiple threads can be waiting and releasing the same semaphore.

3.1 Create Semaphore Object

A semaphore object is created and a handle opened for access to the object with the **NtCreateSemaphore** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtCreateSemaphore (
    OUT PHANDLE *SemaphoreHandle,*
    IN ULONG *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL***,*
    IN LONG *InitialCount,*
    IN LONG *MaximumCount*
    );

Parameters:

SemaphoreHandle - A pointer to a variable that receives the value of the semaphore object handle.

DesiredAccess - The desired types of access for the semaphore. The following object type specific access flags can be specified in addition to the *STANDARD_RIGHTS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

SEMAPHORE_QUERY_STATE - Query access to the semaphore is desired.

SEMAPHORE_MODIFY_STATE - Modify state access (release) to the semaphore is desired.

SYNCHRONIZE - Synchronization access (wait) to the semaphore is desired.

ObjectAttributes - An optional pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details.

InitialCount - The initial count for the semaphore, this value must be positive and less than or equal to the maximum count.

MaximumCount - The maximum count for the semaphore, this value must be greater than zero..

The **NtCreateSemaphore** function causes a semaphore object to be created which contains the specified initial and maximum counts.

3.2 Open Semaphore Object

A handle can be opened to an existing semaphore object with the **NtOpenSemaphore** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtOpenSemaphore *(*
    OUT PHANDLE *SemaphoreHandle,*
    IN ULONG *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
    );

Parameters:

SemaphoreHandle - A pointer to a variable that receives the semaphore object handle value.

DesiredAccess - The desired types of access to the semaphore. The following object type specific access flags can be specified in addition to the *STANDARD_RIGHTS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

SEMAPHORE_QUERY_STATE - Query access to the semaphore is desired.

SEMAPHORE_MODIFY_STATE - Modify state access (release) to the semaphore is desired.

SYNCHRONIZE - Synchronization access (wait) to the semaphore is desired.

ObjectAttributes - A pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details.

3.3 Release Semaphore Object

A semaphore object can be released with the **NtReleaseSemaphore** function:

NTSTATUS
NtReleaseSemaphore *(*
    IN HANDLE *SemaphoreHandle,*
    IN LONG *ReleaseCount,*
    OUT PLONG *PreviousCount* **OPTIONAL**
    );

Parameters:

SemaphoreHandle - An open handle to a semaphore object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

ReleaseCount - The release count for the semaphore. The count must be greater than zero and less than the maximum value specified for the semaphore.

PreviousCount - An optional pointer to a variable that receives the previous count for the semaphore.

When the semaphore is released, the current count of the semaphore is incremented by the *ReleaseCount*. Any threads that are waiting for the semaphore are examined to see if the current semaphore value is sufficient to satisfy their wait.

If the value specified by *ReleaseCount* would cause the maximum count for the semaphore to be exceeded, then the count for the semaphore is not affected and an error status is returned.

3.4 Query Semaphore

The state of a semaphore can be queried with the **NtQuerySemaphore** function:

NTSTATUS
NtQuerySemaphore *(*
IN HANDLE *SemaphoreHandle,*
IN SEMAPHORE_INFORMATION_CLASS *SemaphoreInformationClass,*
OUT PVOID *SemaphoreInformation,*
IN ULONG *SemaphoreInformationLength,*
OUT PULONG *ReturnLength* **OPTIONAL**
);

Parameters:

SemaphoreHandle - An open handle to a semaphore object.

SemaphoreInformationClass - The semaphore information class about which to retrieve information.

SemaphoreInformation - A pointer to a buffer which receives the specified information. The format and content of the buffer depend on the specified semaphore class.

SemaphoreInformation Format by Information Class:

SemaphoreBasicInformation - Data type is *SEMAPHORE_BASIC_INFORMATION.*

SEMAPHORE_BASIC_INFORMATION Structure

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

LONG *CurrentCount* - The current count of the semaphore.

LONG *MaximumCount* - The maximum count that may be obtained by the semaphore.

SemaphoreInformationLength - Specifies the length in bytes of the semaphore information buffer.

ReturnLength - An optional pointer which, if specified, receives the number of bytes placed in the semaphore information buffer.

This function provides the capability to determine the state and granted access of a semaphore object

4.0 Delay Execution

The execution of the current thread can be delayed for a specified interval of time with the **NtDelayExecution** function:

NTSTATUS
NtDelayExecution (
    IN BOOLEAN *Alertable,*
    IN PTIME *DelayInterval*
    );

Parameters:

Alertable - A boolean value that specifies whether the wait is alertable.

DelayInterval - The absolute or relative time over which the wait is to occur.

The **NtDelayExecution** function causes the current thread to enter a waiting state until the specified interval of time has passed. If *Alertable* is specified as *TRUE*, the wait service completes and a condition of *STATUS_ALERTED* is raised. If an **APC** is delivered while the thread is waiting alertable, the **APC** is invoked and the wait operation re-executed.

Revision History:

Original Draft 1.0, January 5, 1989

Revision 1.2, March 12, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Removed Muxwait object and Mutex object.

Revision 1.3, May 11, 1989

1. Added wait for multiple objects.

2. Added NtDelayExecution

Revision 1.4, August 8, 1989

1. Make return parameters for PreviousState and CurrentState optional.

Revision 1.5, October 23, 1989

1. Changed EventName/SemaphoreName in OBJA structure to ObjectName.

2. Added description of notification and synchronization events.

3. Changed *PreviousState* to return a count that indicates the number of times the event was set since the last reset.

4. Added the *EventType* to the query event call.

5. Changed wait services to describe the abandoned state.

Revision 1.6, December 1, 1989

1. Changed desciption of NtCreateSemaphore, NtCreateEvent, NtOpenSemaphore and NtOpenEvent to use OBJECT_ATTRIBUTES and reference Object Management Specification for detials.

2. Changed PULONG to PLONG for PreviousState argument in NtSetEvent, NtResetEvent, and NtPulseEvent.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision 1.7, January 3, 1990

1. Clarified the behavior of sychronzation events and the state of the event count.

2. Changed desired access flags for NtCreateEvent, NtOpenEvent, NtCreateSemaphore, and NtOpenSemaphore.

3. Removed NtWait description. This is now in the Object Management Specification.

Revision 1.8, January 23, 1990

1. Changed NtReleaseSemaphore to return a failure if the ReleaseCount is greater than the maximum count.

2. Changed NtReleaseSemaphore to require the ReleaseCount to be greater than 0.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 System Startup Design Note

Author: *Mark Lucovsky*

Revision 1.2, July 26, 1990
Original Draft May 31, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview 1
1.1 System Initialization 1
1.2 System Startup 1
1.2.1 Executive Level System System Startup 1
1.2.2 Session Manager System Startup 1
1.2.2.1 First Phase SM system startup 2
1.2.2.1.1 Link Keyword 2
1.2.2.1.2 PagingFile Keyword 3
1.2.2.2 Second Phase SM system startup 3
1.2.2.2.1 Subsystem Keyword 3
1.2.2.2.2 Start Keyword 4
1.2.2.2.3 Run Keyword 5
1.2.2.2.4 Debug Keyword 5
1.2.2.2.5 LibPath Keyword 5
1.2.2.2.6 Short Term Keywords 6

2. Configuration File APIs 7
2.1 Configuration File Data Structures 9
2.1.1 CONFIG_FILE 9
2.1.2 CONFIG_SECTION 9
2.1.3 CONFIG_KEYWORD 9
2.1.4 Configuration File APIs 10
2.1.4.1 RtlOpenConfigFile 10
2.1.4.2 RtlCloseConfigFile 10
2.1.4.3 RtlLocateSectionConfigFile 11
2.1.4.4 RtlLocateKeywordConfigFile 11
2.1.4.5 RtlEnumerateSectionConfigFile 12
2.1.4.6 RtlEnumerateKeywordConfigFile 12

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This design note describes system startup for NT OS/2. For the purposes of this paper, system startup begins after phase 1 system initialization completes.

This paper does not describe an agreed to, long term strategy. It describes an interim startup sequence that will be used until the system installation and system configuration plans solidify.

1.1 System Initialization

This paper does not describe system initialization in any great level of detail. System initialization occurs in three phases. After boot loading the system image, the kernel is called at **KiSystemStartup**. After kernel initialization, the executive is called to perform phase 0 initialization. This causes memory management, the object manager, and the process architecture to initialize. During phase 0, a thread is created to perform phase 1 initialization. Upon completion of phase 0 initialization, a context switch to the thread occurs and it proceeds with phase 1 initialization. The bulk of NT OS/2 is initialized during this phase. Upon completion of phase 1 initialization, system startup begins.

1.2 System Startup

System startup occurs in the NT OS/2 executive, and in the NT OS/2 Session Manager (SM).

1.2.1 Executive Level System System Startup

The executive's role in system startup is very simple. It simple creates a process to run SM. This is done using **RtlCreateUserProcess** with an image name of "\BootDevice\smss.exe". If the executive fails to create a process to run SM, then system startup fails. If this occurs on debug systems (compiled with DBG set), the NT OS/2 Command Line Interpreter (CLI) is started; otherwise, the system is halted with a bug check.

1.2.2 Session Manager System Startup

If executive level system startup is successful, then SM starts. SM begins by initializing itself. This includes:

- Creating ports for session manager, and subsystem APIs.
- Creating listen and API threads for session manager, and subsystem APIs.
- Enabling all connection requests and APIs.

Once SM is initialized, it begins system startup. Session Manager level system startup is driven from the NT OS/2 system configuration file located in "\BootDevice\ntos2.cfg". The file is a standard configuration file operated on by the "RtlxxxxConfigFile" APIs (described later in this paper).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Session Manager level system startup occurs in two phases. During the first phase, the "soft", or "configurable" portion of the NT OS/2 executive is configured. During the second phase, the session manager starts and initializes various protected subsystems.

1.2.2.1 First Phase SM system startup

First phase SM system startup begins with an open of the NT OS/2 system configuration file. This is done using **RtlOpenConfigFile** with a pathname of "\BootDevice\ntos2.cfg". If the open fails, then SM system startup does not occur. The system will run, but system level subsystems, and other portions of the system normally controlled by SM startup will not occur.

Once the configuration file is opened, the "ntos2" section is located using **RtlLocateSectionConfigFile** with a section name of "ntos2". If this section can not be located, then first phase SM system startup terminates and second phase SM system startup begins.

If the "ntos2" section is located, then all of the keywords in the section are enumerated and processed. Keywords are processed in configuration file order. Multiple keywords (keywords with the same keyword name) are processed in configuration file order.

SM reckognizes a small set of "ntos2" section keywords. The following sections describe the set of supported keywords.

1.2.2.1.1 Link Keyword

The "Link" keyword causes SM to create a symbolic link object accessible to all processes in the system. Link keywords are processed in configuration file order. The syntax of the link keyword is as follows:

```
Link = NAME-OF-LINK NAME-OF-LINK-TARGET
```

The "standard" set of symbolic link objects can be created by including the following in the "ntos2" section of your configuration file. Note that system initialization no longer creates these symbolic link objects. It only creates the "\BootDevice" symbolic link.

```
[ntos2]
//
// MIPS Links
//
// Link = \A: \Device\SaDisk
// Link = \C: \Device\HardDisk   // for 3240
// Link = \C: \Device\SaDisk     // for sable
//
//
// Simulator Links
//
// Link = \A: \Device\Floppy1
// Link = \B: \Device\Floppy2
// Link = \C: \Device\Filesystem
// Link = \D: \Device\Filesystem     // ddfs
// Link = \C: \Device\HardDisk1      // pinball
//
//
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
// Standard Links
//
Link = \A: \Device\Floppy0
Link = \C: \Device\HardDisk0\Partition1
Link = \D: \Device\HardDisk1\Partition1
Link = \E: \Device\HardDisk2
Link = \SystemDisk \C:
```

1.2.2.1.2 PagingFile Keyword

The "PagingFile" keyword causes SM to create a system wide file used by the modified page writer. If a paging file keyword does not exist in your system configuration file, then the amount of virtual memory available on the system is limited. System initialization no longer creates a paging file. The syntax of the pagingfile keyword is as follows:

```
PagingFile = NAME-OF-PAGING-FILE MAXIMUM-SIZE-OF-PAGING-FILE-MEGABYTES
```

The following example causes the system to use a 10Mb paging file called pagefile.sys in the "\Nt" directory of the system disk.

```
[ntos2]
//
// Create a 10Mb Paging File
//
PagingFile = \SystemDisk\Nt\pagefile.sys 10
```

Note that since the previous example uses the "\SystemDisk" symbolic link, it must occur after the appropriate link keyword.

1.2.2.2 Second Phase SM system startup

Second phase SM system startup begins after first phase SM startup completes. If first phase SM fails to open the NT OS/2 system configuration file, then this phase is skipped.

This phase begins by locating the "sm" section of the configuration file. This is done using **RtlLocateSectionConfigFile** with a section name of "sm". If this section can not be located, then second phase SM system startup terminates.

If the "sm" section is located, then all of the keywords in the section are enumerated and processed. Keywords are processed in configuration file order. Multiple keywords (keywords with the same keyword name) are processed in configuration file order.

SM reckognizes a small set of "sm" section keywords. The following sections describe the set of supported keywords.

1.2.2.2.1 Subsystem Keyword

The "SubSystem" keyword causes SM to start the specified system service emulation subsystem as a registerd subsystem. The subsystem must conform to the subsystem/SM connection protocol, and must accept appropriate proceses.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

When SM encounters a subsystem keyword, it creates a process to run the subsystem, and waits for the subsystem to complete the connection protocol.

The syntax of the subsystem keyword is as follows:

```
SubSystem = [debug ] PATHNAME-OF-SUBSYSTEM [, OPTIONAL-COMMAND-LINE]
```

Note that processes created through the SubSystem keyword must be marked as COFF_TARGET_SUBSYSTEM_NATIVE images.

The debug prefix is optional and if present invokes the subsystem with the DebugFlag passed to main set to 1. If the optional command line test is specified, then a pointer to the text after the comma will be passed in argv[1] to the main procedure of the subsystem.

The following example shows the subsystem keyword needed to start the OS/2 subsystem.

```
[sm]
//
// Start the os2 subsystem
//
SubSystem = \SystemDisk\Nt\SubSys\os2ss.exe
```

1.2.2.2.2 Start Keyword

The "Start" keyword causes SM to create and start a process. SM then waits for the initial thread in the process to terminate. Using this mechanism, you can write a server program whose initial thread performs all initialization then creates its worker threads. When the server is fully initialized and all worker threads are ready, the initial thread in the process could terminate. This allows SM to begin processing other keywords.

The syntax of the start keyword is as follows:

```
Start = [debug ] PATHNAME-OF-PROGRAM-TO-START [, OPTIONAL-COMMAND-LINE]
```

Note that processes created through the start keyword must be marked as COFF_TARGET_SUBSYSTEM_NATIVE images.

The debug prefix is optional and if present invokes the process with the DebugFlag passed to main set to 1. If the optional command line test is specified, then a pointer to the text after the comma will be passed in argv[1] to the main procedure of the process.

The following example causes SM to start the Presentation Manager (PM) and security servers.

```
[sm]
//
// Start PM
//
Start = \SystemDisk\Nt\SubSys\pmsrv.exe
//
// Start SAM
//
start = \SystemDisk\Nt\SubSys\samsrv.exe
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.2.2.2.3 Run Keyword

The "Run" keyword causes SM to create and start a process. SM does not wait for the process. It is simply created and set free to run. This keyword is useful to start logon processes and shells, and various "leaf" system processes.

The syntax of the run keyword is as follows:

```
Run = [debug ] PATHNAME-OF-PROGRAM-TO-RUN [, OPTIONAL-COMMAND-LINE]
```

Note that processes created through the run keyword must be marked as COFF_TARGET_SUBSYSTEM_NATIVE images.

The debug prefix is optional and if present invokes the process with the DebugFlag passed to main set to 1. If the optional command line test is specified, then a pointer to the text after the comma will be passed in argv[1] to the main procedure of the process.

The following example causes SM to run the PM Shell and NT OS/2 Error Logger.

```
[sm]
//
// Start PM Shell
//
Run = \SystemDisk\Nt\Bin\pmshell.exe
//
// Start Error Logger
//
Run = \SystemDisk\Nt\Bin\errlog.exe
```

1.2.2.2.4 Debug Keyword

The "Debug" keyword causes SM to start the debug subsystem and enable debugging.

The syntax of the debug keyword is as follows:

```
Debug = PATHNAME-OF-DEBUG-SUBSYSTEM
```

The following example causes SM to start the debug subsystem.

```
[sm]
//
// Start DBG
//
Debug = \SystemDisk\Nt\SubSys\dbgss.exe
```

1.2.2.2.5 LibPath Keyword

The "LibPath" keyword causes SM to establish a default DLL search path. This path is made available to the loader subsystem and is used to locate DLLs.

The syntax of the LibPath keyword is as follows:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
LibPath = DLL-SEARCH-PATH
```

The following example illustrates the use of the LibPath keyword:

```
[sm]
//
// Establish a DLL search path
//
LibPath = \BootDevice;\SystemDisk\Nt\Dll
```

1.2.2.2.6 Short Term Keywords

There are a number of keywords that are currently supported by SM, but are only supported due to shortcomings in the rest of the system. These keywords will remain until the appropriate components are complete and they are no longer needed. The following sections describe keywords that are supported, but have a limited lifetime.

1.2.2.2.6.1 GlobalFlag Keyword

The "GlobalFlag" keyword causes SM to set the value of *NtGlobalFlag*. This flag controls various debug portions of the system.

The syntax of the GlobalFlag keyword is as follows:

```
GlobalFlag = VALUE-FOR-NTGLOBALFLAG
```

The following example illustrates the use of the GlobalFlag keyword:

```
[sm]
//
// Setup NtGlobalFlag to display object deletion messages
// and to stop on first chance exceptions
//
GlobalFlag = 3
```

1.2.2.2.6.2 Path Keyword

The "Path" keyword causes SM to establish a default search path used when running programs from the NT SM> CLI.

The syntax of the Path keyword is as follows:

```
Path = DEFAULT-SEARCH-PATH
```

The following example illustrates the use of the Path keyword:

```
[sm]
//
// Setup a search path used at the NT SM> prompt
//
Path = \BootDevice;\SystemDisk\Nt\Bin
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.2.2.2.6.3 Quota Keywords

The quota keywords establish a default quota for processes started from the NT SM> CLI. Quota keywords can be used to establish:

- o Non-paged pool limit
- o Paged pool limit
- o Minimum working set size
- o Maximum working set size
- o Pagefile limit

If a configuration file does not specify a quota keyword, then the defult for that resource is unlimited. Quota keywords have the following syntax:

```
PagedPoolLimit = AMOUNT-OF-PAGED-POOL-QUOTA
NonPagedPoolLimit = AMOUNT-OF-PAGED-POOL-QUOTA
MinimumWorkingSetSize = MINIMUM-WORKING-SET-SIZE-IN-PAGES
MaximumWorkingSetSize = MAXIMUM-WORKING-SET-SIZE-IN-PAGES
PagefileLimit = MAXIMUM-PAGE-FILE-USAGE
```

The following example illustrates the use of the quota keywords:

```
[sm]
//
// Setup Quota
//
PagedPoolLimit = 1048576
NonPagedPoolLimit = 1048576
MinimumWorkingSetSize = 45
MaximumWorkingSetSize = 75
```

2. Configuration File APIs

The NT OS/2 user-mode DLL (udll.dll) contains a set of APIs used to manage configuration files. Configuration files are very similar to existing Microsoft configuration files (tools.ini, win.ini...).

Configuration files consist of sections. Within sections, there are keywords that have optional values. The following describes the format of a configuration file (note that "#", or "//" begin line comments).

```
//
// Section names are enclosed in "[" and "]". Section names are
// any valid C identifier. Section names are case insensitive.
//
[NAME-OF-SECTION]
// Keywords appear within sections.  Keyword names are any
// valid C identifier and are case insensitive.  Keywords
// may optionally contain a value.  A keyword's value is all
// characters to the right of the "=" character.  Leading
// and trailing whitespace is trimmed.  If a line comment
// occurs on a keyword line, the keyword's value ends at
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
// the start of the line comment with trailing whitspace
// trimmed.
NAME-OF-KEYWORD                     // keyword without a value
NAME-OF-KEYWORD = VALUE-OF-KEYWORD  // keyword with a value
```

The following sample shows a standard NT OS/2 configuration file.

```
[ntos2]
//
// Standard Links
//
Link = \A: \Device\Floppy0
Link = \C: \Device\HardDisk0\Partition1
Link = \D: \Device\HardDisk1\Partition1
Link = \E: \Device\HardDisk2
Link = \SystemDisk \C:
//
// Create a 10Mb Paging File
//
PagingFile = \SystemDisk\Nt\pagefile.sys 10
[sm]
//
// Set up global flag to show exceptions,
// LibPath to search hard disk, and Path
// to search BootDevice and HardDisk
//
GlobalFlag = 8
LibPath = \BootDevice;\SystemDisk\Nt\Dll
Path = \BootDevice;\SystemDisk\Nt\Bin
//
// Start the debug subsystem
//
Debug = \SystemDisk\Nt\SubSys\dbgss.exe
//
// Start the os2 subsystem
//
SubSystem = \SystemDisk\Nt\SubSys\os2ss.exe
//
// Setup Quota
//
PagedPoolLimit = 1048576
NonPagedPoolLimit = 1048576
MinimumWorkingSetSize = 45
MaximumWorkingSetSize = 75
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1 Configuration File Data Structures

The configuration file APIs export several data structures that are visible to it's users.

2.1.1 CONFIG_FILE

The CONFIG_FILE data structure is a "handle" to a configuration file. This data structure is opaque as far as it's users are concerned.

2.1.2 CONFIG_SECTION

The CONFIG_SECTION data structure is a "handle" to a configuration file section. It is used to enumerate and locate keywords within a section. This data structure is opaque as far as it's users are concerned.

2.1.3 CONFIG_KEYWORD

The CONFIG_KEYWORD is the only data structure that is not opaque. This data structure contains strings for the keyword's name and value. It also contains a pointer to the next keyword whose keyword value is the same.

```
typedef struct _CONFIG_KEYWORD {
    STRING Keyword;
    STRING Value;
    struct_CONFIG_KEYWORD *NextKeyword;
    struct_CONFIG_KEYWORD *LastKeyword;
} CONFIG_KEYWORD, *PCONFIG_KEYWORD;
```

<u>CONFIG_KEYWORD Structure:</u>

Keyword —Supplies the name of the keyword as a counted string. The *Keyword*.Buffer field is a NULL terminated string.

Value —Supplies the value of the keyword as a counted string. If the keyword has no value, then the *Length* and *MaximumLength* fields are zero, and the *Buffer* filed is NULL. Otherwise, *Value*.Buffer is a NULL terminated string.

NextKeyword —Supplies the address of the next keyword having the same keyword name. The end of the list contains a value of NULL. Keywords in this chain are in configuration file order.

LastKeyword —Opaque.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.4 Configuration File APIs

There are several Configuration File APIs.

2.1.4.1 RtlOpenConfigFile

A configuration file is opened using the following API:

NTSTATUS
RtlOpenConfigFile(
    IN PSTRING *ConfigFilePathname*,
    OUT PCONFIG_FILE **ConfigFile*
)

<u>Parameters:</u>

> *ConfigFilePathname* —Supplies the name of the configuration file to open.
>
> *ConfigFile* —Returns a handle to a configuration file.

<u>Return Value:</u>

> SUCCESS() —The configuration file was opened successfully.
>
> !SUCCESS() —A failure occured opening the configuration file, or the configuration file could not be properly parsed.

This function opens the specified configuration file and initializes all associated data structures. Configuration files are broken up into sections, each section contains keywords and associated values.

Sections are identified by a section name enclosed in braces appearing alone on a line. Each section contains zero or more keywords where a keyword is a keyword=value string.

2.1.4.2 RtlCloseConfigFile

VOID
RtlCloseConfigFile(
    IN PCONFIG_FILE *ConfigFile*
)

<u>Parameters:</u>

> *ConfigFile* —Supplies the address of the configuration file to close.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.4.3 RtlLocateSectionConfigFile

PCONFIG_SECTION
RtlLocateSectionConfigFile(
    IN PCONFIG_FILE *ConfigFile*,
    IN PSTRING *SectionName*
)

Parameters:

ConfigFile —Supplies the address of the configuration file to search for the specified section name.

SectionName —Supplies the section name to locate in the configuration file.

Return Value:

NULL —A matching section name was not found.

NON-NULL —Returns a pointer to the specified section.

This function locates the named section in the specified configuration file.

2.1.4.4 RtlLocateKeywordConfigFile

PCONFIG_KEYWORD
RtlLocateKeywordConfigFile(
    IN PCONFIG_SECTION *ConfigSection*,
    IN PSTRING *KeywordName*
)

Parameters:

ConfigSection —Supplies the address of the configuration file section to search for the specified keyword name.

KeywordName —Supplies the keyword name to locate in the configuration file section.

Return Value:

NULL —A matching keyword name was not found.

NON-NULL —Returns a pointer to the specified keyword.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function locates the named keyword in the the specified configuration file section. If multiple values of the same keyword exist, the *NextKeyword* field of the returned keyword points to the list of duplicate keywords.

2.1.4.5 RtlEnumerateSectionConfigFile

PCONFIG_SECTION
RtlEnumerateSectionConfigFile(
    IN PCONFIG_FILE *ConfigFile*,
    IN BOOLEAN *Restart*
)

Parameters:

ConfigFile —Supplies the address of the configuration file whose sections are to be enumerated.

Restart —Supplies a value that causes the enumeration to start at the beginning of the section list (TRUE), or continue from the last returned section (FALSE).

Return Value:

NULL —All sections have been returned.

NON-NULL —Returns a pointer to the next section.

This function enumerates all of the sections in the specified configuration file. To start at the beginning of the configuration file the *Restart* parameter is specified as TRUE, subsequent sections are returned with a *Restart* value of FALSE. A value of NULL is returned when all of the sections have been returned.

To enumerate the sections in a loop:

```
for(p=RtlEnumerateSectionConfigFile(ConfigFile,TRUE);
p;
p=RtlEnumerateSectionConfigFile(ConfigFile,FALSE) )
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1.4.6 RtlEnumerateKeywordConfigFile

PCONFIG_KEYWORD
RtlEnumerateKeywordConfigFile(
    IN PCONFIG_SECTION *ConfigSection*,
    IN BOOLEAN *Restart*
)

Parameters:

ConfigSection —Supplies the address of the configuration file section whose keywords are to be enumerated.

Restart —Supplies a value that causes the enumeration to start at the beginning of the keyword list (TRUE), or continue from the last returned keyword (FALSE).

Return Value:

NULL —All keywords have been returned.

NON-NULL —Returns a pointer to the next keyword.

This function enumerates all of the keywords in the specified configuration file section. To start at the beginning of the configuration file section the *Restart* parameter is specified as TRUE, subsequent keywords are returned with a *Restart* value of FALSE. A value of NULL is returned when all of the keywords have been returned.

Keywords having the same name are linked through the *NextKeyword* field. This function does not walk these links. For each keyword returned by this function, the caller must traverse the *NextKeyword* list to enumerate keywords with the same name.

To enumerate the sections in a loop:

```
for(p=RtlEnumerateKeywordConfigFile(ConfigFile,TRUE);
p;
p=RtlEnumerateKeywordConfigFile(ConfigFile,FALSE) )
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Status Code Specification

Author: *Darryl E. Havens*

Revision 1.0, June 11, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview 1

2. Definition 1

3. Use of Status Codes 2

4. Programming Interfaces 3
   4.1 Obtaining Information for Status Codes 3
   4.2 Determining Success or Failure 5
   4.3 Determining Success Severity 5
   4.4 Determining Information Severity 5
   4.5 Determining WARNING Severity 5
   4.6 Determining ERROR Severity 5
   4.7 Obtaining the Facility from a Status Code 6

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This specification describes the purpose, structure, and use of status codes for the **Windows NT** system.

In its simplest form, a *status code* is a value that is used to indicate whether an operation was successfully completed. If this were its only purpose, however, then all functions that returned a status indicator might simply return a boolean value of TRUE or FALSE. One of the values would indicate that the operation was successful, and the other would indicate that something went wrong. Which value was assigned which meaning would probably be arbitrary.

Systems today do not generally take this oversimplified approach when dealing with success and failure. Most systems are at least concerned with why a function incurred an error. Hence, systems generally have a "success" code, indicating that everything worked properly, and multiple failure codes, each indicating that an error occurred as well as providing some hint or clue as to what went wrong.

Windows NT takes this concept one step further and adds multiple success codes as well as multiple error codes. This allows the system to provide more information about what actually happened, rather than simply indicate that the function worked. For example, rather than just returning "success" with reason information, an *information* status code may be used. Likewise, rather than just returning "error" with reason information associated with it, **Windows NT** provides the ability to express a *warning* as well.

These types of status codes do not adversely affect the efficiency of the system; rather, they provide robustness. A programmer can still ask the basic question, "Was the function successful or not?"

Windows NT provides a common architecture for its status codes, which it uses throughout the native part of the system. That is, all functions that return status return the same type of status code. This common treatment of status values makes them easy to use and easy to understand.

Finally, **Windows NT** provides a mechanism to allow status codes to be local to a given *facility*, such as the kernel or the Session Manager. That is, each component has a separate *facility code*. In this way, message codes used by one facility will not be confused with codes used by another facility.

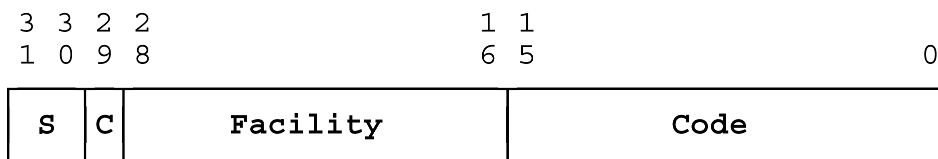
Having different facility codes also provides more useful information to the caller when it is important to determine exactly which part of the system encountered an error. For example, if the C runtime library incurs an error when opening a file, was it the runtime library itself that incurred the error, a subsystem that the runtime

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

called, the operating system, the file system, or a disk driver? This information is sometimes important.

2. Definition

A status code in **Windows NT** contains four fields. The following is the format of a status code:



where:

S —*Severity* field. This field represents the severity of the status code. The following values are defined:

00 —*Success*. This value means that the function was successful.

01 —*Information*. This value means that the function was successful and additional information about what happened is supplied.

10 —*Warning*. This value means that the function incurred an error that was not necessarily fatal.

11 —*Error*. This value means that the function incurred an error.

C —*Customer* field. This field is reserved to customers of Microsoft Corporation to allow them to define their own facility codes.

Facility —*Facility* field. This field indicates the facility from which the status code was issued.

Code —*Code* field. This field describes what actually took place.

All system-defined status codes are defined in the file *NTSTATUS.H*. Each status code has the format **STATUS_xxx**, where **xxx** is a short identifier that describes the meaning of the code. For example, the status code **STATUS_BUFFER_OVERFLOW** indicates, as the name suggests, that an overflow occurred while writing a buffer.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Names are chosen to be as descriptive as possible, making source code easier to read.

3. Use of Status Codes

Status codes in **Windows NT** are used throughout the system. They are returned by all system services to indicate whether the service completed successfully. In some cases, the service returns an alternate success code. That is, rather than return *STATUS_SUCCESS*, the normal success status code, another code may be used. An example of this is an I/O service that returns *STATUS_PENDING*. This indicates that the request was successfully made to the system, but it has not yet completed. While this code indicates that the request was successful, it also provides pertinent information about exactly what was successful.

All library routines in **Windows NT** that can return a success indicator also use status codes. They return status codes in the same format as the rest of the system.

Status codes may also be used in a call to the **NtRaiseException** service, provided the status code represents a warning or error. Information and success codes may not be raised. It should also be noted that if a warning status is raised, and no exception handler handles the exception, the default action is to *continue*. The default action for an unhandled error condition, however, is to terminate the thread. For more information, see the *Windows NT Exception Handling Specification*.

The following is an example utilization of the various types of status codes in a common search utility. For example, the following status codes might be used to indicate various completion statuses:

- *STATUS_SUCCESS* —This code, a *SUCCESS* code, might be used to indicate that all matches were successfully located and displayed.

- *STATUS_NO_MATCHES* —This code, an *INFORMATION* code, might be used to indicate that while no errors occurred, no strings were located that matched the search pattern string.

- *STATUS_BUFFER_OVERFLOW* —This code, a *WARNING* code, might be used to indicate that while a match was found, a buffer overflow occurred and the entire matching string could not be displayed. Notice that this is certainly a problem from which the utility can recover and therefore should not terminate the search.

- *STATUS_FILE_NOT_FOUND* —This code, an *ERROR* code, might be used to indicate that no files were found on which to perform the search. This is not a recoverable error condition. The program can do nothing but terminate.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Programming Interfaces

Windows NT provides the following services and C language macros to use with status codes:

NtQueryStatusCode —Return text associated with a specified status code.
SUCCESS —Return boolean value based on success or failure.
IS_SUCCESS —Return boolean TRUE if status code severity is success.
IS_INFORMATION —Return boolean TRUE if status code severity is information.
IS_WARNING —Return boolean TRUE if status code severity is warning.
IS_ERROR —Return boolean TRUE if status code severity is error.
FACILITY —Returns the facility code associated with a status value.

4.1 Obtaining Information for Status Codes

The message text associated with a status code may be obtained using the **NtQueryStatusCode** service:

NTSTATUS
NtQueryStatusCode (
IN NTSTATUS *StatusCode,*
OUT PVOID *MessageBuffer,*
IN ULONG *MessageLength,*
OUT PULONG *MessageReturnLength* **OPTIONAL,**
OUT PSEVERITY *SeverityLevel,*
OUT PVOID *FacilityBuffer* **OPTIONAL,**
IN ULONG *FacilityLength* **OPTIONAL,**
OUT PULONG *FacilityReturnLength* **OPTIONAL**
);

Parameters:

StatusCode —Supplies the status code value whose associated text should be returned.

MessageBuffer —Supplies a buffer into which the text for the status code is stored.

MessageLength —Supplies the number of bytes in the *MessageBuffer*.

MessageReturnLength —Optionally supplies a variable in which to return the length of the text that was written into *MessageBuffer*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SeverityLevel —Supplies a pointer to a variable that is to receive an enumerated type code representing the severity of the status code.

<u>SeverityLevel Values</u>

Success —Indicates that the severity of *StatusCode* is SUCCESS.

Information —Indicates that the severity of *StatusCode* is INFORMATION.

Warning —Indicates that the severity of *StatusCode* is WARNING.

Error —Indicates that the severity of *StatusCode* is ERROR.

FacilityBuffer —Optionally supplies a buffer into which the facility name associated with the status code is written.

FacilityLength —Optionally supplies the length of the *FacilityBuffer*. If the *FacilityBuffer* parameter is supplied, then this parameter must also be supplied.

FacilityReturnLength —Optionally supplies a variable in which to return the length of the text that was written into *FacilityBuffer*.

The **NtQueryStatusCode** service fetches the text associated with a specified status code and writes it into the supplied buffer. This allows programs to output explicit information about what happened during the execution of a function.

If no message text can be found for the specified status code, then the message text buffer will be set to the string, "NO MESSAGE TEXT", and an information status code of *STATUS_NO_MESSAGE* is returned. Likewise, if no text for the facility of the specified status code can be located, the string, "NOFACILITY", is written to the facility buffer, if one was supplied. If the country code of the process is non-English, the message text appears in the designated language.

The message text for all status codes defined for **Windows NT** can be obtained using the **NtQueryStatusCode** service. Text for user-defined status codes can also be obtained provided that the codes and text have been "added" to the system. For more information, see the *Windows NT Status Code Design Note*.

4.2 Determining Success or Failure

Whether a status code represents success or failure can be determined using the C macro, **SUCCESS**:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SUCCESS(Status)

This macro returns a BOOLEAN value of TRUE if the status code specified by *Status* represents a success or information severity. Otherwise the macro returns a BOOLEAN value of FALSE.

4.3 Determining Success Severity

Whether a status code represents success can be determined using the C macro, **IS_SUCCESS**:

IS_SUCCESS(Status)

This macro returns a BOOLEAN value of TRUE if the severity of the status code specified by *Status* is success. Otherwise the macro returns a BOOLEAN value of FALSE.

4.4 Determining Information Severity

Whether a status code represents information severity can be determined using the C macro, **IS_INFORMATION**:

IS_INFORMATION(Status)

This macro returns a BOOLEAN value of TRUE if the severity of the status code specified by *Status* is information. Otherwise the macro returns a BOOLEAN value of FALSE.

4.5 Determining WARNING Severity

Whether a status code represents warning severity can be determined using the C macro, **IS_WARNING**:

IS_WARNING(Status)

This macro returns a BOOLEAN value of TRUE if the severity of the status code specified by *Status* is warning. Otherwise the macro returns a BOOLEAN value of FALSE.

4.6 Determining ERROR Severity

Whether a status code represents error severity can be determined using the C macro **IS_ERROR**:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IS_ERROR(Status)

This macro returns a BOOLEAN value of TRUE if the severity of the status code specified by *Status* is error. Otherwise the macro returns a BOOLEAN value of FALSE.

4.7 Obtaining the Facility from a Status Code

Obtaining the facility number from a status code may be done using the C macro, **FACILITY**:

FACILITY(Status)

This macro returns a ULONG value which contains the *Facility* field of the status code specified by *Status*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, June 11, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Interlocked Support Routines Specification

Author: *David N. Cutler*

Original Draft 1.0, June 7, 1989Revision 1.1, June 8, 1989Revision 1.2, July 15, 1989Revision 1.3, January 15, 1990
Revision 1.4, June 9, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction.. 1
  1.1 Interlocked Add Functions 1
    1.1.1 Interlocked Add Unsigned Large Integer........................... 1
    1.1.2 Interlocked Add Unsigned Long 1
    1.2.3 Interlocked Add Unsigned Short 2
  1.2 Interlocked Doubly Linked List Functions 2
    1.2.1 Interlocked Insert Head Doubly Linked List 3
    1.2.2 Interlocked Insert Tail Doubly Linked List....................... 3
    1.2.3 Interlocked Remove Doubly Linked List 4
  1.3 Interlocked Singly Linked List Functions 4
    1.3.1 Interlocked Insert Head Singly Linked List 4
    1.3.2 Interlocked Remove Singly Linked List 5

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the interlocked support routines that are available for general use within the **Windows NT** executive. These routines can be used to provide properly synchronized access to nonpaged shared variables in a multiprocessor system.

These routines execute in kernel mode, raise IRQL to the highest level, synchronize with other processors using a spin lock, perform their operation, release the spin lock, and lower IRQL to its original value.

These routines cannot operate on paged data and are noninterruptable.

1.1 Interlocked Add Functions

Interlock add functions are provided to add unsigned short, long, and large integer values.

The addition is performed using a lock sequence so that access to the *Addend* variable is synchronized in a multiprocessor system.

1.1.1 Interlocked Add Unsigned Large Integer

An interlocked add operation can be performed on an integer of type **ULARGE_INTEGER** with the **ExInterlockedAddUlargeInteger** function:

```
ULARGE_INTEGER
ExInterlockedAddUlargeInteger (
    IN PULARGE_INTEGER Addend,
    IN ULARGE_INTEGER Increment,
    IN PKSPIN_LOCK Lock
    );
```

Parameters:

Addend - A pointer to a variable whose value is to be adjusted by the specified *Increment* value.

Increment - The increment value to be added to the specified *Addend* variable.

Lock - A pointer to a spin lock to be used to synchronize access to the *Addend* variable.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function performs an interlocked add of an *Increment* value to an *Addend* variable, and stores the result in the *Addend* variable. The initial value of the *Addend* variable is returned as the function value.

1.1.2 Interlocked Add Unsigned Long

An interlocked add operation can be performed on an integer of type **ULONG** with the **ExInterlockedAddUlong** function:

ULONG
ExInterlockedAddUlong (
    IN PULONG *Addend*,
    IN ULONG *Increment*,
    IN PKSPIN_LOCK *Lock*
    );

Parameters:

Addend - A pointer to a variable whose value is to be adjusted by the specified *Increment* value.

Increment - The increment value to be added to the specified *Addend* variable.

Lock - A pointer to a spin lock to be used to synchronize access to the *Addend* variable.

This function performs an interlocked add of an *Increment* value to an *Addend* variable, and stores the result in the *Addend* variable. The initial value of the *Addend* variable is returned as the function value.

1.2.3 Interlocked Add Unsigned Short

An interlocked add operation can be performed on an integer of type **USHORT** with the **ExInterlockedAddUshort** function:

USHORT
ExInterlockedAddUshort (
    IN PUSHORT *Addend*,
    IN USHORT *Increment*,
    IN PKSPIN_LOCK *Lock*
    );

Parameters:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Addend - A pointer to a variable whose value is to be adjusted by the specified *Increment* value.

Increment - The increment value to be added to the specified *Addend* variable.

Lock - A pointer to a spin lock to be used to synchronize access to the *Addend* variable.

This function performs an interlocked add of an *Increment* value to an *Addend* variable, and stores the result in the *Addend* variable. The initial value of the *Addend* variable is returned as the function value.

1.2 Interlocked Doubly Linked List Functions

Interlocked functions are provided to insert at the head of a doubly linked list, to insert at the tail of a doubly linked list, and to remove from the head of a doubly linked list. The list head for an interlocked doubly linked list can be initialized with the standard **InitializeListHead** function.

1.2.1 Interlocked Insert Head Doubly Linked List

An entry can be inserted at the head of an interlocked doubly linked list with the **ExInterlockedInsertHeadList** function:

```
VOID
ExInterlockedInsertHeadList (
    IN PLIST_ENTRY ListHead,
    IN PLIST_ENTRY ListEntry,
    IN PKSPIN_LOCK Lock
    );
```

Parameters:

ListHead - A pointer to the head of the doubly linked list into which an entry is to be inserted.

ListEntry - A pointer to the list entry to be inserted at the head of the list.

Lock - A pointer to a spin lock to be used to synchronize access to the interlocked list.

This function inserts an entry at the head of a doubly linked list so that access to the list is synchronized in a multiprocessor system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1.2.2 Interlocked Insert Tail Doubly Linked List

An entry can be inserted at the tail of an interlocked doubly linked list with the **ExInterlockedInsertTailList** function:

VOID
ExInterlockedInsertTailList (
  IN PLIST_ENTRY *ListHead*,
  IN PLIST_ENTRY *ListEntry*,
  IN PKSPIN_LOCK *Lock*
  );

Parameters:

ListHead - A pointer to the head of the doubly linked list into which an entry is to be inserted.

ListEntry - A pointer to the list entry to be inserted at the tail of the list.

Lock - A pointer to a spin lock to be used to synchronize access to the interlocked list.

This function inserts an entry at the tail of a doubly linked list so that access to the list is synchronized in a multiprocessor system.

1.2.3 Interlocked Remove Doubly Linked List

An entry can be removed from the head of an interlocked doubly linked list with the **ExInterlockedRemoveHeadList** function:

PLIST_ENTRY
ExInterlockedRemoveHeadList (
  IN PLIST_ENTRY *ListHead*,
  IN PKSPIN_LOCK *Lock*
  );

Parameters:

ListHead - A pointer to the head of the doubly linked list from which an entry is to be removed.

Lock - A pointer to a spin lock to be used to synchronize access to the interlocked list.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function removes an entry from the head of a doubly linked list so that access to the list is synchronized in a multiprocessor system. If there are no entries in the list, then a value of NULL is returned. Otherwise, the address of the entry that is removed from the list is returned as the function value.

1.3 Interlocked Singly Linked List Functions

Interlocked functions are provided to insert and remove from the head of a singly linked list. The list head for an interlocked singly linked list can be initialized by simply placing a value of NULL in the link pointer.

1.3.1 Interlocked Insert Head Singly Linked List

An entry can be inserted at the head of an interlocked singly linked list with the **ExInterlockedPushEntryList** function:

```
VOID
ExInterlockedPushEntryList (
    IN PSINGLE_LIST_ENTRY ListHead,
    IN PSINGLE_LIST_ENTRY ListEntry,
    IN PKSPIN_LOCK Lock
    );
```

Parameters:

ListHead - A pointer to the head of the singly linked list into which an entry is to be inserted.

ListEntry - A pointer to the list entry to be inserted at the head of the list.

Lock - A pointer to a spin lock to be used to synchronize access to the interlocked list.

This function inserts an entry at the head of a singly linked list so that access to the list is synchronized in a multiprocessor system.

1.3.2 Interlocked Remove Singly Linked List

An entry can be removed from the head of an interlocked singly linked list with the **ExInterlockedPopEntryList** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PSINGLE_LIST_ENTRY
ExInterlockedPopEntryList (
    IN PSINGLE_LIST_ENTRY *ListHead*,
    IN PKSPIN_LOCK *Lock*
    );

Parameters:

ListHead - A pointer to the head of the singly linked list from which an entry is to be removed.

Lock - A pointer to a spin lock to be used to synchronize access to the interlocked list.

This function removes an entry from the head of a singly linked list so that access to the list is synchronized in a multiprocessor system. If there are no entries in the list, then a value of NULL is returned. Otherwise, the address of the entry that is removed from the list is returned as the function value.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, June 7, 1989

Revision 1.1, June 8,1989

1. Added comments about nonpageability of data to the introduction.

2. Clarified functions as operating on singly or doubly linked lists.

3. Added section on the manipulation of singly linked interlocked lists.

Revision 1.2, July 15, 1989

1. Add interlocked add short routine.

Revision 1.3, January 15, 1990

2. Remove restriction that page faults cannot be taken during interlocked sequences.

Revision 1.4, June 9, 1990

1. Change the operation of the interlocked add long and short functions to be unsigned and change the names as appropriate.

2. Add a function to perform an interlocked add unsigned large integer operation.

3. Add text to warn users of these routines that page faults can not be tolerated.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Suspend/Resume Design Note

Author: *David N. Cutler*

Original Draft 1.0, February 9, 1989
Revision 1.1, March 30, 1989

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

This design note discusses a proposal to implement suspend and resume as part of the kernel rather than in the executive layer.

The suspension of a thread is controlled by a suspend count and a semaphore object that is built into the thread object. This semaphore has an initial value of zero and a maximum count of two (see explanation at end of this document as to why the maximum count must be two rather than one).

When an attempt is made to suspend a thread, the suspend count is incremented and a check is made to determine if the thread is already suspended (indicated by a nonzero initial suspend count). If the thread is not suspended, then a normal kernel **APC** is queued to the thread which will cause it to wait on its builtin semaphore.

A special case arises when the builtin **APC** is already queued to the target thread. This situation occurs when the target thread has been suspended and then resumed, but has never actually received the **APC** and suspended itself. Since the target thread has never actually suspended itself, the builtin semaphore count is decremented to indicate that the thread should suspend rather than resume.

The following pseudo code describes the logic of SuspendThread;

```
PROCEDURE SuspendThread (
     IN Tcb : POINTER KtThread;
     ) RETURNS integer;

VARIABLE

     OldCount : integer;

BEGIN

     Acquire dispatcher database lock;
     OldCount = Tcb.SuspendCount;
     IF Tcb.SuspendCount == 0 THEN
          IF NOT QueueApc(Tcb.SuspendAcb) THEN
               Tcb.SuspendSemaphore.Signal =
                    Tcb.SuspendSemaphore.Signal - 1;
          END IF;
     END IF;
     Tcb.SuspendCount = Tcb.SuspendCount + 1;
     Release dispatcher database lock;
     RETURN OldCount;
END SuspendThread;
```

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

Resuming a thread checks to determine if the thread has been suspended by examining the suspend count. If the thread has not been suspended, then no operation is performed. Otherwise the suspend count is decremented. If the resultant value is zero, then the target thread's builtin suspend semaphore is released.

The following pseudo code describes the logic of ResumeThread;

```
PROCEDURE ResumeThread (
     IN Tcb : POINTER KtThread;
     ) RETURNS integer;

VARIABLE

     OldCount : integer;

BEGIN

     Acquire dispatcher database lock;
     OldCount = Tcb.SuspendCount;
     IF Tcb.SuspendCount <> 0 THEN
          Tcb.SuspendCount = Tcb.SuspendCount - 1;
          IF Tcb.SuspendCount == 0 THEN
               Release Tcb.SuspendSemaphore;
          END IF;
     END IF;
     Release dispatcher database lock;
     RETURN OldCount;
END SuspendThread;
```

The maximum count of the builtin semaphore must be two so that the following race condition can be avoided.

1. a target thread is suspended by incrementing its suspend count to one and queuing its builtin suspend **APC**

2. before the thread can respond to the suspend **APC**, it is resumed which causes the suspend count to be decremented to zero and the builtin suspend semaphore to be incremented to one

3. the thread receives the suspend **APC**, but before it can wait on the builtin semaphore it is interrupted to deliver a special kernel **APC**

4. the special kernel **APC** code page faults and waits on the page to be brought into memory

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

5. the target thread is again suspended which causes its suspend count to be incremented and its builtin suspend **APC** to be queued

6. the thread is resumed before it has finished processing the special kernel **APC** which causes the suspend count to be decremented to zero and the builtin semaphore to be incremented to two

No additional nesting can occur since further attempts to queue the **APC** will fail which cause the semaphore count to be decremented. Thus the maximum count does not need to be greater than two.

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, February 9, 1989

Revision 1.1, March 30, 1989

1. Minor edits to conform to standard format.

[end of suspend.doc]

Copyright (c) Microsoft Corporation
Use subject to the Windows Research Kernel License

Portable Systems Group

NT OS/2 Time Conversion Specification

Author: *Gary D. Kimura*

Revision 1.2, August 14, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction.. 1

2. Converting From TIME to TIME_FIELDS 2

3. Converting From TIME_FIELDS to TIME 2

4. Converting From TIME to OS/2-Based Time 3

5. Converting From OS/2-Based Time to TIME 3

6. Converting From TIME to POSIX-Based Time 4

7. Converting From POSIX-Based Time to TIME 4

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the **NT OS/2** routines that implement absolute time conversion. Time, in **NT OS/2**, is expressed as a 64-bit signed large integer value with 100ns resolution and contained in variables of type **TIME**. Absolute times, such as August 29, 1989 at 11:08:30.245, are expressed as positive values (excluding zero), while time intervals (e.g., 1 hour and 20 minutes) are expressed as negative values (including zero).

The routines described in this document deal with absolute time and converting from a value of type **TIME** to a structure representing year, month, day, hour, minute, second, and millisecond, and back again. They do not address the manipulation of time intervals or time zones. Time interval manipulation is simply done using large integer arithmetic, and the time zone issue is beyond the realm of the package.

Though **NT OS/2** time has a 100ns resolution, the routines described here have only a one millisecond resolution. Time precision smaller than a millisecond is either ignored or lost by these routines.

The basis for **NT OS/2** time is the beginning of the year 1601 AD, which is chosen because 1601 AD is the start of a new quadricentury. The algorithms used in this package are based on the Gregorian calendar.

The structure **TIME_FIELDS** is used to represent a time value divided into its logical components. The declaration for **TIME_FIELDS** is the following:

```
typedef struct _TIME_FIELDS {
        CSHORT Year;
        CSHORT Month;
        CSHORT Day;
        CSHORT Hour;
        CSHORT Minute;
        CSHORT Second;
        CSHORT Milliseconds;
        CSHORT Weekday;
} TIME_FIELDS;
```

where:

Year - Denotes the year in a range between 1601 AD to around 30000 AD.

Month - Denotes the month in a range between 1 (January) to 12 (December).

Day - Denotes the day in the month with a range between 1 to either 28, 29, 30, or 31, depending on the month and the year.

Hour - Denotes the hour, on a 24 hour clock, in a range between 0 and 23.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Minute - Denotes the minute in a range between 0 and 59.

Second - Denotes the second in a range between 0 to either 59 or 60, where 60 denotes a leap second.

Milliseconds - Denotes the fraction of a second in a range between 0 and 999.

Weekday - Denotes the day of the week represented by the rest of the time fields in a range between 0 (Sunday) and 6 (Saturday). This field is only used when translating a 64-bit time value into a time field structure and not when mapping the other direction.

In addition to providing routines to convert between **TIME** and a **TIME_FIELDS** structure, this package also provides routines that convert between **TIME** and **OS/2** time and **POSIX** time. **OS/2** time is expressed as the number of seconds since the start of 1980. **POSIX** time is the number of seconds since the start of 1970.

The **APIs** that implement time conversion are the following:

RtlTimeToTimeFields - Converts a **TIME** value to a **TIME_FIELDS** structure.
RtlTimeFieldsToTime - Converts a **TIME_FIELDS** structure to a **TIME** value.
RtlTimeToSecondsSince1980 - Converts a **TIME** value to seconds with a 1980 base.
RtlSecondsSince1980ToTime - Converts seconds with a 1980 base to a **TIME** value.
RtlTimeToSecondsSince1970 - Converts a **TIME** value to seconds with a 1970 base.
RtlSecondsSince1970ToTime - Converts seconds with a 1970 base to a **TIME** value.

2. Converting From TIME to TIME_FIELDS

A **TIME** value is converted to a corresponding **TIME_FIELDS** structure with the **RtlTimeToTimeFields** procedure.

VOID
RtlTimeToTimeFields (
    IN PTIME *Time*,
    OUT PTIME_FIELDS *TimeFields*
    );

Parameters:

Time - Supplies the value being converted

TimeFields - A pointer to the variable being set

The input time can be any non-negative large integer value and is interpreted as the number of 100ns tics since the start of 1601 AD. The resulting TimeFields variable will never contain a leap second value of 60.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. Converting From TIME_FIELDS to TIME

A **TIME_FIELDS** structure is converted to a corresponding **TIME** value with the **RtlTimeFieldsToTime** procedure.

BOOLEAN
RtlTimeFieldsToTime (
    IN PTIME_FIELDS *TimeFields*,
    OUT PTIME *Time*
    );

Parameters:

TimeFields - Supplies the time field structure initialized by the caller to convert to a time value

Time - A pointer to the variable being set

The function result is TRUE if the input time fields is well formed and is expressible by a time variable and FALSE otherwise.

The input time must be well formed (i.e., the year must be 1601 or later, month must be between 1 and 12, day must be between 1 and the maximum day for the given month and year, hour must be between 0 and 23, minute must be between 0 and 59, second must be between 0 and 60 where the value 60 is only allowed during the last time in a month, and milliseconds must be between 0 and 999). The Weekday field is ignored by this procedure.

4. Converting From TIME to OS/2-Based Time

A **TIME** value is converted to the corresponding number of seconds since the start of 1980 with the **RtlTimeToSecondsSince1980** procedure.

BOOLEAN
RtlTimeToSecondsSince1980 (
    IN PTIME *Time*,
    OUT PULONG *ElapsedSeconds*
    );

Parameters:

Time - Supplies the value being converted, it must represent a time between 1980 AD and around 2115 AD

ElapsedSeconds - A pointer to the variable being set

The function result is TRUE if the input value is within the range expressible by the output value [1980 to 2115] and otherwise FALSE.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Converting From OS/2-Based Time to TIME

A **ULONG** value representing the number of elapsed seconds since the start of 1980 is converted to a corresponding **TIME** value with the **RtlSecondsSince1980ToTime** procedure.

VOID
RtlSecondsSince1980ToTime (
    IN ULONG *ElapsedSeconds*,
    OUT PTIME *Time*
    );

Parameters:

ElapsedSeconds - Supplies the value (i.e., number of seconds since the start of 1980) being converted

Time - A pointer to the variable being set

6. Converting From TIME to POSIX-Based Time

A **TIME** value is converted to the corresponding number of seconds since the start of 1970 with the **RtlTimeToSecondsSince1970** procedure.

BOOLEAN
RtlTimeToSecondsSince1970 (
    IN PTIME *Time*,
    OUT PULONG *ElapsedSeconds*
    );

Parameters:

Time - Supplies the value being converted, it must represent a time between 1970 AD and around 2105 AD

ElapsedSeconds - A pointer to the variable being set

The function result is TRUE if the input value is within the range expressible by the output value [1970 to 2105] and otherwise FALSE.

7. Converting From POSIX-Based Time to TIME

A **ULONG** value representing the number of elapsed seconds since the start of 1970 is converted to a corresponding **TIME** value with the **RtlSecondsSince1970ToTime** procedure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
VOID
RtlSecondsSince1970ToTime (
    IN ULONG ElapsedSeconds,
    OUT PTIME Time
    );
```

Parameters:

ElapsedSeconds - Supplies the value (i.e., number of seconds since the start of 1970) being converted

Time - A pointer to the variable being set

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, August 29, 1989

Revision 1.1, January 4, 1990

1. Included zero time as an interval time.
2. Make 60 a valid value in the second field of the TIME_FIELDS structure to handle leap seconds.

Revision 1.2, August 14, 1990

1. Fix procedure prototype for RtlTimeFieldsToTime to return a BOOLEAN result.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Timer Specification

Author: *David N. Cutler*

Original Draft 1.0, May 12, 1989
Revision 1.1, July 15, 1989
Revision 1.2, August 8, 1989
Revision 1.3, January 6, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction.. 1

2. Create Timer Object 1

3. Open Timer Object.. 2

4. Cancel Timer 2

5. Query Timer 3

6. Set Timer 4

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This specification describes the **Windows NT** *timer object* which is used to record the passage of time. A timer object is set to a specified time, and then expires when the time becomes due. When a timer object is set, its state is changed to Not-Signaled, and it is inserted in the timer queue according to its expiration time. When the timer expires, it is removed from the timer queue and its state is set to Signaled.

When a timer is set, an Asynchronous Procedure Call (**APC**) routine can optionally be specified. This routine is called asynchronously in the context of the establishing thread when the timer expires.

Waiting for a timer object causes the execution of the subject thread to be suspended until the timer attains a state of Signaled. Satisfying the Wait for a timer does not cause the state of the timer to change. Therefore, when a timer attains a Signaled state, an attempt is made to satisfy as many Waits as possible.

API's that support the timer object include:

> **NtCreateTimer** - Create a timer object and open a handle to it
> **NtOpenTimer** - Open a handle to existing timer object
> **NtCancelTimer** - Cancel a timer object that is set to expire
> **NtQueryTimer** - Get information about a timer object
> **NtSetTimer** - Set a timer object to expire at a specified time

2. Create Timer Object

A timer object can be created and a handle opened for access to the object with the **NtCreateTimer** function:

NTSTATUS
NtCreateTimer (
    OUT PHANDLE *TimerHandle*,
    IN ULONG *DesiredAccess*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**
    );

Parameters:

TimerHandle - A pointer to a variable that receives the timer object handle value.

DesiredAccess - The desired types of access to the timer object. The following type specific access flags can be specified in addition to the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

STANDARD_ACCESS_REQUIRED flags described in the Object Management Specification.

DesiredAccess Flags:

TIMER_QUERY_STATE - Query access to the timer object is desired.

TIMER_MODIFY_STATE - Modify access (set and cancel) to the timer object is desired.

SYNCHRONIZE - Synchronization access (wait) to the timer object is desired.

TIMER_ALL_ACCESS - All possible types of access to the timer object are desired.

ObjectAttributes - An optional pointer to a structure that specifies the object atributes; refer to the Object Management Specification for details.

If the *OBJ_OPENIF* flag is specified, and a timer object with the specified name already exists, then a handle to the existing object is opened, provided the desired access types can be granted. Otherwise, a new timer object is created with an initial state of Not-Signaled and a handle is opened to the new timer object.

3. Open Timer Object

A handle can be opened to an existing timer object with the **NtOpenTimer** function:

```
NTSTATUS
NtOpenTimer (
    OUT PHANDLE TimerHandle,
    IN ULONG DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes
        );
```

Parameters:

TimerHandle - A pointer to a variable that receives the timer object handle value.

DesiredAccess - The desired types of access to the timer object. The following type specific access flags can be specified in addition to the *STANDARD_ACCESS_REQUIRED* flags described in the Object Management Specification.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

<u>DesiredAccess Flags:</u>

TIMER_QUERY_STATE - Query access to the timer object is desired.

TIMER_MODIFY_STATE - Modify access (set and cancel) to the timer object is desired.

SYNCHRONIZE - Synchronization access (wait) to the timer object is desired.

TIMER_ALL_ACCESS - All possible types of access to the timer object are desired.

ObjectAttributes - A pointer to a structure that specifies the object atributes; refer to the Object Management Specification for details.

If the desired types of access can be granted, then a handle is opened to the specified timer object.

4. Cancel Timer

A timer can be cancelled with the **NtCancelTimer** function:

NTSTATUS
NtCancelTimer (
    IN HANDLE *TimerHandle,*
    OUT PBOOLEAN *CurrentState* **OPTIONAL**
    );

<u>Parameters:</u>

TimerHandle - An open handle to a timer object.

CurrentState - An optional pointer to a boolean variable that receives the current state of the timer object.

Canceling a timer object causes the timer to be removed from the timer queue if it is currently set, and returns the current state of the timer. If the current state of the timer object is Not-Signaled, then a value of **FALSE** is returned. Otherwise, the current state of the timer object is Signaled and a value of **TRUE** is returned.

Canceling a timer object that is not currently set to expire has no effect on the timer. Canceling a timer object also does not affect the state of the timer object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

5. Query Timer

The state of a timer can be queried with the **NtQueryTimer** function:

NTSTATUS
NtQueryTimer (
    IN HANDLE *TimerHandle*,
    IN TIMERINFOCLASS *TimerInformationClass*,
    OUT PVOID *TimerInformation*,
    IN ULONG *TimerInformationLength*,
    OUT PULONG *ReturnLength* **OPTIONAL**
    );

Parameters:

TimerHandle - An open handle to a timer object.

TimerInformationClass - The timer information class for which information is to be returned.

TimerInformation - A pointer to a buffer that will receive the specified information. The format and content of the buffer is dependent on the specified information class.

TimerInformation Format by Information Class:

TimerBasicInformation - Data type is *TIMERBASICINFO*.

TIMERBASICINFO Structure:

TIME *RemainingTime* - The amount of time remaining before the timer will expire.

BOOLEAN *TimerState* - The current state of the timer.

TimerInformationLength - Specifies the length in bytes of the timer information buffer.

ReturnLength - An optional pointer which, if specified, receives the number of bytes placed in the timer information buffer.

This function provides the capability to determine the state of a timer object and how much time remains before the timer will expire.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the current state of the timer object is Not-Signaled, then a value of **FALSE** is returned. Otherwise the current state of the timer object is Signaled and a value of **TRUE** is returned.

The remaining time is returned as the difference between the expiration time of the timer and the current system time. If the timer has already expired, then a negative time is returned which represents the amount of time that has lapsed since the timer expired. Otherwise, a positive value is returned that represents the amount of time remaining before the timer will expire.

6. Set Timer

A timer can be set to expire at a specified time with the **NtSetTimer** function:

NTSTATUS
NtSetTimer (
  IN HANDLE *TimerHandle*,
  IN PTIME *DueTime*,
  IN PTIMER_APC_ROUTINE *TimerApcRoutine* **OPTIONAL**,
  IN PVOID *TimerContext* **OPTIONAL**,
  OUT PBOOLEAN *PreviousState* **OPTIONAL**
  );

Parameters:

TimerHandle - An open handle to a timer object.

DueTime - The absolute or relative time at which the timer is to expire.

TimerApcRoutine - An optional pointer to a function that is called asynchronously when the timer expires. If this parameter is not specified, then the *TimerContext* parameter is ignored.

TimerContext - A pointer to an arbitrary data structure that is passed to the function specified by the *TimerApcRoutine* parameter. This parameter is ignored if the *TimerApcRoutine* parameter is not specified.

PreviousState - An optional pointer to a boolean variable that receives the previous state of the timer.

The function specified by the *TimerApcRoutine* parameter has the following type definition:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef
VOID
(*PTIMER_APC_ROUTINE) (
    IN PVOID TimerContext,
    IN ULONG TimerLowValue,
    IN LONG TimerHighValue
    );
```

Parameters:

TimerContext - A pointer to an arbitrary data structure which was specified when the timer was set.

TimerLowValue - The low half of the timer expiration time.

TimerHighValue - The high half of the timer expiration time.

Setting a timer object causes the absolute expiration time to be computed, the state of the timer set to Not-Signaled, and the timer object to be inserted in the timer queue.

If the timer is already in the timer queue, then it is implicitly canceled before it is set to the new expiration time.

The expiration time of the timer object is specified as either the absolute time that the timer is to expire, or a time relative to the current system time. If the value of the *DueTime* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

If an Asynchronous Procedure Call (**APC**) routine is specified, then the respective procedure is called in the context of the subject thread when the timer expires. The subject thread is also the only thread that can cancel the timer.

When the timer expires, it is removed from the timer queue and its state is set to Signaled.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, May 12, 1989

Revision 1.1, July 15, 1989

1. Change type name of timer **APC** routine.

2. Remove restriction that only the thread that set a timer with an **APC** routine could cancel the timer.

Revision 1.2, August 8, 1989

1. Change the output parameters of NtCancelTimer and NtSetTimer to be optional.

Revision 1.3, January 6, 1990

1. Change type name of object attributes parameter and refer to the Object Management Specification for the definition of this parameter.

2. Change the description of the desired access flags to include the standard rights, object specific rights, and generic rights.

3. Delete the handle flags and object name parameter from the **NtOpenTimer** service and replace with a pointer to an object attributes structure.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

NT Utilities Coding Conventions

Author: *David J. Gilman*

Revision 1.1, October 29, 1990
Revision 1.0, October 18, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction 1

2. The Existing Code Rule 1

3. Module Headers 1

4. Function Headers 3
  4.1 Modifiers 3
  4.2 Function Declarations 4
  4.3 Function Definitions 5

5. Header Files 6
  5.1 Header File Inclusion 6
    5.1.1 Description 6
    5.1.2 Special Header Files 7
      5.1.2.1 Ulibdef.hxx 7
      5.1.2.2 Ulib.hxx 7

6. Naming 8
  6.1 Variable Names 8
    6.1.1 Initial Caps Format 9
    6.1.2 Unstructured Format 9
  6.2 Data Type Names 9
  6.3 Returning or Accepting Pointers 11
  6.4 Structure Fields, Class Member Data and Enumeration Constants 11
  6.5 Macro and Constant Names 11

7. Indentation and Placement of Braces 12

8. Language Usage Guidelines 15
  8.1 Known Problems 15
  8.2 C++ Specific Guidelines 15
  8.3 Debugging Support 16

9. Appendix A - Example : Class EXAMPLE 16
  9.1 Ulib.hxx 16
  9.2 Example.hxx 17
  9.3 Examplep.hxx 18
  9.4 Example.inl 19
  9.5 Example.cxx 20
  9.6 Client.cxx 23

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This document describes the coding conventions that are used by the **NT OS/2 Utilities** group. Both the document and the conventions are heavily based on the document, "NT OS/2 Coding Conventions" written by Helen Custer and Mark Lucovsky.

There are primarily two reasons why the **NT OS/2 Utilities** group warrants a separate convention. First, work is done on existing code from many different sources. Second, all new code will be written in C++. This requires a number of changes and additions from the convention documented in the above mentioned document.

All code written for **NT OS/2 Utilities** adheres to a common coding style. This style gives the utilities a uniform appearance which allows group members to read, modify, and maintain each other's modules without learning several different coding conventions.

The following items are standardized:

- Module headers
- Function (member and non-member) headers and declarations
- Header file format
- Names of variables, data types (including classes), structure fields, macros, and constants
- Control structure indentation and placement of braces

2. The Existing Code Rule

When existing code is being ported to **NT OS/2**, every effort should be made to maintain the conventions and style that already exist in that code.

3. Module Headers

The following prototype should appear at the beginning of each module. The source to the prototype can be found in the file *\nt\public\oak\inc\modhdr.c.*

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    name-of-module-filename

Abstract:

    abstract-for-module.

Author:
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
    name-of-author (email-name) creation-date-dd-Mmm-yyyy

[Environment:]

    optional-environment-info (e.g. kernel mode only...)

[Notes:]

    optional-notes

Revision History:

    most-recent-revision-date email-name
        description
          .
          .
          .
    least-recent-revision-date email-name
    description

--*/
```

/ Note that no Revision History will be maintained until after the product has been released. /

The following is a sample of a completed module header:

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    object.hxx

Abstract:

    Definition of the root class for the ULIB class hierarchy.

Author:

    David J. Gilman (davegi) 12-Oct-1990

Environment:

    ULIB, User Mode

Notes:

    Note the PURE VIRTUAL functions.

--*/
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

\ The /++ <text> --*/ construct is used by a comment extractor program that will be developed to assist in our documentation efforts.*

4. Function Headers

In C++ member functions are declared within a class definition. These declarations contain a lot of information and as such will be enhanced by the use of *modifiers*. Some of these modifiers are also used by the function definition.

4.1 Modifiers

There are essentially three different types of modifiers; function specifiers, type specifiers and argument direction. All can be used by function declarations. Those that can also be used in function definitions are noted.

- All member function declarations are preceded by one of the following modifiers:

  VIRTUAL
  Indicates that the implementation of a member function can be overridden by a derived class

  NONVIRTUAL
  Indicates that the implementation of a member function can not be overridden by a derived class

  STATIC
  Indicates that the member function is static and therefore callable without an object instance of the class.

- Member function declarations may also be preceded by the following modifier:

  CONST (definition)
  Indicates that the function returns a constant value (usually a pointer).

- All formal arguments are preceded by one of the following modifiers:

  IN (definition)
  Indicates that the argument is a non-modifiable input value (i.e., call-by-value semantics)

  OUT (definition)
  Indicates that the argument is an address which refers to a variable or structure that will be modified by the function (i.e., call-by-reference semantics)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

IN OUT (definition)
Indicates that the argument is the address of an input variable or structure that is both read and written by the function (i.e., call-by-reference semantics).

- Formal arguments may also be followed by one of the following modifiers:

  OPTIONAL
  Indicates that an argument can be or NULL (or zero). To determine whether the actual value supplied is NULL, the programmer must use the macro ARGUMENT_PRESENT, which takes the argument and returns a value of type BOOLEAN. OPTIONAL arguments must be specified by the caller and can occur at any position in the argument list

  DEFAULT
  Indicates that the argument is optional and need not be specified by the caller. DEFAULT arguments may only occur at the end (*i.e.* right end) of an argument list and must be initialized in the class definition

- Member function declarations may also be followed by the following modifiers:

  CONST (definition)
  Indicates that the member function is *safe*. That is, it does not directly, or indirectly via a call, modify the object's state

  PURE
  Indicates that the member function is a pure virtual function. That is, all derived classes must supply their own implementation

- The order of the arguments in the comment block is the same as the order in which they appear in the function declaration.

4.2 Function Declarations

When a member function is declared in a class definition, it's declaration contains the function prototype and appropriate modifiers. For example:

```
NONVIRTUAL
CONST
POBJECT
GetNext (
    IN  POBJECT    LastObject,
      OUT BOOLEAN    WrapAround
    ) CONST
```

Note that modifiers, types and argument names should be aligned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.3 Function Definitions

Below is a prototype function definition and declaration. The definition form is to appear with the implementation of the function. The source to the prototype can be found in the file *\nt\public\oak\inc\prochdr.cxx*.

Note that a form-feed character should appear one line before the "return-type" line. This convention is noted in this document with the string "<form-feed>".

The function declaration follows:

```
<form-feed>
modifier
  .
  .
  .
return-type
procedure-name (
    direction type-name argument-name [modifier],
    direction type-name argument-name [modifier]
      .
      .
      .
    ) [modifier]

/*++

Routine Description:

    description-of-function.

Arguments:

    argument-name - Supplies (IN) | Returns (OUT) description of argument.
      .
      .

Return Value:

    return-value - Description of conditions needed to return value.
 - or -
    None.

--*/

{
  .
  .
  .
}
```

/ Note the space between the procedure name and the opening parenthesis for it's argument list. This is needed so that overloaded operators will be more readable. /

The following is a sample of a completed member function declaration:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
<form-feed>
NONVIRTUAL
CONST
POBJECT
COLLECTION::GetNext (
    IN POBJECT    LastObject OPTIONAL
    ) CONST

/*++

Routine Description:

    Get the next object from the collection.

Arguments:

    LastObject - Supplies the current object.

Return Value:

    POBJECT - A constant pointer to the next OBJECT in the
              COLLECTION.

--*/

{
  .
  .
  .
}
```

5. Header Files

The following sections define the requirements for inclusion and format of header files.

5.1 Header File Inclusion

5.1.1 Description

There are two types of header files used by the **NT OS/2 Utilities**:

- o Header files that are private to a single class:
  - o Types, constants etc.
  - o Inline functions
- o A public header file that contains the class declaration and associated types, constants etc.

The naming convention for private header files is <class-name>*p.h*xx. For example, the private header file for the object class, would be called *objectp.h*xx.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The public style of header files are the most important as they define class interfaces. An example can be found in section 0.

Header files should not be nested. That is, one header file should not include another.

5.1.2 Special Header Files

There are two special header files used by the ULIB class library: ulibdef.hxx and ulib.hxx. These files are exceptions to the nested include file rule.

5.1.2.1 Ulibdef.hxx

The file *\nt\private\os2\programs\ulib\inc\ulibdef.hxx* contains global information which is required by all classes and client's of ULIB. It should *not* be directly included. Rather, it will be included by *ulib.hxx*.

5.1.2.2 Ulib.hxx

The file *\nt\private\os2\programs\ulib\inc\ulib.hxx* is the master header file for the ULIB library. It should be included by all classes and clients of ULIB by the statement

```
#include "ulib.hxx"
```

In turn *ulib.hxx* will include, in the correct order, the header files that are needed by a particular class. This will be controlled by symbols of the form

```
_CLASSNAME_
```

which will be defined by the class client.

Class definitions will support this architecture by conditionally expanding to themselves, or to nothing if they have already been expanded.

As mentioned, class writers will use *ulib.hxx*. This will ensure that it is accurate and usable by any class clients. This means that special care should be taken to ensure that private header files are not listed within *ulib.hxx.*

In the example below, if the class definition in *collection.hxx* was not previously referenced, then the macro _COLLECTION_ is defined and the header file is expanded. Otherwise, _COLLECTION_ is already defined and the remainder of the header file is ignored. This results in the header file being included only once.

The following header file style should be used:

```
/*++
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Copyright (c) 1990  Microsoft Corporation

Module Name:

    object.hxx

Abstract:

    Definition of the abstract container class.

Author:

    David J. Gilman (davegi) 12-Oct-1990

Environment:

    ULIB, User Mode

Notes:

Revision History:

--*/

#if ! defined( _COLLECTION_ )
#define _COLLECTION_
   .
   .
   .

//
// body
//

#endif // _COLLECTION_
```

6. Naming

The following sections describe the naming conventions for variables, structure fields, types, constants, and macros.

6.1 Variable Names

Variable names are either in "initial caps" format, or they are unstructured. The following two sections describe when each is appropriate.

Note that the **NT OS/2** system, utilities included, do not use the Hungarian naming convention used in some of the other Microsoft products.

6.1.1 Initial Caps Format

All global variables and formal argument names must use the initial caps format. The following rules define this format:

- Words within a name are spelled out; abbreviations are discouraged.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o The first character of each word in a name is capitalized.
- o Acronyms are treated as words, that is, only the first character of the acronym is capitalized.

The following list shows some sample names that conform to these rules:

NumberOfBytes

TcbAddress

BilledProcess

6.1.2 Unstructured Format

Local variables may appear in either the initial caps format, or in a format of the programmer's preference. The following list shows some possibilities for local variable names:

loopindex

LoopIndex

loop_index

6.2 Data Type Names

A set of primitive data types for use in the **NT OS/2 Utilities** is defined in *ulibdef.hxx.* All **NT OS/2 Utilities** software must declare variables using these defined types rather than standard C++ types, where appropriate. The following are some examples of **NT OS/2 Utilities** types:

ULONG

PULONG

VOID

PVOID

BOOLEAN

PBOOLEAN

All new type names should be created in uppercase using **typedef**. Words within the name may either be packed together or separated by underscores. All types should have a corresponding **typedef** which defines a pointer and a reference to the type. The name for the pointer is the type name with a "P" prefix. Similarly the reference is the type name with a "R" prefix.

The following example illustrates how to use **typedef** to create a class type:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
typedef class COLLECTION : public OBJECT {

    public:

        NONVIRTUAL
        COLLECTION (
            IN ULONG    InitialNumberOfElements,
            IN ULONG    IncrementNumberOfElements DEFAULT = 10
            );

        VIRTUAL
        ~COLLECTION (
            );

        NONVIRTUAL
        POBJECT
        QueryNextElement (
            IN POBJECT    CurrentElement
            ) CONST;

        VIRTUAL
        CONST
        POBJECT
        GetNextElement (
            IN POBJECT    CurrentElement
            ) PURE;

    protected:

        POBJECT        mCollection;

    private:

        ULONG    _InitialNumberOfElements;
        ULONG    _IncrementNumberOfElements;
} POINTER_AND_REFERENCE_TYPES( COLLECTION );
```

Note that there should only be one *public:,* one *protected:* and one *private*: section in each class definition. In addition constructors and destructors should appear at the top of the list followed by logical groupings of other member functions.

C++ does not require a typedef for structures, and enumerated types as it considers them to be types when they are defined. However typedefs should be used so that a pointer and reference to the type are defined at the same time as the underlying type. For example,

```
typedef struct RANGE {
    ULONG    Start
    ULONG    Count;
} POINTER_AND_REFERENCE_TYPES( RANGE );


typedef enum COLLECTION_TYPE {
    Array,
    List,
    Table
} POINTER_AND_REFERENCE_TYPES( COLLECTION_TYPE );
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6.3 Returning or Accepting Pointers

In order to minimize performance impacts of using objects, the following conventions are used when pointers to objects, or other dynamic structures, are passed to and from an object:

- Member function names that have the prefix:
  - **Query**
    Return a pointer to an object which will be de-allocated by the client.
  - **Get**
    Return a constant pointer which will be de-allocated by the object.
  - **Set**
    Take a pointer to an object which will be de-allocated by the object.
  - **Put**
    Take a pointer to an object which will be de-allocated by the client.

6.4 Structure Fields, Class Member Data and Enumeration Constants

Notice from the above examples that structure field names, enumeration constants and class member data should follow initial caps format. They should not have field name prefixes tied to a type.

The subtle exception to this rule is for member data. The names used for a class' member data should be preceded by an '_' so that they can be more easily recognized in member function implementations.

6.5 Macro and Constant Names

All macros and manifest constants should have uppercase names. Words within a name may either be packed together, or separated by underscores.

The following statements illustrate some manifest constant and macro names:

```
#define PAGE_SIZE   4096
#define CONTAINING_RECORD(address, type, field) \
        ((type *)((LONG)(address) - \
                  (LONG)(&((type *)0)->field)))
```

Any macro that is likely to be replaced by a function at a later time should use the naming conventions for functions.

In C++ it is preferable to use constant variables and inline functions instead of manifest constants and macros.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7. Indentation and Placement of Braces

Source files should contain real tab characters. Tab stops should be set to four characters. This can be accomplished for the following tools by adding the following entry to the *tools.ini* file:

```
[pwb]
    entab:1
    filetab:4
    tabstops:4
    realtabs:yes

[list]
    tabamt:4

[ppr]
    flags = -e 4
```

/ The entries for list and ppr do not work. /

The following skeletal statements illustrate the proper indentation and placement of braces for C++ control structures.

```
<form-feed>
INT
FooBar(
    INT ArgumentOne,
    PULONG ArgumentTwo
    )

/*++

Routine Description:

    This is the routine description.

Arguments:

    ArgumentOne - Supplies the value for argument 1.

    ArgumentTwo - Supplies the address of argument 2.

Return Value:

    0 - Success

    1 - Failure

--*/

{
    //
    // Local variables are indented one tab (tabs are 4 spaces)
    //

    ULONG LocalVariable1;
    LONG Counter;

    //
    // for loops
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
//    - all for loops must have braces
//    - closing brace is at same indentation level as
//      for statement
//

for ( Counter = 0; Counter < 10; Counter++ ) {

    //
    // Body of loop
    //

}

//
// if statement
//

if ( Counter == 0 ) {

    //
    // Then statements
    //

}

//
// if then else
//

if ( Counter == 1 ) {

    //
    // Then statements
    //

} else {

    //
    // Else statements
    //

}

//
// switch statement
//

switch ( Counter ) {

case 1 :

    //
    // case 1 statements
    //
    break;

case 2 :

    //
    // case 2 statements
    //
    break;

default :
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
            //
            // default case
            //
            break;

        }
    }
```

8. Language Usage Guidelines

The **NT OS/2 Utilities** are written in portable C++ as defined by "The Annotated C++ Reference Manual" written by Margaret A. Ellis and Bjarne Stroustrup[1]. Care should be taken not to write any code that breaks with this language definition or with the ANSI C standard. When the two language definitions are at odds, side with the C++ definition.

8.1 Known Problems

There are two known problems that have been encountered by the **NT OS/2** group:

- Left Hand Side Typecasts

```
ULONG i;
( FLOAT ) i = 2.0;     // PROBLEM!
```

- Zero Length Arrays in Structures

```
struct X {
      ULONG i;
      ULONG arr[ ];     // PROBLEM!
};
```

Fortunately, C++ will not allow either of these constructs.

8.2 C++ Specific Guidelines

Following are a number of C++ specific guidelines which will aid in readability, consistency and debugging:

- File names should have the following extensions:
    - **hxx**
      for class definitions and related types and constants

[1] This book is also referred to as the "ANSI Base Document".

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

  - **inl**
    for inline function implementations

  - **cxx**
    for non-inline function implementations.

- In order to benefit from C++'s strong type checking:

  - Dynamic allocation and de-allocation should be performed with the C++ *new* and *delete* operators.

  - Constant, global variables should be used in lieu of pre-processor definitions.

- Do not declare inline functions within a class definition.

- Declare inline functions in the appropriate *.inl* file as described above and as shown in 0.

- Do not use multiple inheritance.

- Avoid using global, static objects.

- Do not use the C++ specific form of casting (*i.e.* ULONG(x)).

- Do not declare protected member data (use private data and access member functions).

8.3 Debugging Support

Debug code is enabled by the compiler symbol *DBG*. Debug code should not be defined within the body of non-debug code. Instead a macro should be defined which conditionally compiles to a, possibly inlined, function call. For example (from *ulibdef.hxx*),

```
#if defined( DBG )
     #define DebugAssert( b )    DbgAssert( b )
#else
     #define DbgAssert
#endif // DBG
```

Programmers should use the symbol *REGISTER* instead of the C++ storage specifier, *register*. This will disable register storage when *DBG* is enabled.

The macro, *INLINE_INCLUDE*, should be used to conditionally (depending on *DBG*) include (or compile) inline functions. See 0 for an example. Note that usage of this macro will cause the *DBG* symbol to effect the list of source files to be compiled. This macro will make tracing and stepping of inline functions easier.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

9. Appendix A - Example : Class EXAMPLE

9.1 Ulib.hxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    ulib.hxx

Abstract:

    Master include file for the ULIB class hierarchy.

Author:

    David J. Gilman (davegi) 19-Oct-1990

Environment:

    ULIB

Notes:

Revision History:

--*/

#include "ulibdef.hxx"
#include "object.hxx"
  .
  .
  .
#if defined( _EXAMPLE_ )

//
//  include files that the EXAMPLE class definition (not
//  implementation) is dependent upon
//

    #include "example.hxx"

#endif // _EXAMPLE_
```

9.2 Example.hxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    example.hxx

Abstract:

    Definition for class EXAMPLE.
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Author:

    David J. Gilman (davegi) 19-Oct-1990

Environment:

    ULIB

Notes:

Revision History:

--*/

#if ! defined( _EXAMPLE_ )
#define _EXAMPLE_

typedef class EXAMPLE : public OBJECT {

    public:

        NONVIRTUAL
        EXAMPLE (
            IN ULONG    Value
            );

        VIRTUAL
        ~EXAMPLE (
            );

        VIRTUAL
        ULONG
        SetValue (
            IN ULONG    Value
            );

        NONVIRTUAL
        ULONG
        QueryValueDoubled (
            ) CONST;

    private:

        ULONG    mValue;

} POINTER_AND_REFERENCE_TYPES( EXAMPLE );

INLINE_INCLUDE( example.inl );

#endif // _EXAMPLE_
```

9.3 Examplep.hxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    examplep.hxx
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Abstract:

    Private header file for class EXAMPLE.

Author:

    David J. Gilman (davegi) 19-Oct-1990

Environment:

    ULIB

Notes:

Revision History:

--*/

#if ! defined( _EXAMPLE_P )
#define _EXAMPLE_P

CONST DOUBLEVALUE = 2;

#endif // _EXAMPLE_P
```

9.4 Example.inl

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    example.inl

Abstract:

    Inline functions for class EXAMPLE.

Author:

    David J. Gilman (davegi) 19-Oct-1990

Environment:

    ULIB

Notes:

Revision History:

--*/

#define _EXAMPLE_
#include "ulib.hxx"

#include "examplep.hxx"

<form-feed>

ULONG
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
EXAMPLE::QueryValueDoubled (
    ) CONST

/*++

Routine Description:

    Compute double the value.

Arguments:

    None.

Return Value:

    ULONG - double the value.

--*/

{
    return( mValue * DOUBLEVALUE );
}
```

9.5 Example.cxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    example.cxx

Abstract:

    Implementation for class EXAMPLE.

Author:

    David J. Gilman (davegi) 19-Oct-1990

Environment:

    ULIB

Notes:

Revision History:

--*/

#define _EXAMPLE_
#include "ulib.hxx"

#include "examplep.hxx"


<form-feed>

EXAMPLE::EXAMPLE (
    ULONG    Value
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
    )

/*++

Routine Description:

    Construct an EXAMPLE object.

Arguments:

    Value - Initial value for the EXAMPLE object.

Return Value:

    None.

--*/

{
    mValue = Value;
}


<form-feed>

EXAMPLE::~EXAMPLE (
    )

/*++

Routine Description:

    Destroy an EXAMPLE object.

Arguments:

    None.

Return Value:

    None.

--*/

{
    DbgPrint( "Example destroying...\n" );;
}


<form-feed>

ULONG
EXAMPLE::SetValue (
    ULONG    Value
    )

/*++

Routine Description:

    Set an EXAMPLE's value.
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Arguments:

    Value - The value to set in EXAMPLE.

Return Value:

    ULONG - The set value.

--*/

{
    return( mValue = Value );
}
```

9.6 Client.cxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    client.cxx

Abstract:

    Sample usage of class EXAMPLE.

Author:

    David J. Gilman (davegi) 19-Oct-1990

Environment:

    ULIB

Notes:

Revision History:

--*/

extern "C" {
    #include <stdio.h>
};

#define _EXAMPLE_
#include "ulib.hxx"


<form-feed>

VOID
main (
    )

/*++

Routine Description:

    Constructs and demonstrates usage of an EXAMPLE object.
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
Arguments:

    None.

Return Value:

    None.

--*/

    EXAMPLE    example = 4;
    PEXAMPLE   pexample;

    pexample = &example;

    printf( "Value = %d\n", example.QueryValueDoubled( ) / 2 );
    printf( "Value = %d\n", pexample->QueryValueDoubled( ) / 2 );
}
```

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Revision 1.1, October 29, 1990 - djg

1. Changed definition of IN argument modifier.
2. Added IN OUT argument modifier.
3. Added POINTER_AND_REFERENCE_TYPES macro.
4. Added STATIC member modifier.
5. Clarified OPTIONAL versus DEFAULT argument modifiers.
6. Changed member data prefix from 'm' to '_'.
7. Added style guidelines for public, private and protected sections.
8. Miscellaneous edits for clarity.

Revision 1.0, October 18, 1990 - djg

1. Incorporated comments from stever and loup.
2. Added reference types.
3. Fixed formatting errors.

Original Draft, October 16, 1990 - djg

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Virtual Memory Specification

Author: *Lou Perazzoli*

Original Draft 1.0, December 15, 1988
Revision 4.0 April 28, 1993

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Windows NT Virtual Memory Specification

1. Overview	1
1.1 Object Orientation	1
1.2 Virtual Memory	1
1.3 Page Protections	2
1.4 Page File Quota and Commitment	3
2. Virtual Memory Operations	3
2.1 Create Section	4
2.2 Open Section	7
2.3 Map View Of Section	8
2.4 Extend Size Of Section	12
2.5 Unmap View Of Section	13
2.6 Allocate Virtual Memory	14
2.7 Free Virtual Memory	17
2.8 Read Virtual Memory	19
2.9 Write Virtual Memory	20
2.10 Flush Virtual Memory	21
2.11 Lock Virtual Memory	22
2.12 Unlock Virtual Memory	23
2.13 Protect Virtual Memory	24
2.14 Query Virtual Memory	26
2.15 Query Section Information	29
2.16 Create Paging File	31
2.17 Flush Instruction Cache	31
2.18 Flush Write Buffer	32
2.19 Close Handle	32

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Windows NT Virtual Memory Specification

1. Overview

This specification describes the virtual memory component for the portable New Technology (**Windows NT**) system. **Windows NT** virtual memory includes the following:

- Virtual memory support for the **POSIX** *fork* and *exec* operations, which enable compliance with the **POSIX** standard.
- Mapping of files into virtual memory and paging directly to/from those files. Files larger than 2 Gb are mapped via partial views of the file.
- Protection of shared memory and mapped files via Access Control Lists (**ACL**s), which is required to achieve a **DOD** security rating of **C2** or higher.
- Application control of virtual address space allocation and the mapping of shared memory.
- Copy-on-write pages with the ability to establish guard pages and set page protection.
- Creation of committed and/or reserved private memory without creating any kind of memory object.

1.1 Object Orientation

The basic architecture of the **Windows NT** system is object based. This means that all operating system abstractions presented at the API level are in the form of objects and a set of operations on those objects. This allows a stylized set of operations for each object, uniform naming across objects, uniform protection of objects, and uniform sharing of objects.

Typically there is an operation to create a new instance of an object (**NtCreate_***object)* and to establish access (create a handle) for an existing object (**NtOpen_***object)*. These basic operations are generally augmented by a set of object-specific operations. A handle is closed with a generic close operation (**NtClose**).

The treatment of objects here is minimal. A separate specification, *Windows NT Object Management,* more fully covers the object orientation of **Windows NT**.

1.2 Virtual Memory

Virtual memory is supported in the **Windows NT** system by section objects, a set of operations that may be performed on section objects, and various other services that directly manipulate the process virtual address space. In addition to section objects and their corresponding services, a set of operations are also provided to reserve and commit virtual memory private to a process.

A section object is a shareable entity that can be mapped into the virtual address space of a process. It can be backed by a paging file (e.g., demand zero pages) or by a file (mapped file).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Mapping a section into the virtual address space creates a process-private *view* of the section. The view can be partial or complete. Several different views of a section can be concurrently mapped within the same or different processes. When a view of a section is created, the corresponding process virtual address space is reserved and optionally committed.

Views are allocated on a hardware dependent **Allocation Granularity** (64kb on x86 and MIPS) virtual address boundary. The allocation granularity is determined by cache coherency issues and the desire to support larger page sizes in a compatible manner.

In general, it is not desirable for programs to directly control the allocation of the process virtual address space. However, the system has a need for this capability when a program is activated (i.e., the program file is mapped into the address space when the image is started), and some applications that use tag bits in pointers also want to control placement so that certain address bits are guaranteed to be zero. Thus the proposed interface provides for placement control, but it is optional.

Each section can have an optional name and Access Control List (**ACL**). This provides the basis whereby a section can be shared in a controlled manner. An unnamed section is a private section and can only be shared with another process via inheritance (i.e., the fork mechanism required to support **POSIX** compliance). Named sections can be shared by any other process that has access to the section.

The operating system does not use views as protection domains, and therefore never checks to ensure that an argument data structure resides within a single view. Thus an argument to a **Windows NT** service may span one or more views or private pages.

The default base address of all program images is the **Allocation Granularity** (zero-based program images do not allow uninitialized pointers to manifest themselves as access violations and are more difficult to debug). However, the base may be explicitly set to any desired value.

1.3 Page Protections

The virtual memory services allow the specification of *execute* access for page protections. On hardware which does not support execute access, the page protections for execute access will be treated as read. Therefore execute-only access would be treated as read-only, execute-read-write would be treated as read-write, etc. However, in the query operations, the actual set page protection would be returned.

1.4 Page File Quota and Commitment

The memory management system keeps track of page file usage on a global basis, termed *commitment*, and on a per process basis as *page file quota*. Commitment and page file quota are charged whenever virtual memory is created which requires backing store from the paging file.

The following explains the actions for each service which potentially creates pages destined for the paging file:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NtCreateSection (mapping file) __No commitment or page file quota is charged.

NtCreateSection (paging file) __Charge commitment for any committed pages within the section. The commitment is returned when the section is deleted.

NtAllocateVirtualMemory (reserve) __Charge commitment and page file quota for the page table pages required to map the potentially committed pages.

NtAllocateVirtualMemory (reserve & commit) __Charge commitment and page file quota for both the page table pages required to map the virtual memory and committed pages.

NtAllocateVirtualMemory (commit private pages) __Charge commitment and page file quota for each page of memory committed.

NtAllocateVirtualMemory (commit shared pages) __Charge page file quota and commitment if page protection is write-copy. Charge commitment if the page is within a view of a paging file backed section.

NtMapViewOfSection (mapping file) __Charge commitment and page file quota for the page table pages required to map the virtual memory. If the protection of the section is write-copy, charge page file quota and commitment for all pages in the view.

NtMapViewOfSection (paging file) __Charge page file quota as though all pages in the section are committed. Charge commitment and page file quota for the page table pages required to map the virtual memory. If the protection of the section is write-copy, charge page file quota and commitment for all pages in the view.

NtProtectVirtualMemory (within a view) __If the page protection is write-copy, charge commitment and page file quota for each newly protected page which is not already write-copy or private. If the page protection is not write-copy, and the previous page protection was write-copy, return the commitment and page file quota for that page.

NtFreeVirtualMemory __Returned the charged commitment and page file quota.

2. Virtual Memory Operations

The following subsections describe the virtual memory operations that can be performed in the **Windows NT** system. A definition and an explanation of each operation is given.

The **API**s described include:

NtCreateSection - Create section and open handle
NtOpenSection - Open handle to existing section
NtMapViewOfSection - Map view of section

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NtExtendSection - Extend the size of section
NtUnmapViewOfSection - Unmap view of section
NtAllocateVirtualMemory - Commit/reserve region
NtFreeVirtualMemory - Decommit/release region
NtReadVirtualMemory - Read memory from specified process
NtWriteVirtualMemory - Write memory to specified process
NtFlushVirtualMemory - Flush modified pages to file
NtLockVirtualMemory - Lock region process/system
NtUnlockVirtualMemory - Unlock region process/system
NtProtectVirtualMemory - Protect region
NtQueryVirtualMemory - Get information about region
NtQuerySection - Get information about section
NtCreatePagingFile - Create a paging file
NtFlushInstructionCache - Flushes the instruction cache.
NtFlushWriteBuffer - Flushes the write buffer on the current processor.
NtClose - Close handle

Each **API** returns a status value (error code) that signifies the success or failure of the operation.

2.1 Create Section

A section object can be created and a handle opened for access to the section with the **NtCreateSection** function:

NTSTATUS
NtCreateSection (
    OUT PHANDLE *SectionHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
    IN PLARGE_INTEGER *MaximumSize* **OPTIONAL**,
    IN ULONG *SectionPageProtection,*
    IN ULONG *AllocationAttributes,*
    IN HANDLE *FileHandle* **OPTIONAL,**
    );

Parameters:

SectionHandle - A pointer to a variable that will receive the section object handle value.

DesiredAccess - The desired types of access for the section. The following object type specific access flags can be specified in addition to the *STANDARD_ACCESS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

SECTION_MAP_EXECUTE - Execute access to the section is desired.

SECTION_MAP_READ - Read access to the section is desired.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SECTION_MAP_WRITE - Write and read access to the section is desired.

SECTION_QUERY - Query access to the section is desired.

SECTION_EXTEND_SIZE - The ability to extend the size of the section is desired.

ObjectAttributes - An optional pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details.

MaximumSize - A pointer to the maximum size of the section in bytes. For page file backed sections, this value is rounded up to the host page size. If this argument is unspecified or the value is specified as zero, and a file handle is specified, the section size is set to the size of the file.

SectionPageProtection - Specifies the underlying page protection for the section. For files mapped as images, this parameter is ignored and the underlying page protection is taken from the mapped file's image header.

Section Page Protection Values

PAGE_READONLY - Read access to the committed region of pages is allowed. An attempt to write or execute the committed region results in an access violation.

PAGE_READWRITE - Read, and write access to the region of committed pages are allowed. If write access to the underlying section is allowed, then a single copy of the pages is shared. Otherwise the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_EXECUTE - Execute access to the committed region of pages is allowed. An attempt to read or write the committed region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute, and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

AllocationAttributes - A set of flags that describes the allocation attributes of the section. One of *SEC_RESERVE, SEC_COMMIT* or *SEC_IMAGE* must be

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

supplied. If *SEC_IMAGE* is specified the only other valid option is *SEC_BASED*.

AllocationAttributes Flags

SEC_BASED - The section is a based section. Attempt to find a location that allows mapping in the current process which does not conflict with other SEC_BASED sections. If a view to a SEC_BASED section cannot be mapped at the specified address in the process and error is returned. **NOTE: SEC_BASED does not prevent other mappings or allocations from colliding with the based section, it merely guarantees that either the section is mapped at the based address or an error is returned.**

SEC_RESERVE - All pages of the section are set to the reserved state.

SEC_COMMIT - All pages of the section are set to the commit state.

SEC_IMAGE - The file specified by the file handle is an executable image file.

SEC_NOCACHE - All pages of the section are to be set as non-cacheable.

FileHandle - An optional handle of an open file object. If the value of this handle is NULL, then the section is backed by a paging file. Otherwise, the section is backed by the specified data file.

Creating a section creates an object that describes a region of potentially shareable memory and opens a handle for access to the section object. The section can be backed by a paging file or a specified data file. An open section handle can be used to map a view of the section into the virtual address space of the subject process.

If the section is given a name, then it can be shared at any virtual address with other processes that can open the section (see **NtOpenSection** below). The section can also be specified as "based" in which case it can also be shared at a fixed address in all processes that map a view of the section.

If the section is shareable (i.e., it is given a name), then the Access Control List (**ACL**) specifies which users can access the section. If the section is not given a name, then only the creating process and its descendants can access the section.

Various object attributes can be chosen for the section such that access to the section can be inherited by the child process when a new process is created. This capability is required to support the **POSIX** standard.

The *OBJ_OPENIF* object attribute allows the section to be created if a section object with the specified name doesn't already exist. This is useful when two or more processes dynamically create a temporary section to hold shared data while one or more processes that operate on the shared data are active. If this option is specified and a section object with the same name already exists, then the desired access to

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

the section object by the subject process is verified and an open handle is returned for the existing object.

A section can be specified as temporary or permanent. A temporary object is deleted when the last open handle to the object is closed. This can result from closing the handle (see **NtClose** below) or by terminating a process. A permanent object is deleted by first opening a handle to the object, marking it temporary, and then closing the handle. The object then behaves much like a temporary object and is deleted when the last open handle is closed.

If the section is mapped by a file, then the **ACL** on the file is used to control access to the section unless the user ID of the subject process is the owner of the file, in which case the specified **ACL** is used. The desired access types must be allowed by the section **ACL** and must be compatible with the open mode of the file (i.e., write access is not allowed to a file that is opened for read-only access).

If the file is open for read-write access, then the file acts as backing store for both reads and writes of pages in the section. Otherwise, the file is used for inpaging and no outpaging to the file occurs (i.e., any modified pages are written to a paging file).

In addition to quota errors and object management errors associated with creating objects, the following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_INVALID_PAGE_PROTECTION* - Error, an invalid page protection was specified.
- *STATUS_INVALID_FILE* - Error, an invalid file handle was specified.
- *STATUS_NOT_IMAGE* - Error, an attempt to map file as an image which is not an image file.
- *STATUS_SECTION_TOO_BIG* - Error, an attempt to map create a section which is bigger than the file which it backs.

2.2 Open Section

A handle can be opened for access to an existing section object with the **NtOpenSection** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
NTSTATUS
NtOpenSection(
    OUT PHANDLE SectionHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes
    );
```

Parameters:

Sectionhandle - A pointer to a variable that will receive the section object handle value.

DesiredAccess - The desired types of access for the section. The following object type specific access flags can be specified in addition to the *STANDARD_ACCESS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

SECTION_MAP_EXECUTE - Execute access to the section is desired.

SECTION_MAP_READ - Read access to the section is desired.

SECTION_MAP_WRITE - Write and read access to the section is desired.

SECTION_QUERY - Query access to the section is desired.

SECTION_EXTEND_SIZE - The ability to extend the size of the section is desired.

ObjectAttributes - A pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details.

Opening a section causes a handle for the object to be opened so that a view of the section can be mapped into the virtual address space of the subject process.

A process cannot open a section object unless the desired access types are allowed by the section object **ACL**, and, if the section is backed by a data file, are also compatible with the open mode of the associated data file.

In addition to quota errors and object management errors associated with opening objects, the following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.

2.3 Map View Of Section

A view of a section can be mapped into the virtual address space of a subject process with the **NtMapViewOfSection** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

NTSTATUS
NtMapViewOfSection(
IN HANDLE *SectionHandle,*
IN HANDLE *ProcessHandle,*
IN OUT PVOID **BaseAddress,*
IN ULONG *ZeroBits,*
IN ULONG *CommitSize,*
IN OUT PLARGE_INTEGER *SectionOffset* **OPTIONAL**,
IN OUT PULONG *ViewSize,*
IN SECTION_INHERIT *InheritDisposition,*
IN ULONG *AllocationType,*
IN ULONG *Protect*
);

Parameters:

SectionHandle - An open handle to a section object.

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the view. If the initial value of this argument is not NULL, then the view is allocated starting at the specified virtual address must be a multiple of the allocation granularity. If the initial value of this argument is NULL, then the operating system determines where to allocate the view using the information specified by the *ZeroBits* argument value and the section allocation attributes (i.e., SEC_BASED).

ZeroBits - The number of high-order address bits that must be zero in the base address of the section view. The value of this argument must be less than 21 and is only used when the operating system determines where to allocate the view (i.e., when *BaseAddress* is NULL).

CommitSize - The size of the initially committed region of the view in bytes. *CommitSize* is only meaningful for page-file backed sections, mapped sections, both data and image are always committed at section creation time and is ignored for mapped files. This value is rounded up to the next host-page-size boundary.

SectionOffset - Optionally supplies a pointer to the offset from the beginning of the section to the view in bytes. This value must be a multiple of allocation granularity. If the section was created with the *SEC_IMAGE*, this argument must be NULL.

ViewSize - A pointer to a variable that will receive the actual size in bytes of the view. If the value of this argument is zero, then a view of the section will be mapped starting at the specified section offset and continuing to the end of the section. Otherwise the initial value of this argument specifies the size of the view in bytes and is rounded up to the next host page size boundary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

InheritDisposition - A value that specifies how the view is to be shared by a child process created with a create process operation.

InheritDisposition Values

ViewShare - Inherit view and share a single copy of the committed pages with a child process using the current protection value.

ViewUnmap - Do not map the view into a child process.

AllocationType - A set of flags that describes the type of allocation that is to be performed for the specified region of pages.

AllocationType Flags

MEM_TOP_DOWN - The specified region is to be allocated from the highest portion of the address space possible based on the *ZeroBits* argument.

MEM_LARGE_PAGES - Only valid with physical memory mappings. The specified view should be mapped with the largest page size possible.

MEM_DOS_LIM - Only valid on x86, provided for DOS/VDM compatibility. Allows views to be mapped on 4kb boundaries rather than allocation granularity.

Protect - The protection desired for the region of initially committed pages.

Protect Values

PAGE_NOACCESS - No access to the committed region of pages is allowed. An attempt to read, write, or execute the committed region results in an access violation (i.e., a GP fault).

PAGE_READONLY - Read access to the committed region of pages is allowed. An attempt to write or execute the committed region results in an access violation.

PAGE_READWRITE - Read, and write access to the region of committed pages are allowed. If write access to the underlying section is allowed, then a single copy of the pages is shared. Otherwise the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_EXECUTE - Execute access to the committed region of pages is allowed. An attempt to read or write the committed region results in an access violation.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_GUARD - Protect the page with the underlying page protection, however, access to the region causes a "*guard page entered*" condition to be raised in the subject process. This value is only valid with one of the page protections except PAGE_NOACCESS.

Mapping a view of a section into the virtual address space of a subject process causes a region of the virtual address space to be reserved and, optionally, committed. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process, and the following access to the section:

Protect Value	**Section Access Required**
PAGE_NOACCESS	*SECTION_MAP_READ*
PAGE_READONLY	*SECTION_MAP_READ*
PAGE_READWRITE	*SECTION_MAP_WRITE, SECTION_MAP_READ*
PAGE_WRITECOPY	*SECTION_MAP_READ*
PAGE_EXECUTE	*SECTION_MAP_EXECUTE*
PAGE_EXECUTE_READ	*SECTION_MAP_READ, SECTION_MAP_EXECUTE*
PAGE_EXECUTE_READWRITE	*SECTION_MAP_EXECUTE, SECTION_MAP_READ, SECTION_MAP_WRITE*
PAGE_EXECUTE_WRITECOPY	*SECTION_MAP_EXECUTE, SECTION_MAP_READ*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

In addition to the section access, the specified page protection must be compatible with the *SectionPageProtection* specified when the section was created.

Desired View Protection	**Section Protection Required**
PAGE_NOACCESS	Any
PAGE_READONLY	Any except *PAGE_NOACCESS* and *PAGE_EXECUTE*
PAGE_READWRITE	*PAGE_READWRITE, PAGE_EXECUTE_READWRITE*
PAGE_WRITECOPY	Any except *PAGE_NOACCESS* and *PAGE_EXECUTE*
PAGE_EXECUTE	*PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY*
PAGE_EXECUTE_READ	*PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY*
PAGE_EXECUTE_READWRITE	*PAGE_EXECUTE_READWRITE*
PAGE_EXECUTE_WRITECOPY	*PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY*

The view size and section offset determine the region of the section that is mapped into the virtual address space of the subject process. The commit size determines how much of the view is initially committed. The committed pages, if any, start at the beginning of the view and extend upward.

Several different views of a section can be concurrently mapped into the virtual address space of a process. Likewise, several different views of a section can also be concurrently mapped into the virtual address space of several processes.

If the operating system determines the virtual address allocation for the view (i.e. *BaseAddress* is NULL) and the section is based, then the region chosen is the one that was reserved when the section was created.

If the operating system determines the virtual address allocation for the view and the section is not based, then the allocation is such that the specified number of high-order address bits are zero in the base address of the view. This capability is provided so that applications that use address bits for tag bits need not explicitly manage the virtual-address-space allocation themselves.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If the operating system determines the virtual address allocation for the view, the *ViewSize* is zero, and the complete section has been mapped before, the returned based address will be the base address where the complete section is already mapped. This allows library routines to map complete views of sections without having to determine if the section has been previously mapped.

If the operating system does not determine the virtual address allocation (i.e., *BaseAddress* is not null), then an attempt is made to map the view starting at the specified base address and extending upward. If any page within this region is already reserved or committed, then the view cannot be mapped.

Committed pages are initialized with the specified protection value which must be compatible with the granted access to the section. Reserved pages are given a protection value of no access. Any attempt to access these pages results in an access violation unless another sharer has previously committed the pages.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_NO_QUOTA* - Error, insufficient quota to create the specified section.
- *STATUS_NO_MEMORY* - Error, insufficient virtual memory to map specified view.
- *STATUS_SECTION_PROTECTION* - Error, the specified protection is not compatible with the underlying section protection.
- *STATUS_INVALID_PAGE_PROTECTION* - Error, an invalid page protection was specified.
- *STATUS_CONFLICTING_ADDRESSES* - Error, the specified address range conflicts with an existing address range.

2.4 Extend Size Of Section

An existing section which maps a data file can be extended with the **NtExtendSection** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
NTSTATUS
NtExtendSection (
    IN HANDLE SectionHandle,
    IN OUT PLARGE_INTEGER NewSectionSize
    );
```

Parameters:

SectionHandle - An open handle to a section object that maps a data file. *SECTION_EXTEND_SIZE* access to this handle is required.

NewSectionSize - A pointer to a variable that supplies the new size for the section. This variable receives the new size of the section. If the specified size is less than the current size, this variable receives the current size.

The extend section service allows a user to extend the size of a section that maps a data file. If the current size of the section is greater than the specified size, the section size is not changed and the current section size is written to the *NewSectionSize*.

If the current section size is less than the new section size, the current file allocation size is checked and if the file allocation size is greater than the specified new section size, the section is extended.

If, however, the file allocation size is less than the specified section size, an attempt is made to set the file allocation size to the specified new section size. If this succeeds, the section is extended. If this fails the section size is unchanged and the returned status indicates why the file allocation could not be increased.

2.5 Unmap View Of Section

A view of a section can be unmapped from the virtual address space of a subject process with the **NtUnmapViewOfSection** function:

```
NTSTATUS
NtUnmapViewOfSection(
    IN HANDLE ProcessHandle,
    IN PVOID BaseAddress
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A virtual address within the view which is to be unmapped.

The entire view of the section specified by the base address parameter is unmapped from the virtual address space of the specified process. The base address argument may be any virtual address within the view. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The virtual address region occupied by the view is no longer reserved and is available to map other views or private pages. If the view was also the last reference to the underlying section (i.e., no open handles exist to the section object), then all committed pages in the section are decommitted and the section is deleted.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.

2.6 Allocate Virtual Memory

A region of pages within the virtual address space of a subject process can be reserved and/or committed with the **NtAllocateVirtualMemory** function:

```
NTSTATUS
NtAllocateVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN ULONG ZeroBits,
    IN OUT PULONG RegionSize,
    IN ULONG AllocationType,
    IN ULONG Protect
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the allocated region of pages. If the initial value of this argument is not NULL and the memory is being reserved, then the region is allocated starting at the specified virtual address rounded down to the next 64K byte boundary. If the memory is already reserved and is being committed, this value is rounded down to a host-page-size boundary. If the initial value of this argument is NULL, then the operating system determines where to allocate the region.

ZeroBits - The number of high-order address bits that must be zero in the base address of the section view. The value of this argument must be less than 21 and is only used when the operating system determines where to allocate the view (i.e., when *BaseAddress* is NULL).

RegionSize - A pointer to a variable that will receive the actual size in bytes of the allocated region of pages. The initial value of this argument specifies the size in bytes of the region and is rounded up to the next host-page-size boundary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

AllocationType - A set of flags that describes the type of allocation that is to be performed for the specified region of pages. One of *MEM_COMMIT* or *MEM_RESERVED* is required, both are acceptable (i.e., MEM_COMMIT | MEM_RESERVE).

AllocationType Flags

MEM_COMMIT - The specified region of pages is to be committed.

MEM_RESERVE - The specified region of pages is to be reserved.

MEM_TOP_DOWN - The specified region is to be allocated from the highest portion of the address space possible based on the *ZeroBits* argument.

Protect - The protection desired for the committed region of pages.

Protect Values

PAGE_NOACCESS - No access to the committed region of pages is allowed. An attempt to read, write, or execute the committed region results in an access violation (i.e., a GP fault).

PAGE_READONLY - Read access to the committed region of pages is allowed. An attempt to write or execute the committed region results in an access violation.

PAGE_READWRITE - Read and write access to the committed region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages are shared. Otherwise, the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_EXECUTE - Execute access to the committed region of pages is allowed. An attempt to read or write the committed region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute, and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PAGE_GUARD - Protect the page with the underlying page protection, however, access to the region causes a "*guard page entered*" condition to be raised in the subject process. This value is only valid with one of the page protections except PAGE_NOACCESS.

PAGE_NOCACHE - Disable the placement of committed pages into the data cache. This is only valid for pages which are not contained within a view of a section. For pages which are contained in a view of a section, the nocache attribute may be specified when the section is created, in which case it cannot be changed. This value is only valid with one of the page protections except PAGE_NOACCESS.

This function can be used to commit a region of previously reserved pages (i.e., from a mapped view or a previous call to this function), to reserve a region of private pages, or to reserve and commit a region of private pages. This function also can be used to create a sparse population of committed private or mapped pages. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process.

If the initial value of the base address parameter is NULL, then the operating system allocates a region of private pages large enough to fulfill the specified allocation request from the virtual address space of the subject process. The base address of this region is returned in the base address parameter. Private pages are given an inherit disposition of equivalent to *ViewShare*.

Process address map entries are scanned from the base address upward until the entire range of pages can be allocated or a failure occurs. If the entire range cannot be allocated, an appropriate status value is returned and no pages are mapped.

Each page in the process virtual address space is either private or mapped into a view of a section. Private pages can be in one of three states:

1. Free - Not committed or reserved, and inaccessible

2. Committed - Allocated backing storage with access controlled by a protection code

3. Reserved - Reserved, not committed, and inaccessible

Pages that are mapped into a view of a section can be in one of two states:

1. Committed - Allocated backing storage with access controlled by a protection code

2. Reserved - Reserved, not committed, and inaccessible, but can be auto-committed if an access to the page is attempted and the page has already been committed in the section mapped by the view (i.e., the page has been committed by another sharer of the section)

As each page is considered for allocation, its state and whether it is a private or mapped page is determined. Private pages are handled as follows:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Free - A private page that is free can be reserved and/or committed.

2. Committed - A private page that is already committed is left unchanged (i.e., it is still committed and its protection is not changed).

3. Reserved - A private page that is reserved can be committed. An attempt to reserve a page already in the reserved state has no effect.

Pages that are mapped into a view of a section are handled as follows:

1. Committed - A mapped page that is already committed cannot be changed to reserved. A shared page that is already committed is unchanged, however, in certain cases it's protection may be changed. This is due to the fact that shared pages, even though committed, may not be active in the process and hence have the original protection of the mapping. In committing the page the mapping state is not checked on a page by page basis.

2. Reserved - A mapped page that is reserved can be committed.

The protection value applied to committed pages that are contained within a mapped view of a section must be compatible with the access granted to the underlying section. Note that the underlying protection of the section does not change, only the specified pages contained in the process's view. Any protection value can be applied to committed private pages. Reserved pages are given a protection value of no access.

Pages that are backed by a paging file are committed as demand-zero pages (i.e., the first attempt to read or write the page causes a page of zeros to be created). Pages that are backed by a data file are committed such that they map pages of the data file.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.

- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.

- *STATUS_INVALID_PAGE_PROTECTION* - Error, an invalid page protection was specified.

- *STATUS_CONFLICTING_ADDRESSES* - Error, the specified address range conflicts with an existing address range.

- *STATUS_NO_ACCESS* - Error, access denied to specified object.

2.7 Free Virtual Memory

A region of pages within the virtual address space of a subject process can be decommitted and/or released with the **NtFreeVirtualMemory** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
NTSTATUS
NtFreeVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN OUT PULONG RegionSize,
    IN ULONG FreeType
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the region of pages to be freed. The initial value of this argument is the base address of the region of pages to be freed. This value is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that will receive the actual size in bytes of the freed region of pages. The initial value of this argument is rounded up to the next host-page-size boundary. If this value is zero and the *BaseAddress* is the starting address of the allocated region, the complete range of pages allocated together is freed or decommitted.

FreeType - A set of flags that describes the type of free that is to be performed for the specified region of pages. One of the following:

FreeType Flags

MEM_DECOMMIT - The specified region of pages is to be decommitted.

MEM_RELEASE - The specified region of pages is to be released.

This function can be used to decommit a region of previously committed pages (i.e. from a mapped view or from an allocation of virtual memory), to release a region of previously reserved private pages, and to decommit and release a region of previously committed private pages. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process.

Process address map entries are scanned from the base address upward until the entire range of pages can be freed or until a failure occurs. If the entire range cannot be freed, an appropriate status value is returned and no pages are freed.

As each page is considered for deallocation, its state and whether it is a private or mapped page is determined. Private pages are handled as follows:

1. Free - A private page that is free cannot be released or decommitted.

2. Committed - A private page that is committed can be released and/or decommitted.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. Reserved - A private page that is reserved can be released or decommitted. Decommitting a reserved page leaves the page in the reserved state.

Pages that are mapped into a view of a section are handled as follows:

1. Committed - A mapped page that is committed cannot be decommitted or released.

2. Reserved - A mapped page that is reserved cannot be decommitted or released.

If the desired type of free is allowed for the specified *RegionSize*, then page attributes are established as necessary in the process address map, and the current length of the freed region is updated.

If the desired type of free cannot be performed on the entire range, then an appropriate status value is returned and none of the specified region is freed.

Decommitting a private page causes the backing storage for the page to be released to the appropriate paging file and the address map entry for the corresponding page to be returned to the reserved state.

Decommitted and released pages are given a protection value of no access.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.

- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.

- *STATUS_NO_ACCESS* - Error, access denied to specified object.

- *STATUS_UNABLE_TO_FREE_VM* - Error, the specified virtual memory could not be released. This could be caused by the virtual memory being a system structure (TEB or PEB) or being part of a mapped view, or the specified size larger than the original allocation.

- *STATUS_VM_NOT_AT_BASE* - Error, the region size was specified as zero, but the starting address was not the beginning of the allocation.

- *STATUS_MEMORY_NOT_ALLOCATED* - Error, no memory as been allocated at the specified base address.

2.8 Read Virtual Memory

Data can be read from the address space of another process with the **NtReadVirtualMemory** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
NTSTATUS
NtReadVirtualMemory(
    IN HANDLE ProcessHandle,
    IN PVOID BaseAddress,
    OUT PVOID Buffer,
    IN ULONG BufferSize,
    OUT PULONG NumberOfBytesRead OPTIONAL
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - The base address in the specified process of the region of pages to be read.

Buffer - The address of a buffer that receives the contents from the specified process address space.

BufferSize - The requested number of bytes to read from the specified process.

NumberOfBytesRead - Receives the actual number of bytes transferred into the specified buffer.

This function reads data from the base address in the specified process and places the data in the specified buffer. The *NtReadVirtualMemory* function probes both the input and the output buffers before any bytes are copied. If either the *Buffer* fails a probe for write, or the *BaseAddress* fails a probe for read, the function returns an error and the *NumberOfBytesRead* parameter is returned as zero.

If the probe operations are successful, an attempt is made to copy the number of bytes specified in *BufferSize*. The *NumberOfBytesRead* parameter returns the actual number of bytes copied from the specified process into the buffer. The issuing process must have *PROCESS_VM_READ* access to the subject process.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_PARTIAL_COPY* - Warning, due to protection conflicts not all the requested bytes could be copied.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_ACCESS_VIOLATION* - Error, one of the memory regions was not completely accessible.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.9 Write Virtual Memory

Data can be written to the address space of another process with the **NtWriteVirtualMemory** function:

NTSTATUS
NtWriteVirtualMemory(
    IN HANDLE *ProcessHandle,*
    OUT PVOID *BaseAddress,*
    IN PVOID *Buffer,*
    IN ULONG *BufferSize,*
    OUT PULONG *NumberOfBytesWritten* **OPTIONAL**
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - The base address in the specified process of the region of pages to be written.

Buffer - The address of a buffer that contains the contents to be written into the specified process address space.

BufferSize - The requested number of bytes to write into the specified process.

NumberOfBytesWritten - Receives the actual number of bytes transferred into the specified address space.

This function writes data from the specified buffer in the current process to the specified base address in the specified process. The *NtWriteVirtualMemory* function probes both the input and the output buffers before any bytes are copied. If either the *Buffer* fails a probe for read, or the *BaseAddress* fails a probe for write, the function returns an error and the *NumberOfBytesRead* parameter is returned as zero.

If the probe operations are successful, An attempt is made to copy the number of bytes specified in *BufferSize*. The *NumberOfBytesWritten* parameter returns the actual number of bytes copied from the buffer to the specified process. The issuing process must have *PROCESS_VM_WRITE* access to the subject process.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_PARTIAL_COPY* - Warning, due to access violations not all the requested bytes could be copied.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- *STATUS_ACCESS_VIOLATION* - Error, one of the memory regions was not completely accessible.

2.10 Flush Virtual Memory

A region of pages within the virtual address space of a subject process can be forced to be written back into the corresponding data file (if they have been modified since they were last written) with the **NtFlushVirtualMemory** function:

NTSTATUS
NtFlushVirtualMemory(
    IN HANDLE *ProcessHandle,*
    IN OUT PVOID **BaseAddress,*
    IN OUT PULONG *RegionSize,*
    OUT PIO_STATUS_BLOCK *IoStatus*
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the region of pages to flush. The initial value of this argument is the base address of the region of pages to flush. This value is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that will receive the actual size in bytes of the flushed region of pages. The initial value of this argument is rounded up to the next host-page-size boundary. If the *RegionSize* is specified as 0, the range from the base address until the last address mapped in this view is flushed.

IoStatus - A pointer to an I/O status block that receives the I/O status from the last page written.

Process address map entries are scanned from the base address upward until the entire specified range of pages has been flushed. The actual size of the flushed region and an appropriate status value are returned. The issuing process must have *PROCESS_VM_OPERATE* access to the subject process.

As each page is considered for flushing, its state is determined. If the page is committed, mapped into a view of a section that is backed by a data file, and has been modified in memory but not yet written back into the file, then a write of the modified page is initiated. Otherwise, no operation is performed on the page.

This function can be used to ensure that a consistent state of the data within a file is maintained in the presence of various sequences of updates (e.g., forced writes of log pages, etc.).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

If an I/O error occurs while writing pages, the *RegionSize* contains the starting virtual address of the write which failed, and the *IoStatus* contains the failure status.

The following status values may be returned by the function:

- o *STATUS_NORMAL* - Normal, successful completion.
- o *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- o *STATUS_NO_ACCESS* - Error, access denied to specified object.
- o Any errors possible from an NtWriteFile service.

2.11 Lock Virtual Memory

A region of pages within the virtual address space of a subject process can be locked for process residency and/or system residency with the **NtLockVirtualMemory** function:

```
NTSTATUS
NtLockVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN OUT PULONG RegionSize,
    IN ULONG MapType
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the region of pages to lock. The initial value of this argument is the base address of the region of pages to lock. This value is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that receives the actual size in bytes of the locked region of pages. The initial value of this argument is rounded up to the next host-page-size boundary.

MapType - The map type flags.

MapType Flags

MAP_PROCESS - Process residency

MAP_SYSTEM - System residency

Locking a region of pages in an address map causes the residency attributes of the corresponding pages to be set such that they are not eligible for paging.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Locking a page for system residency causes the page to remain memory resident until it is explicitly unlocked. A special privilege is required in a server system to lock a page for system residency.

Locking a page for process residency causes the page to remain memory resident while the subject process is a member of the balance set (i.e., the set of processes that are actively being considered for execution).

Note that changing the protection of a locked page to *PAGE_NOACCESS* or *PAGE_GUARD* causes the page to become unlocked. In addition, locked pages are not inherited as locked, they are unlocked in the new process.

If the entire *RegionSize* cannot be locked, an appropriate status code is returned and none of the pages is locked. The issuing process must have *PROCESS_VM_OPERATE* access to the subject process.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_WAS_LOCKED* - Warning, at least one of the pages in the specified region was already locked.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_NO_QUOTA* - Error, insufficient quota to lock the specified region.

2.12 Unlock Virtual Memory

A region of pages within the virtual address space of a subject process can be unlocked from process residency and/or system residency with the **NtUnlockVirtualMemory** function:

```
NTSTATUS
NtUnlockVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN OUT PULONG RegionSize,
    IN ULONG MapType
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the region of pages to unlock. The initial value of this argument is the base address of the region of pages to unlock. This value is rounded down to the next host-page-address boundary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

RegionSize - A pointer to a variable that receives the actual size in bytes of the unlocked region of pages. The initial value of this argument is rounded up to the next host-page-size boundary.

MapType - The map type flags.

MapType Flags

MAP_PROCESS - Process residency

MAP_SYSTEM - System residency

Unlocking a region of pages causes the residency attributes of the corresponding pages to be set such that they are eligible for paging.

Unlocking a process-resident page causes the page to become pageable until it is explicitly locked.

Unlocking a system-resident page causes the page to become pageable until it is explicitly locked. A special privilege is required in a server system to unlock a system-resident page.

If the entire *RegionSize* cannot be unlocked, an appropriate status code is returned and none of the pages is unlocked. The issuing process must have *PROCESS_VM_OPERATE* access to the subject process.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.

2.13 Protect Virtual Memory

The protection on a region of committed pages within the virtual address space of the subject process can be changed with the **NtProtectVirtualMemory** function:

```
NTSTATUS
NtProtectVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN OUT PULONG RegionSize,
    IN ULONG NewProtect,
    OUT PULONG OldProtect
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

BaseAddress - A pointer to a variable that will receive the actual base address of the protected region of pages. The initial value of this argument is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that will receive the actual size in bytes of the protected region of pages. The initial value of this argument is rounded up to the next host-page-size boundary.

NewProtect - The new protection desired for the specified region of pages.

NewProtect Values

PAGE_NOACCESS - No access to the specified region of pages is allowed. An attempt to read, write, or execute the specified region results in an access violation (i.e., a GP fault).

PAGE_READONLY - Read-access to the specified region of pages is allowed. An attempt to execute or write the specified region results in an access violation.

PAGE_READWRITE - Read and write access to the specified region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages are shared. Otherwise, the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write. This value may only be specified for an address ranges which is within a mapped view of a section.

PAGE_EXECUTE - Execute access to the specified region of pages is allowed. An attempt to read or write the specified region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute, and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write. This value may only be specified for an address ranges which is within a mapped view of a section.

PAGE_GUARD - Protect the page with the underlying page protection, however, access to the region causes a "*guard page entered*" condition to be raised in the subject process. This value is only valid with one of the page protections except PAGE_NOACCESS.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PAGE_NOCACHE - Disable the placement of committed pages into the data cache. This value is only valid when specified in combination with one of the above underlying page protections with the exception of PAGE_NOACCESS, e.g., (*PAGE_NOCACHE | PAGE_READWRITE*). The *PAGE_NOCACHE* attribute may not be specified on an address range which is within a mapped view of a section.

OldProtect - A pointer to a variable that will receive the old protection of the first page within the specified region of pages.

Setting the protection on a range of pages causes the old protection value to be replaced by a new protection value. The protection value can only be set on committed pages. The issuing process must have *PROCESS_VM_OPERATE* access to the subject process.

Note that setting page protections to *PAGE_NOACCESS* or *PAGE_GUARD* on a page which is locked in memory or locked in the process causes the locked page to become unlocked.

Setting the protection value to *PAGE_GUARD* causes guard pages to be established. If an access to a guard page is attempted, then the protection of the accessed page to be set to its declared access, and "guard page entered" condition is raised. This capability is intended to provide automatic stack checking, but can also be used to separate other data structures where appropriate.

As each page is considered for protecting, its state is determined. If the state of the page is not committed, the page is reserved and cannot be auto-committed, or the page is contained within a mapped view of a section and the granted access to the section is incompatible with the new protection, then an appropriate status value is returned and none of the pages in the specified region is modified.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.

- *STATUS_WAS_UNLOCKED* - Warning, at least one of the pages in the specified region was unlocked due to a page protection of PAGE_NOACCESS or PAGE_GUARD.

- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.

- *STATUS_INVALID_PAGE_PROTECTION* - Error, an invalid page protection was specified.

- *STATUS_NO_ACCESS* - Error, access denied to specified object.

- *STATUS_NOT_COMMITTED* - Error, some pages within the range are not committed.

- *STATUS_IS_WRITECOPY* - Warning, the protection of the region was set to *PAGE_WRITECOPY* due to the underlying nature of the section.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.14 Query Virtual Memory

Information about a range of pages within the virtual address space of the subject process can be obtained with the **NtQueryVirtualMemory** function:

NTSTATUS
NtQueryVirtualMemory(
  IN HANDLE *ProcessHandle,*
  IN PVOID *BaseAddress,*
  IN MEMORY_INFORMATION_CLASS *MemoryInformationClass,*
  OUT PVOID *MemoryInformation,*
  IN ULONG *MemoryInformationLength,*
  OUT PULONG *ReturnLength* **OPTIONAL**
  );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - The base address of the region of pages to be queried. This value is rounded down to the next host-page-address boundary.

MemoryInformationClass - The memory information class about which to retrieve information.

MemoryInformation - A pointer to a buffer that receives the specified information. The format and content of the buffer depend on the specified information class.

MemoryInformation Format by Information Class:

MemoryBasicInformation - Data type is *PMEMORY_BASIC_INFORMATION*.

MEMORY_BASIC_INFORMATION Structure

PVOID *BaseAddress* - The base address of the region.

PVOID *AllocationBase* - The allocation base of the allocation this page is contained within.

ULONG *AllocationProtect* - The protection specified when the region was initially allocated.

ULONG *RegionSize* - The size of the region in bytes beginning at the base address in which all pages have identical attributes.

ULONG *State* - The state of the pages within the region.

State Values

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

MEM_COMMIT - The state of the pages within the region is committed.

MEM_FREE - The state of the pages within the region is free.

MEM_RESERVE - The state of the pages within the region is reserved.

If the memory state is *MEM_FREE* other the *AllocationBase, AllocationProtect, Protect* and *Type* fields in the information are undefined.

If the memory state is *MEM_RESERVE* the information in the *Protect* field is undefined.

ULONG *Protect* - The protection of the pages within the region.

Protect Values

PAGE_NOACCESS - No access to the region of pages is allowed. An attempt to read, write, or execute within the region results in an access violation (i.e., a GP fault).

PAGE_READONLY - Read-access to the region of pages is allowed. An attempt to execute or write within the region results in an access violation.

PAGE_READWRITE - Read and write access to the region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages are shared. Otherwise, the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_EXECUTE - Execute access to the region of pages is allowed. An attempt to read or write within the region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

PAGE_GUARD - Protect the page with the underlying page protection, however, access to the region causes a "*guard page entered*" condition to be raised in the subject process. This value is only valid with one of the page protections except PAGE_NOACCESS.

PAGE_NOCACHE - Disable the placement of committed pages into the data cache. This value is only valid with one of the other page protections except PAGE_NOACCESS.

ULONG *Type* - The type of pages within the region.

Type Values

MEM_PRIVATE - The pages within the region are private.

MEM_MAPPED - The pages within the region are mapped into the view of a section.

MEM_IMAGE - The pages within the region are mapped into the view of an image section.

MemoryInformationLength - Specifies the length in bytes of the memory information buffer.

ReturnLength - An optional pointer which, if specified, receives the number of bytes placed in the process information buffer.

This function provides the capability to determine the state, protection, and type of a region of pages within the virtual address space of the subject process. The issuing process must have *PROCESS_QUERY_INFORMATION* access to the subject process.

The state of the first page within the region is determined and then subsequent entries in the process address map are scanned from the base address upward until either the entire range of pages has been scanned or until a page with a nonmatching set of attributes is encountered. The region attributes, the length of the region of pages with matching attributes, and an appropriate status value are returned.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_INFO_LENGTH_MISMATCH* - Error, the specified buffer size is not large enough to hold the requested information.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o *STATUS_INVALID_INFO_CLASS* - Error, the specified information class is not valid for this service.

2.15 Query Section Information

Information about a section can be obtained with the **NtQuerySection** function:

NTSTATUS
NtQuerySection(
  IN HANDLE *SectionHandle,*
  IN SECTION_INFORMATION_CLASS *SectionInformationClass,*
  OUT PVOID *SectionInformation,*
  IN ULONG *SectionInformationLength,*
  OUT PULONG *ReturnLength* **OPTIONAL**
  );

Parameters:

SectionHandle - An open handle to a section object.

SectionInformationClass - The section information class about which to retrieve information.

SectionInformation - A pointer to a buffer that receives the specified information. The format and content of the buffer depend on the specified section class.

SectionInformation Format by Information Class:

SectionBasicInformation - Data type is *PSECTION_BASIC_INFORMATION..*

SECTION_BASIC_INFORMATION Structure

PVOID *BaseAddress* - The base virtual address of the section if the section is based.

ULONG *AllocationAttributes* - The allocation attributes flags.

AllocationAttributes Flags

SEC_BASED - The section is a based section.

SEC_FILE - The section is backed by a data file.

SEC_RESERVE - All pages of the section were initially set to the reserved state.

SEC_COMMIT - All pages of the section were initially set to the committed state.

SEC_IMAGE - The section was mapped as an executable image file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

SEC_NOCACHE - All pages of the section are to be set as non-cacheable.

LARGE_INTEGER *MaximumSize* - The maximum size of the section in bytes.

SectionImageInformation - Data type is *PSECTION_IMAGE_INFORMATION*.

SECTION_IMAGE_INFORMATION Structure

PVOID *TransferAddress* - The transfer address of the image.

ULONG *ZeroBits* - The zero bits requirement for the creation of the stack.

ULONG *MaximumStackSize* - The maximum stack size required by the image.

ULONG *CommittedStackSize* - The amount of stack space to initially commit.

ULONG *SubSystemType* - Subsystem image is linked for.

ULONG *SubSystemVersion* - Subsystem version number.

ULONG *GpValue* - The value for the global pointer register.

USHORT *ImageCharacteristics* - Image characteristics.

USHORT *DllCharacteristics* - Dll characteristics.

USHORT *Machine*- Hardware platform image was built for.

USHORT *Spare1* - unused.

ULONG *LoaderFlags* - Flags specified in image for loader usage.

SectionInformationLength - Specifies the length in bytes of the section information buffer.

ReturnLength - An optional pointer which, if specified, receives the number of bytes placed in the section information buffer.

This function provides the capability to determine the base address, size, granted access, and allocation of an opened section object. The issuing process must have *SECTION_QUERY* access to the specified section.

The following status values may be returned by the function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_INFO_LENGTH_MISMATCH* - Error, the specified buffer size is not large enough to hold the requested information.
- *STATUS_INVALID_INFO_CLASS* - Error, the specified information class is not valid for this service.
- *STATUS_SECTION_NOT_IMAGE* - Error, attempt to get image information on a section which does not map an image.

2.16 Create Paging File

An existing file can be declared as a paging file with the **NtCreatePagingFile** function:

```
NTSTATUS
NtCreatePagingFile (
    IN PSTRING PageFileName,
    IN PLARGE_INTEGER InitialSize,
    IN PLARGE_INTEGER MaximumSize,
    IN ULONG Priority
    );
```

Parameters:

PageFileName - Supplies the name of an existing file to utilize as a paging file. This file must already exist.

InitialSize - Supplies the initial size of the specified paging file in bytes. This value is rounded up to the next host size boundary and the specified paging file is extended or truncated to the initial size.

MaximumSize - Supplies the maximum number of bytes to store in the specified paging file. This value is rounded up to the next host page size. This value must be greater than or equal to the *InitialSize*.

Priority - Supplies the relative priority of the paging file with zero being the lowest priority and 0xFFFFFFFF being the highest priority. Page file space on paging files is searched for based on the priority of each paging file.

At least 8 paging files may be created. The modified page writer attempts to write pages to all specified paging file simultaneously, therefore, for maximum performance, each paging file should reside on a separate disk drive.

The following status values may be returned by the function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o *STATUS_NORMAL* - Normal, successful completion.
- o Errors resulting from attempting to open, extend, or truncate the specified file.

2.17 Flush Instruction Cache

The instruction cache for a specific process can be flushed with the **NtFlushInstructionCache** function:

NTSTATUS
NtFlushInstructionCache *(*
    IN HANDLE *ProcessHandle,*
    IN PVOID *BaseAddress* **OPTIONAL***,*
    IN ULONG *Length*
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - Optionally supplies the base address to begin the flush operation at. If not specified the whole cache is flushed.

Length - Supplies the length of the buffer to flush. Only used if BaseAddress is specified.

This routine is provided for use by system debuggers and routines which dynamically modify code segments. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process

The following status values may be returned by the function:

- o *STATUS_NORMAL* - Normal, successful completion.
- o Errors resulting from referencing the specified process handle.

2.18 Flush Write Buffer

The write buffers are flushed with the **NtFlushWriteBuffer** function:

NTSTATUS
NtFlushWriteBuffer (
    VOID
    );

This routine flushes the write buffer on the current processor. On processors without write buffers no action is taken.

The following status values may be returned by the function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- *STATUS_NORMAL* - Normal, successful completion.

2.19 Close Handle

An open handle to any object can be closed with the **NtClose** function:

NTSTATUS
NtClose(
    IN HANDLE *Handle*
    );

Parameters:

Handle - An open handle to an object.

This is a generic function and can be used to close an open handle to any object.

Closing an open handle to an object causes the reference count of the associated object to be decremented. If the resultant count is zero (i.e., there are no other references to the section), then the object is deleted. If the resultant count is one, the object has a name, and the object is temporary, then an attempt is made to delete the object by removing its name from the appropriate object directory. (Note that this operation may fail if another sharer manages to open the object before the name can be deleted, i.e., the removal of the name is conditional.)

Closing a handle to a section object causes all modified pages to be written to the associated file, if the section is backed by a data file.

After a close operation, the specified section handle is no longer valid.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Windows NT Virtual Memory Specification

Revision History:

Revision 1.3, January 4, 1989

1. Add section that describes the difference between this proposal and the proposal included in the **IBM IPFS** for **Cruiser**.

2. Drop expand stack and allow set protection to establish a guard region. Accessing a guard region causes the corresponding page to be turned into a read/write page and a guard page exception to be raised.

3. Define all API functions as returning a status value that determines the success or failure of the operation.

4. Use the words "commit" and "reserve" when referring to virtual address space allocation.

5. Add flags argument to allocate and free virtual memory which signifies whether the commitment and/or reservation of the specified region is to be changed. This allows a region of private pages to be reserved without creating any kind of memory object.

6. Correct definition of giveable and gettable sections so they are temporary and mapped at a fixed address in the virtual address space of each process.

7. Correct definition of tiled to mean that the preferred mapping of the section is within the first **512mb** of the virtual address space of a process.

8. Change give and get section to work with a virtual address rather than a section handle.

9. Add function to query a region of virtual memory.

10. Clear up confusion about protection types by defining types to be no access, execute-only, read-only, read/write, and guard region.

11. Drop section offset parameter on create section operation which allows any number of section to be backed by the same data file.

12. If no **ACL** is specified for an object, then use a process default **ACL**.

13. Change close section handle to be a generic function that closes any type of handle.

14. More clearly define what permanent objects are and how they are deleted.

Revision 2.0, February 28, 1989

1. Changed format of calls to match the **Windows NT** coding guidelines.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision 2.1, March 16, 1989

1. Changed format of calls to match the new **Windows NT** coding guidelines.
2. Added *ProcessHandle* argument create operations which operated on the address space.
3. Change *Fork* attribute to *Inherit*.
4. Removed giveable and gettable attributes and replaced them with the based attribute.
5. Eliminated **NtGetSection** and **NtGiveSection** services.
6. Changed semantics of services that change virtual memory attributes on a range of pages to either change the total specified range or fail and change none of the range. This matches OS/2 behavior.
7. Added OBJ_EXCLUSIVE and OBJ_SYSTEM_TABLE flags to handle attributes in create section.
8. Added handle attributes to OpenSection service.
9. Enhanced map view to recognize multiple mappings of the same complete section and return the base address in subsequent mappings.
10. Changed FreeVirtualMemory to not allow previously committed shared pages to be decommitted. This matches the OS/2 behavior.
11. Changed lock and unlock virtual memory to talk about system and process residency rather than system and process address maps.
12. Add zero bits parameter to **NtAllocateVirtualMemory**.
13. Add **NtReadVirtualMemory** function.
14. Add **NtWriteVirtualMemory** function.
15. Add error return values.

Revision 2.2, May 9, 1989

1. Fix typos and minor inconsistencies.

Revision 2.3, August 7, 1989

1. Add SEC_IMAGE option to **NtCreateSection**.
2. Add PAG_NOCACHE option to the protection values.

Revision 2.4, September 7, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Change names of PAG_READ, PAG_READWRITE, PAG_EXECUTE, PAG_NOACCESS, PAGE_NOCACHE, PAG_GUARD to PAGE_READ, etc.

2. Change names of PAG_COMMIT, PAG_RESERVE, PAG_RELEASE, PAG_DECOMMIT, PAG_PRIVATE, PAG_MAPPED, to MEM_COMMIT, MEM_RESERVE, etc.

3. Changed NtMapViewOfSection to have AllocationType parameter and changed the type of InheritDisposition from ULONG to SECITON_INHERIT.

4. Added MEM_TOP_DOWN to NtAllocateVirtualMemory and NtMapViewOfSection.

5. Changed BaseAddress from an IN to an IN OUT in NtFreeVirtualMemory, NtProtectVirtualMemory, NtLockVirtualMemoryh and NtUnlockVirtualMemory.

6. Added BaseAddress field to MEMORYBASICINFO type.

7. Added PAGE_WRITECOPY protection to NtMapViewOfSection, NtProtectVirtualMemory, and NtQueryVirtualMemory.

8. Added note to NtProtectVirtualMemory indicating that changing a locked page to PAGE_NOACCESS causes the page to be unlocked.

9. Added note to NtLockVirtualMemory to indicate that locked pages are not locked in a process which inherits the memory.

10. Changed SectionOffset in NtMapViewOfSection to IN OUT.

Revision 2.5, October 23, 1989

1. Add PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE and PAGE_EXECUTE_WRITECOPY.

2. Change semantic of PAGE_GUARD to be similar to PAGE_NOACCESS, but instead of an "access violation" being raised, the page protection is changed to its declared protection and a "guard page entered" exception is raised. Like PAGE_NOACCESS, guard pages unlocked locked pages.

3. Added SectionPageProtection argument to NtCreateSection.

4. Added SEC_NOCACHE attribute to NtCreateSection.

5. Made SetionOffset optional for NtMapViewOfSection and changed its allocation from host page size, to system allocation granularity (64k).

6. Changed SEC_COPY, SEC_SHARE, SEC_UNMAP to ViewCopy, ViewShare, ViewUnmap in NtMapViewOfSection.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

7. Removed PAGE_NOCACHE option from NtMapViewOfSection.
8. Added PAGE_GUARD option to NtMapViewOfSection.
9. Added PAGE_GUARD option to NtAllocateVirtualMemory.
10. Clarified PAGE_NOCACHE option in NtAllocateVirtualMemory.
11. Changed region size of zero to operate on complete range in NtFreeVirtualMemory.
12. Added AllocationBase and AllocationProtect to NtQueryVirtualMemory.
13. Added SECTIONIMAGEINFO to NtQuerySection.
14. For NtReadVirtualMemory the NumberOfBytesRead was changed to be OPTIONAL.
15. For NtWriteVirtualMemory the NumberOfBytesWritten was changed to be OPTIONAL.

Revision 2.6, December 1, 1989

1. Changed description of NtCreateSection and NtOpenSection to sue OBJECT_ATTRIBUTES and reference the Object Management Specification for details.
2. Changed Query services info structure names.
3. Removed all references to TILE.

Revision 2.7, January 5, 1990

1. Changed section access rights from READ, WRITE, and EXECUTE to SECTION_MAP_READ, SECTION_MAP_WRITE, and SECTION_MAP_EXECUTE.
2. Added SECTION_QUERY access right.
3. Described the type of access required on the section and process handles for various virtual memory services.

Revision 2.8, February 8, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Changed NtReadVirtualMemory to have OUT on the buffer argument rather than IN.

2. Changed NtWriteVirtualMemory to have OUT on the base address argument rather than IN.

3. Changed both NtReadVirtualMemory and NtWriteVirtualMemory to remove the base address rounding down to the host page size.

4. Removed PAGE_NOACCESS and PAGE_GUARD as valid page protections when creating a section.

5. PAGE_NOACCESS may not be specified in combination with PAGE_GUARD or PAGE_NOCACHE.

6. Removed STATUS_BUFFER_TOO_SMALL.

7. Added status's of STATUS_INVALID_INFO_CLASS and STATUS_INFO_LENGHT_MISMATCH to query functions.

8. Disallow the combination of Commit and Release to NtFreeVirtualMemory.

9. Add STATUS_NOT_IMAGE to query vm and create section.

10. Add section on page file quota and commitment.

11. Clarify protection rules in MapViewOfSection.

12. Don't allow protection of PAGE_WRITECOPY or PAGE_EXECUTE_WRITECOPY on address ranges not mapping a view of a section.

13. Don't allow a protection of PAGE_NOCACHE on address ranges mapping a view of a section.

Revision 2.9, March 9, 1990

1. Added the following status values to various calls: STATUS_SECTION_TOO_BIG and STATUS_CONFLICTING_ADDRESS.

2. Changed DesiredAccess to type ACCESS_MASK.

3. When SEC_IMAGE is specified in NtCreateSection only accept SEC_BASED with it.

4. Limit MaximumSize in NtCreateSection to 0xFFFEFFFF.

5. Removed ViewCopy from NtMapViewOfSection.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision 3.0, May 31, 1990

1. Added the NtExtendSection service.
2. Added SECTION_EXTEND_SIZE access.
3. In create section SEC_COMMIT is only meaningful for page file backed sections.
4. Changed SectionSize parameter to type PLARGE_INTEGER in NtCreateSection.
5. Changed description of MaximumSize parameter in NtCreateSection.
6. Changed SectionOffset parameter to type PLARGE_INTEGER in NtMapViewOfSection.
7. Added NtCreatePagingFile routine.
8. Added NtFlushInstructionCache routine.
9. Added NtFlushWriteBuffer routine.
10. Changed NtFreeVirtualMemory to require the base address to be the start of the region if the region size is specified as zero.
11. Added more status codes to NtFreeVirtualMemory.

Revision 3.1, October 4, 1990

1. Added STATUS_NOT_COMMITTED to NtProtectVirtualMemory.
2. Added MEM_IMAGE as another type to NtQueryVirtualMemory.

Revision 3.2, January 24, 1991

1. Added SECTION_EXTEND_SIZE to NtOpenSection.
2. Clarified that SECTION_WRITE access also grants read access.
3. Clarified that private pages are inherited on a fork operation.
4. Changed parameters to NtCreatePagingFile.
5. Clarified NtReadVirtualMemory and NtWriteVirtualMemory to state that the buffers are probed before any bytes are copied.

Revision 3.3, April 25, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. For NtFlushVirtualMemory if RegionSize is zero, flush from the base address to the end of the mapped view.

Revision 4.0, April 28, 1993

1. Reflect Windows NT version 3.1.

2. Remove all references to OS/2.

3. Change references to 64k alignment to Allocation Granularity as this alignment is hardware architecture dependent.

4. Changed description of SEC_BASED.

5. Added MEM_LARGE_PAGES and MEM_DOS_LIM to NtMapViewOfSection.

6. Changed NtAllocateVirtualMemory to reflect the fact that committed pages may be committed.

7. Change NtFreeVirtualMemory to reflect the fact that reserved pages may be decommitted.

8. Updated section information structure.

9. Added parameters to NtFlushInstructionCache.

[end of vm.doc]

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Portable Systems Group

Windows NT Memory Management Design Note

Author: *Lou Perazzoli*

Revision 1.8, July 24, 1990

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview 1

2. Address Space Layout 1

3. Initial Page Directory at Address Space Creation 2

4. Page Frame Database (PFN) 3

5. Paged and Nonpaged Dynamic Memory 5

6. Page Fault Handling 5
   6.1 Valid PTE 6
   6.2 Transition PTE 7
   6.3 Demand Zero PTE 7
   6.4 PTE Referring to Page in Paging File 7
   6.5 PTE Referring to Prototype PTE (protection code in protoPTE) 7
   6.6 PTE Referring to Prototype PTE (protection code here) 8

7. Prototype PTEs 8
   7.1 Valid Prototype PTE 9
   7.2 Transition Prototype PTE 9
   7.3 Demand Zero Prototype PTE 9
   7.4 Prototype PTE Referring to Page in Paging File 9
   7.5 Prototype PTE Referring to Page in Mapped File 10

8. Page Protection 10

9. Retrieving a Free Page 10

10. In-Paging I/O 10
   10.1 Collided Page Faults 11

11. Sections 12
   11.1 Segments 12
   11.2 Segment Control Area 12
   11.3 Subsection Descriptors 13

12. Extending Sections 13

13. Image Activation 14
   13.1 Activation Process 14
   13.2 Create Section Operation For Images 15
   13.3 Map view of section 16
   13.4 Image Fixup 16

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

14. Virtual Address Descriptors 16

15. POSIX Fork Support 17
  15.1 Structures to Support Fork 17
  15.2 Fork Operation 17

16. Working Set Management 19

17. Physical Page Management 20
  17.1 Modified Page Writer 20
  17.2 Balance Set Manager 21

18. I/O Support 21
  18.1 Locking Pages in Memory 22
  18.2 Unlocking Pages from Memory 22
  18.3 Mapping Locked Pages into the Current Address Space 23
  18.4 Unmapping Locked Pages from the Current Address Space ... 23
  18.5 Mapping I/O Space 23
  18.6 Unmapping I/O Space 24
  18.7 Get Physical Address 24

19. File System Caching Support 24
  19.1 Mapping a View in the Cache 25
  19.2 Unmapping a View from the Cache 26
  19.3 Unlock Checked Pages 27
  19.4 Read Mapped File 27
  19.5 Purge Section 28
  19.6 Force Section Closed 28

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Overview

This specification discusses memory management issues as related to the Intel 860. The Intel 386/486 architecture has a similar PTE format and the software PTE definition is identical between the two architectures.

The memory management subsystem is responsible for the mapping of physical memory into the virtual address space of a process. The memory management functions are implemented by several distinct pieces of the executive. The *translation-not-valid fault handler* (pager) is the exception service routine that responds to page faults and makes virtual pages resident on behalf of a process. The *modified page writer* is responsible for writing modified pages to the appropriate backing store so the physical pages can be reused. The *balance set manager* is responsible for reducing process working set sizes to gain more pages of memory. Executive routines and system services are provided to allow processes some degree of control over the behavior of their memory while executing and to support various executive functions.

The memory management subsystem has the following requirements:

- A number of processes may exist in memory simultaneously, each only allowed access to its own address space
- Support for the I/O subsystem and process structure.
- A process's pages need not be totally resident at any one time.
- Virtual pages of a process are not physically contiguous.
- Processes executing the same image will share read only code and data.

The memory management subsystem imposes requirements due to the nature of page fault handling that page faults cannot occur at interrupt request levels (**IRQL**) greater than **APC_LEVEL**. This allows the pager to acquire and release mutexes in a deadlock free manner.

Because page faults can occur at **IRQL**s **0** or **APC_LEVEL**, and routines executing at **IRQL 0** can be interrupted by **APCs** at **IRQL APC_LEVEL**, all memory management function which acquire mutexes operate at **IRQL APC_LEVEL**.

2. Address Space Layout

The Intel 860 supports a 4-gigabyte virtual address space. The virtual address space is divided into 3 parts.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

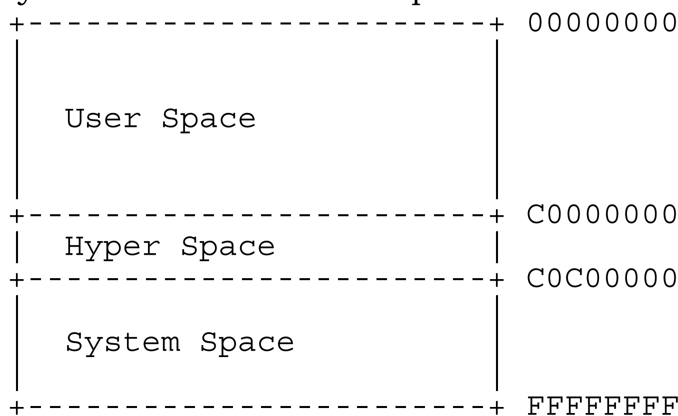
- o User Space - Consists of 3 gigabytes which is unique for each address space. The page ownership for this region is user mode.

- o Hyper Space - Consists of 12 megabytes with a page ownership of kernel mode and unique for each address space. Page table pages, working set lists, PTEs reserved for temporary mappings, and other address space unique structures reside in this region.

- o System Space - Consists of almost 1 gigabyte which is shared among all address spaces and has a page ownership of kernel mode.

The page ownership (user mode or kernel mode) is used for access checks for operations on virtual addresses.

Layout of Virtual Address Space:



System space contains a paged and a non-paged area. The paged area starts at the low addresses and grows upward, while the nonpaged area starts at the high addresses and grows downward.

The highest 32k bytes of the address space are reserved for mapping non-paged constructs used by the kernel. The Intel 860 dispatches traps to virtual address FFFFFF00, which is in the last page of the virtual address space. The page before the trap page is used to map constructs for the kernel to utilize in trap handling. This page is used for saving and restoring various registers, locating the kernel process object and locating the user's thread control block (TEB). In addition, the page is double mapped into the user portion of the address space as read-only to allow routines executing in user mode to locate the TEB.

The last 64K bytes of user space (BFFF0000 - BFFFFFFF) is set as no access. This allows argument probing within the executive to be accomplished by comparing the specified address to the highest valid address in user space (BFFEFFFF). If the address less than the highest valid address, it resides within user space, and the

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

access is allowed (note that the actual page protection may not permit the access). By setting the last 64K bytes as no access, an argument that straddles the boundary causes an access violation.

If access to the user's data is allowed, the routine requesting the access check eventually attempts to read or write the data. If the page protection prevents the access, an access violation exception is delivered and is handled by the routine attempting the access.

Any time data which resides in the user space portion of the address space is being read or written from kernel mode, an exception handler must be present to handle access violations. This is required because another thread within the process can potentially change the protection or validity of any page within user space after argument validation has occurred.

3. Initial Page Directory at Address Space Creation

When an address space is initially created, the user portion of the address space is set to no access. This is accomplished by zeroing entries 0 through 767 of the new process's PD (page directory).

Nonpaged system space is initialized by copying PD entries 992 through 1023 of the current process to the PD of the new process. This provides an identical view of a 128 megabyte nonpaged system space in every process.

Paged system space is created by building a *Virtual Address Descriptor* which describes the range of paged system space and the section which maps the paged system space. Initial references to paged system space result in page faults which are then resolved to map the corresponding page within the paged system space.

Hyper Space is created by mapping PD entry 768 to the physical page that contains the PD itself, and mapping entry 769 to a private page table page that will contain the page table entries to map the address space control structures. During the life of the process, this region can grow expand by 4 megabytes due to large working sets.

By mapping the PD to itself means that all references through PD entry 768 refer to page table pages. For example, the 32-bit word at address 0xC0000000 is the page table entry (PTE) which describes the page which maps addresses 0 through 4095. By mapping the page directory onto itself, all processes have their page table pages mapped at the same address, and one process does not have a view of another process's page table pages.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. Page Frame Database (PFN)

The PFN database contains information about each page of physical memory. The fact that this information must be available while the page is being used prevents this information from being stored in the page itself.

Every physical page of memory is in one of 5 states:

- **Active and valid**. A valid PTE refers to this page.
- **Transition**. The page is on either the modified list or the standby list. The page contents are still valid, but the page is not currently in any process's working set. A non-valid, transition PTE refers to this physical page.
- **Free**. The page is on the free list and may be used immediately.
- **Zeroed**. The page is on the zeroed list. It may be used immediately and contains all zeroes.
- **Bad**. The page is on the bad list. The page has parity or hard ECC errors which prevent it from being used.

The PFN database contains information to link all pages except active/valid pages together in lists. This allows pages to be easily manipulated to satisfy page faults.

The PFN database consists an array of records indexed by the physical page number, and has the following structure:

```
+------------------------------------+
| Forward link,  Event Address or    |
|     Working Set Index Hint         |
+------------------------------------+
| Virtual Address of (Proto) PTE     |
+------------------------------------+
| Backward link  or  Share Count     |
+------------------------------------+
| Reference Count | # valid PTEs     |
+------------------------------------+
| Original PTE contents              |
+------------------------------------+
| PFN of PTE      | flags            |
+------------------------------------+
```

When the page is on one of the lists (free, standby, modified, zeroed, or bad), the Forward and Backward link fields link the elements of the list together. Note that when a page is on one of the lists, the *ShareCount* must be zero and therefore can be overlayed with the backward link, but the *ReferenceCount* may not be zero as

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

there could be I/O in progress for this page. This is true when the page is being written to backing storage.

When a page is active/valid or transition the original contents of the PTE (which could be a prototype PTE, see section 7 for information about prototype PTEs) are stored in the PFN element. This allows the PTE to be restored when the physical page is no longer resident. In addition to the contents of the PTE, the virtual address of the PTE and the physical page number of the page which contains the PTE are stored in the PFN element. These fields provide the virtual and the physical address of the PTE which maps the page.

When a page is active/valid, the *ShareCount* field represents the number of PTEs which refer to this page. As long as the *ShareCount* is greater than zero, the page is not eligible for removal from memory.

The *ReferenceCount* field represents the number of reasons the page must be kept in memory. The *ReferenceCount* field is incremented when a page initially becomes valid (*ShareCount* becomes non-zero) and when the page is locked in memory for I/O. The *ReferenceCount* is decremented when the *ShareCount* becomes zero and when pages are unlocked from memory. When the *ReferenceCount* becomes zero, the page is placed on the free, standby or modified list depending on the contents of various flags.

The working set index hint field is only valid when the *ShareCount* is non-zero. This field indicates the index into the working set where the virtual address that maps this physical page resides. If the page is a private page, then the working set index field always contains the proper value as the page is only mapped at a single virtual address. In the case of a shared page, this hint value is only correct for the first process which made the page valid. The process which sets the hint field is guaranteed that the hint field refers to the proper index and does not need to add the Working Set List Entry referenced by the hint index into the working set tree. This reduces the size of the working set tree allowing faster searches for particular entries.

The following information is contained in the flags field:

- **Modified state** - Indicates if the page was modified requiring its contents to be saved if it is removed from memory.

- **Prototype PTE** - Indicates the PTE referenced by the PFN element is a prototype PTE.

- **In-page read in progress** - Indicates that an in-page operation is in progress for the page. In this case the event address is stored in the field used for the forward link.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o **Parity error** - Indicates the physical page contains parity or ECC errors.
- o **In-page error** - Indicates an I/O error occurred during the in-page operation on this page.

The number of valid PTEs field contains the count of PTEs within this physical page which are either valid or in transition. If this field is not zero, the page cannot be removed from memory.

5. Paged and Nonpaged Dynamic Memory

At system initialization the memory management subsystem creates two dynamic memory regions for use by the executive for storage allocation and deallocation. These storage pools are located in the system part of the address space and are mapped at the same virtual address in every process.

The *nonpaged pool* consists of a range of virtual addresses which are guaranteed to be resident in physical memory at all times and thus can be accessed from any address space without page faults. This is mapped through a single set of page table pages which are shared by each process.

The *paged pool* consists of a range of virtual addresses which may be paged in and out of a process's working set. Each process has a unique copy of page table pages which refer to the paged portion of system space. This means that there is no guarantee that an address within the paged portion of the system will not cause a page fault at any time. For this reason data structures which are operated at **IRQL** levels greater than **APC_LEVEL** must be allocated from nonpaged pool.

6. Page Fault Handling

A page fault occurs when the valid (present) bit in a PTE is zero indicating the desired page is not resident in memory. When a page fault occurs, the memory management system examines various structures to determine if the fault is a page fault or an access violation.

Upon entry, the page fault handler locates the PTE which describes the page. Note, that the page table page containing the PTE could be non-resident, or could not exist. If it does not exist, the PTE which describes the page is treated as if it were zero.

Note, that before the page is made valid, access checks are performed to ensure the requestor has access to the page.

Once the PTE is found, it can be in one of six states:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. **Active and valid** - Another thread already faulted the page into memory.
2. **Transition** - The desired page is in memory on either the standby or modified list.
3. **Demand Zero** - The desired page should be satisfied with a page of zeroes.
4. **Page File Format** - The desired page resides within a paging file and should be in-paged.
5. **Prototype PTE Format** - The desired page is potentially shared and this PTE refers to the Prototype PTE for the shared page.
6. **Unknown** - The PTE is zero. This means the *virtual address descriptors* should be examined to determine if this virtual address has been allocated.

The following figures describe the contents of a PTE for the first 5 cases. The below abbreviations are used;

Hardware usage (defined by Intel 860 reference manual):

vld - valid bit (present bit)
wrt - write bit
own - owner (user)
wt - write through cache bit
cd - cache disable bit
acc - accessed bit
dty - dirty bit
rsv - reserved, must be zero when vld is one
gp - guard page
prot. code - protection code

value	protection
0	no-access
1	read-only
2	execute-read
3	execute-only
4	read-write
5	execute-read-write
6	read-writecopy
7	execute-read-writecopy
8+(0-7)	no-cache + protection (0-7)
16+(0-15)	guard page + protection (0-15)

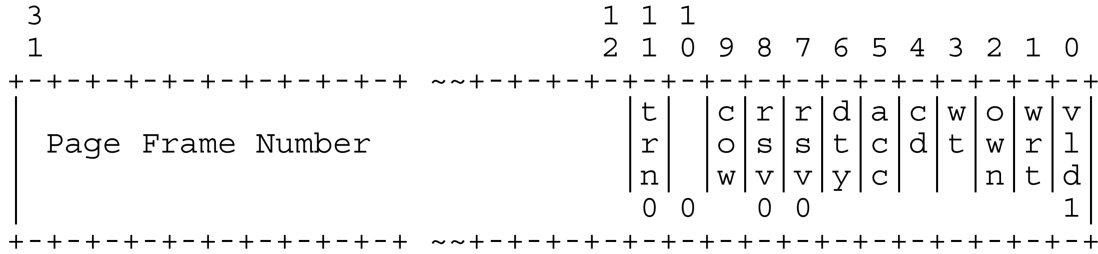
Software usage (defined by **Windows NT** memory management):

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

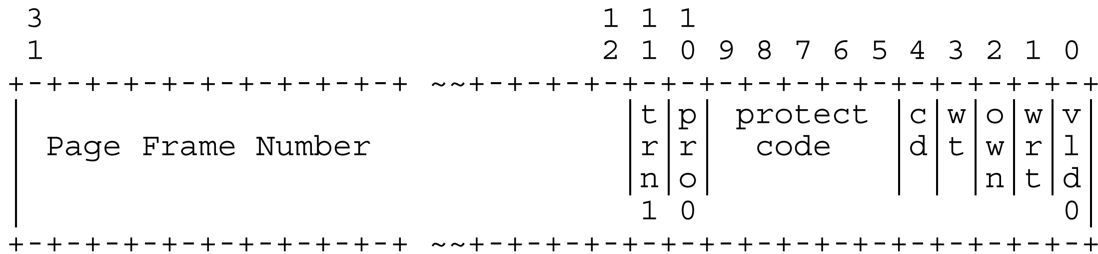
trn - if set page is in transition state (pro must be zero)
cow - if set page should be copied on write operation
pro - if set in a PTE, then the PTE refers to a prototype PTE. If set in a prototype PTE, then the prototype PTE is in mapped format.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

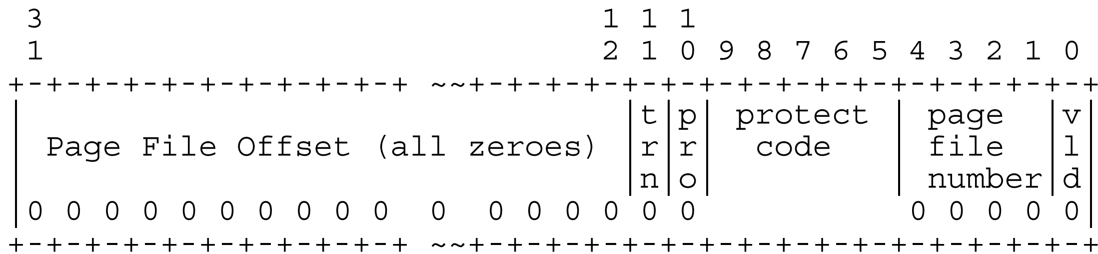
6.1 Valid PTE



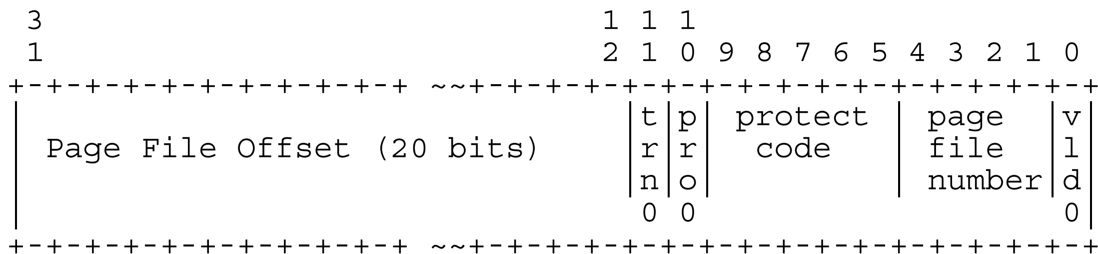
6.2 Transition PTE



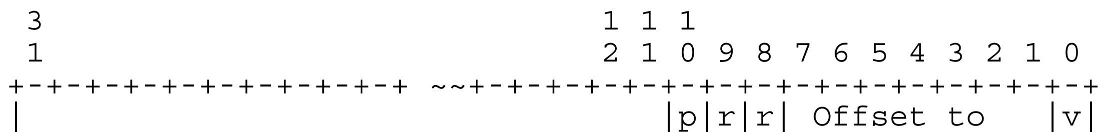
6.3 Demand Zero PTE



6.4 PTE Referring to Page in Paging File



6.5 PTE Referring to Prototype PTE (protection code in protoPTE)



Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
|  Offset to Prototype PTE          |r|s|s|  Proto PTE    |l|
|            <27:7>                 |o|v|v|    <6:0>      |d|
|                                   1 0 0                 0|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.6 PTE Referring to Prototype PTE (protection code here)

```
 3                              1 1 1
 1                              2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                 |p| protect  |       |v|
|  Offset to Prototype PTE        |r|  code    |       |l|
|            <27:7>                |o|          |       |d|
|1 1 1 1 1 1 1 1 1 1     1 1 1 1 1 1                    0|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note that a PTE in the transition state is indicated by the transition bit being set to 1 and the bit normally occupied by the prototype PTE indicator set to 0. In this case, the state of the prototype PTE indicator is maintained in the page frame database.

7. Prototype PTEs

Prototype PTEs have a similar format as normal PTEs, but are a memory management structure and never reside in a page table page. Prototype PTEs are created as a result of section creation and POSIX fork and provide the mechanism for sharing pages between multiple address spaces.

When a view of a section is mapped, the PTEs in the address space refer to the corresponding prototype PTE for the section. Hence when multiple address spaces map the same view, the same prototype PTEs are referenced allowing sharing of the physical page.

When a PTE refers to a prototype PTE, the address of the prototype PTE is calculated by multiplying the offset value by 4 and adding it to the system prototype PTE base address. Note that prototype PTEs are allocated from paged pool.

Once the prototype PTE has been located, it can be in one of 5 states:

1. **Active and valid** - Another thread already faulted the page

2. **Transition** - The desired page is in memory on one of the memory management lists (standby or modified list)

3. **Demand Zero** - The desired page should be satisfied with a page of zeroes

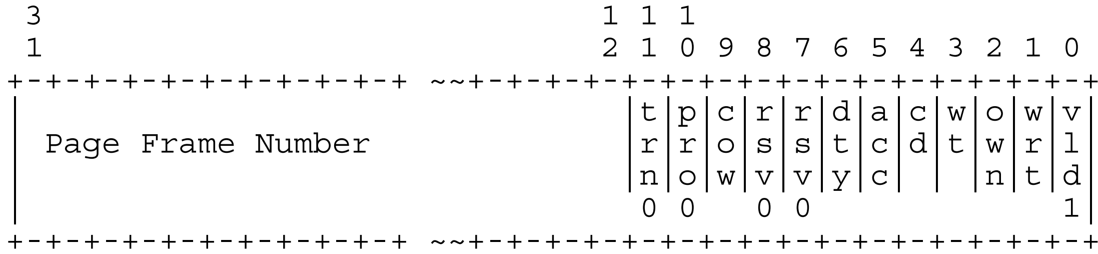
Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4. **Page File Format** - The desired page resides within a paging file and should be in-paged

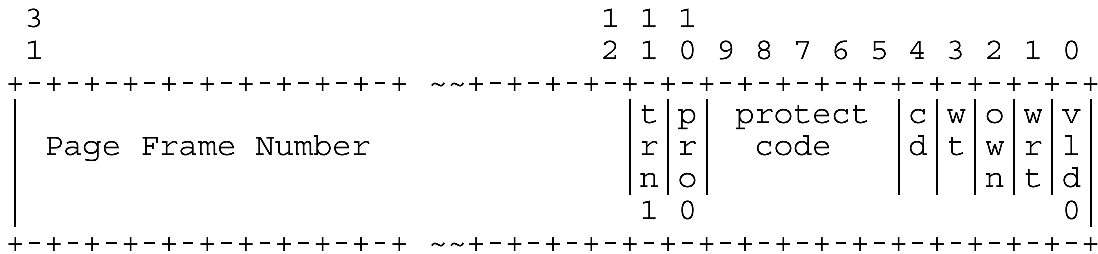
5. **Mapped File Format** - The desired page is in a mapped file and should be in-paged

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

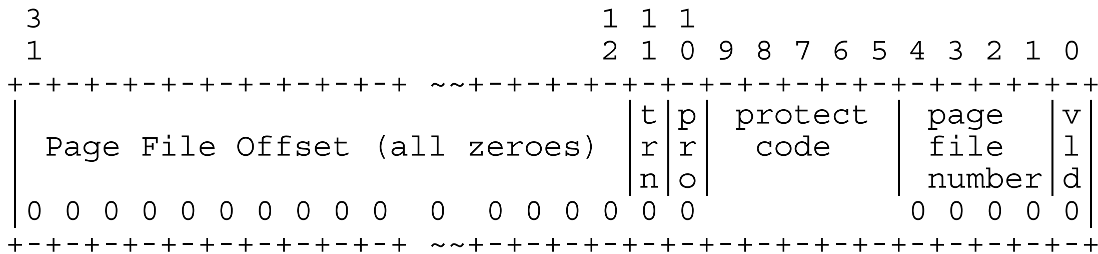
7.1 Valid Prototype PTE



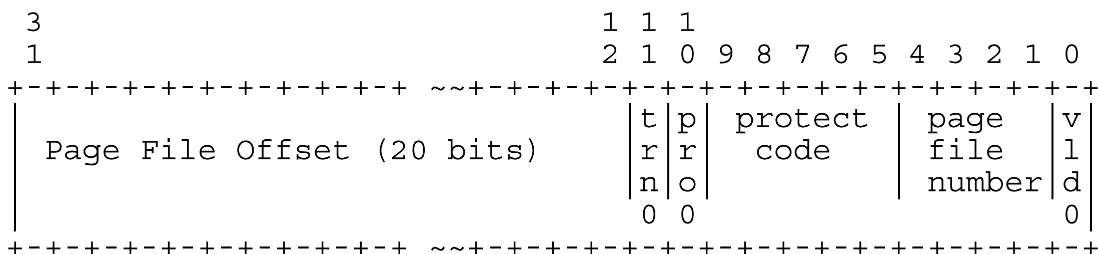
7.2 Transition Prototype PTE



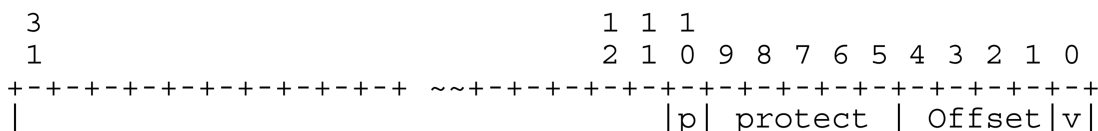
7.3 Demand Zero Prototype PTE



7.4 Prototype PTE Referring to Page in Paging File



7.5 Prototype PTE Referring to Page in Mapped File



Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

```
 | Offset to SubSection              |r|  code    | Subsct|l|
 |         &lt;24:4&gt;                     |o|          | <3:0> |d|
 |                                   1               0|
 +-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

8. Page Protection

The Intel i860 supports read only and read/write pages. Pages that are protected as either no-access or as guard pages are managed through software. This is accomplished by setting the protection mask appropriately and on the access trap, examining the protection mask and returning the appropriate status.

9. Retrieving a Free Page

The pager must supply a free page for demand zero faults, copy on write operations, and to satisfy in-page requests. For the demand zero case and in-paging less than a full page, the pager checks the zeroed page list for a page. If the zeroed page list is empty, the free list is checked and, if empty, the standby list is checked. If the standby list is empty, then no free pages are available and the thread enters a wait state for a free page.

In the case of copy on write and in-paging a full page, the free list is first checked, then the zeroed list and lastly the standby list.

If the page is to be removed from the zeroed or the free lists, the pager removes the page from the head of the list and updates the appropriate list structures. However, if the page is removed from the standby list, the PTE referenced by the corresponding PFN database element is currently in a transition state and must have its original contents restored.

This is accomplished by using the physical address information located in the PFN database element and mapping the PTE within the current process's hyper space. The transition PTE is then restored to its original state using the mapped virtual address, and the virtual address is unmapped. Note, to prevent unnecessary TB and cache flushes, a pool of PTEs for mapping addresses is maintained in hyper space and the unmapping of these virtual addresses is only done when the pool becomes exhausted. At this time all the reserved mapping PTEs are made invalid and the TB and caches are flushed permitting reuse of the PTEs.

10. In-Paging I/O

In-paging I/O occurs when a read operation must be issued to a file (paging or mapped) to complete a page fault operation. The in-page I/O operation is essentially synchronous. It is not APC interruptible, and the thread waits on an event once the I/O operation is issued.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

In order to support in-paging, each thread control block contains two events and two I/O status blocks which are used for I/O requests. There are two events and IOSBs to allow the initial in-page I/O to incur a page fault, which could issue another I/O request. The second I/O request would use the other event/IOSB pair. This allows file retrieval pointers and other information to be kept in the pagable portion of the executive. Note, however, that information relating to a paging file must not be maintained in the pagable portion or the page fault cannot complete.

The pager uses a special modifier in the I/O request function to indicate paging I/O. Upon I/O completion, the *IoCompletion* mechanism recognizes this as paging I/O, and writes the IOSB and sets the event. It does not attempt to deliver a kernel mode APC or unlock buffers. Note that both the event and the IOSB are allocated in nonpaged pool and the event is set to false before the I/O is issued. Setting the event to false prevents a race condition where another thread issues a wait on the event before the I/O system has cleared the event.

When the event is set, the wait is satisfied, allowing the pager to continue in-page processing. The pager checks the IOSB status, updates the PFN database and other structures, and completes the page fault.

While the paging I/O operation is in progress, the faulting thread does not own any mutexes. This allows other threads within the process to incur and handle page faults and issue virtual memory APIs. This exposes a number of interesting conditions which must be recognized by the pager when the I/O completes and the mutexes are again acquired. These conditions are:

- Another thread could have faulted the same page.
- The page could have been deleted (and remapped) from the virtual address space.
- Another process could have faulted the same page.
- The protection on the page could have changed.
- The fault could have been for a prototype PTE and the page which maps the prototype PTE could be out of the working set.

The pager handles these conditions by saving enough state on the kernel stack before the paging I/O request such that when the request is complete, it can detect these conditions, and, in the pathological cases, just dismiss the page fault without making the page valid. When the faulting instruction is reissued, the pager will again be invoked, and the proper action taken.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

10.1 Collided Page Faults

The case when another thread or process faults a page which is currently being in-paged is known as a *collided page fault*. This case is detected and handled optimally by the pager as it is a common occurrence in multi-threaded systems.

When an in-page operation occurs, a physical page is allocated for the I/O request, and the corresponding PTE is placed into the transition state referring to the physical page. The PFN database element for the physical page contains the original PTE contents, the physical address of the transition PTE, and the virtual address of the event which is used in the I/O request.

If another thread or process faults the same page, the collided page fault is detected by the pager noticing that the page is in transition and the read-in-progress flag in the PFN database element is set. In this case the pager issues a wait operation on the event address specified in the PFN database element.

When the I/O operation completes, all threads waiting on the event have their wait satisfied. The first thread to acquire the PFN database lock, is responsible for performing the in-page completion operations. This consists of checking the IOSB (which is at a known offset from the event) to ensure the I/O operation completed successfully, clearing the read-in-progress bit in the PFN database, and updating the PTE element.

If the IOSB indicates an in-page I/O failure, the PFN database element for the page has the in-page error flag set. The corresponding PTE element is not updated and an in-page error exception is raised in the faulting thread.

When subsequent threads acquire the PFN database lock to complete the collided page fault, the pager recognizes that the initial updating has been performed as the read-in-progress bit is clear, and checks the in-page error flag in the PFN database element to ensure the in-page completed successfully. If the in-page error flag is set, the PTE is not updated and an in-page error exception is raised in the faulting thread.

11. Sections

When a section is created, a section object is created. The section object is allocated from paged pool and contains the following information:

- Size of the section in bytes.
- Type of section (page file backed, mapped file, or image file).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Pointer to the segment structures that describes the prototype PTE and the control area structures for the section.

11.1 Segments

A segment structure is created for every section that is backed by a paging file, and for each section that maps a file that is not already described by a segment. That means that multiple sections that map the same file all refer to the same segment. A segment structure is allocated from paged pool and contains:

- o Size of the segment in pages and bytes.
- o Type of segment (page file backed, mapped file, or image file)
- o Pointer to control area for the segment
- o Prototype PTEs which describe the segment

11.2 Segment Control Area

The segment control area maintains, in nonpaged pool, the information required to perform I/O to and from mapped files. The segment control area contains:

- o A pointer to the file descriptor for the associated file
- o Number of valid/transition pages in the section
- o Total number of prototype PTEs in section
- o Pointer to section
- o One or more subsection descriptors

11.3 Subsection Descriptors

A subsection contains the necessary information to calculate the prototype PTE to logical sector number (LSN) correspondence. It contains:

- o A pointer to the segment control area
- o Address of first prototype PTE for this subsection
- o Base LSN for this subsection
- o Number of LSNs in this subsection

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Pointer to any extended subsections

When a prototype PTE refers to a subsection, the logical LSN is calculated by subtracting the address of the prototype PTE from the address of first prototype PTE for this subsection. This result is doubled and added to the base LSN for this subsection. This yields the LSN for the I/O.

The size of the I/O, which is 8 sectors for a full page, is checked against the number of LSNs within this subsection and if a full 8 sectors are not read, the remainder of the page is filled with zeroes.

12. Extending Sections

Sections which map data files may be extended. This is accomplished by comparing the current section size to the requested extension size. If the section size is greater than the requested extension size, no extension is done.

If the section size is less than the extended size, the segment which corresponds to the mapped file is examined. If its size is greater than the requested extension, the section is extended by merely adjusting the size value in the section object. If, however, the segment size is smaller than the requested size, the segment is extended.

Segment extension is accomplished by allocating enough prototype PTEs to map the requested extension rounded up to the next 4mb boundary. This allows multiple small extensions to be made without allocating any additional structures. Associated with the prototype PTEs is a new subsection which points to the segment's control area and the extended prototype PTEs. This subsection is added to the singly linked list of subsections for the segment. In addition, the segment structures are updated to indicate the large size.

Mapping a view to the extended part of the section no different than mapping a view to a non-extended section, however, when a page fault occurs for a page within the extended part of the section, the subsection list is searched until the subsection which contains the page is located. This is accomplished by calculating the PTE offset from the start of the section and then finding the subsection which contains this offset.

13. Image Activation

Image activation is a multi-step process which consists of open the image file for execute access, creating a section for the image, mapping the image into the address space, performing fixup operations on the image, "activating any DLLs", and starting the image at it's main entry point.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Image activation has the following goals:

- o All executable images are automatically shared among all users of that image.
- o A minimum number of disk I/Os are issued to load and fixup the image.
- o Image fixup operations occur in user mode in the context executing the image.
- o Code, data, and fixup information from images is brought into the address space via the pager. No special code exists to read images into memory.

13.1 Activation Process

The following steps occur to activate an image:

- o Open the image file.

  The image file is opened using the NtOpenFile or NTCreateFile service. The specified desired access is *FILE_EXECUTE* and the file sharing specifies reading with no writers.

- o Create the section.

  A section is created using the NtCreateSection service passing in the file handle for the image and an attribute indicating the file should be treated as an image rather than as a data file.

- o Create a new process.

  A new process is created specifying the section handle of the mapped image file. When the new process is created, the image file will be mapped in the user part of the address space at its specified base address. If this image cannot be mapped at the specified base address, the process creation fails.

  During the process creation phase, the system DLL (UDLL.DLL) is also mapped into the address space of the newly created process.

- o Thread startup.

  The initial thread in the process is created with a context that will cause it to start at the entry-point specified in the image header. The routine **RtlCreateUserProcess** is provided to perform the above steps.

  The executive starts all user-mode threads with a user-mode APC targeted at **LdrInitializeThunk**. This procedure exists in the UDLL.DLL mapped in all address spaces. The **LdrInitializeThunk** routine does the following:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- Determines if this is the initial thread of a "non-forked" process. If this is the not the case, then return as initialization has already been performed. Note, in the future, initialization of such things as thread local storage, or other thread specific initialization would be performed at this time.

- DLL initialization is now performed. The PEB contains the address of the image header for the initial image. Using the image header, the image's DLL table is located and for each DLL referenced by the image, the following steps occur:

  1. If the DLL is already mapped, it doesn't need to be mapped again, skips steps 2 through 4.

  2. Open the file for the DLL image.

  3. Create a section for the DLL image.

  4. Map the DLL image into the address space. If the DLL could not be mapped at its base address, fixups are performed on the DLL image.

  5. For each entry-point used by the main image, resolve references to the mapped DLL.

  6. Examine the newly loaded DLL image for DLL and resolve any external references using these same steps.

  7. If the DLL contains an initialization routine, call the specified initialization routine.

  8. Repeat until all DLL references are resolved.

- At this point all static DLL references have been resolved and **LdrInitializeThunk** returns.

o The thread context is restored and the APC is dismissed. The thread begins execution at the specified address.

13.2 Create Section Operation For Images

Once the image file has been opened, the **NtCreateSection** service is invoked to create the necessary structures to allow the image to mapped and shared between multiple address spaces.

Create section performs the following:

1. Translate the file handle to an object pointer with execute access.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2. Check to see if the section field in the file object is NULL. If the section field is not NULL, a section object for this file already exists and the section has already been created and may be shared (don't do the rest of these steps).

3. Allocate a page of memory and read in the first 4k bytes of the image header (the image header may be less than 4k bytes and the file may be less than 4k bytes, but this read will produce the correct results).

4. Analyze the image header to ensure that the file is a valid image.

5. Using the information found in the image header, create the prototype PTEs, the subsection descriptors, update the section field in the file object, and create the section object.

6. Return the appropriate status and section handle (if successful) to the caller.

13.3 Map view of section

The **NtMapViewOfSection** maps the specified section into the specified address space and returns the base address of the section. For images, the image header mapped is at the returned base address. This allows the image header to be analyzed using offsets from the base address. An attempt is made to map the image at its specified base address thereby eliminating the need for internal fixups on the image.

From the image header information for DLLs, fixups, entry point, debugger, etc. may be obtained.

13.4 Image Fixup

The image file contains the necessary fixup information to resolve internal image addresses if the image cannot be based at the specified address. The information is mapped in the *TBD* section which can be located using the image header.

Image fixup is accomplished by mapping the pages to be modified as read/write, and changing the specified addresses by the difference between the desired base address and the actual base address. Once the fixups have been performed the page protection is changed back to its previous protection.

14. Virtual Address Descriptors

Whenever a range of virtual addresses is created (committed or reserved), a virtual address descriptor is built. A virtual address descriptor contains the following information:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Virtual address range for this descriptor
2. Section information, if any
3. Attributes for range including protection and inheritance
4. Link words to form an ordered set of virtual address descriptors

As pages within the virtual address descriptor are referenced, page table entries (which are initially zero) are updated to map the appropriate page.

15. POSIX Fork Support

POSIX fork requires that a new process (the child) be created with its virtual address mappings and contents identical to the process that initiated the fork (the parent). In order to provide efficient fork operation, pages that are shared between the parent and child are shared copy-on-write.

15.1 Structures to Support Fork

There are three structures created during a fork operation:

- The *fork prototype PTEs* that describe the unique address space that is shared between the parent and the child. A fork prototype PTE is a simple structure that contains a prototype PTE and a reference count indicating the number of processes that refer to the prototype PTE. When the reference count is decremented to zero, any resources (paging file space, physical page) are released.
- The *fork header* is a system-wide structure allocated from non-paged pool that contains the size and address of the fork prototype PTEs created during the fork operation. It also contains a reference count indicating the number of processes that refer to at least one of the fork prototype PTEs. When the reference count becomes zero, all the fork prototype PTEs and the associated fork header are deleted.
- The *fork descriptor* is a process structure allocated from non-paged pool that contains a pointer to the fork header, the starting and ending address of the fork prototype PTEs, and a reference count indicating the number of PTEs that refer to fork prototype PTEs. When the reference count in the fork descriptor becomes zero, the fork descriptor is deleted and the reference count in the fork header is decremented.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

15.2 Fork Operation

When a fork operation is performed, the following steps occur:

- o A new address space is created containing a page directory page, hyper space structures and the mapping of the non-paged executive.
- o An attach process is issued with the parent process as the target.
- o The parent process's virtual address descriptors (VADs) are examined.
- o If the VAD indicates the range is inherit on fork and the VAD is not a section with the PAG_COPY attribute, do the following:
  - Allocate a new VAD from non-paged pool and initialize it as a copy of the parent's VAD.
  - Examine the page directory entry in the child process for the starting virtual address for the memory described by the VAD. If no PDE is allocated, get a free page of memory, otherwise, use the page which has previously been allocated.
  - Examine the PTEs described by the VAD and for each PTE take the following action:
    - o Valid - Examine the PFN element to assess the PTE type.
      - If it is a prototype PTE and it was not created from a previous fork operation, put the prototype PTE address into the PFN element preserving the current protection values.
      - If it is a fork prototype PTE, put the prototype PTE address into the PFN element preserving the current protection values and increment the reference count for that fork prototype PTE.
      - If it is not a prototype PTE, allocate a fork prototype PTE, put the new PTE into prototype PTE format, set the reference count for the fork prototype PTE to 2, and change the PFN element to refer to the "fork prototype PTE."
    - o Transition
      - Acquire the PFN lock and check the PTE again. If it is still in transition, this page is private and therefore, must be converted into "fork prototype PTE format". This is identical to the valid case,

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

except the parent's PTE is changed to refer to the fork prototype PTE which is placed into the transition state.

- o Demand Zero
  - Make the corresponding PTE in the child process demand zero.
- o Page File Format
  - Allocate a fork prototype PTE, set the reference count to 2, move the page file format PTE into the fork prototype PTE and put the address of the fork prototype PTE into the PTE of both the parent and the child.
- o Prototype PTE Format
  - Copy the PTE contents to the child process and check to see if this is fork prototype PTE format, if so increment the reference count for the fork prototype PTE.

- o If the VAD indicates PAG_COPY, each committed page of the section must be copied into a private page. This is accomplished by creating a VAD to describe the section as a copy-on-write section. Later, in the context of the child process, all pages in the section are read and written, causing a private copy of the page to be created.
- o Once all the VADs have been processed, detach from the parent process and attach to the child process. In the context of the child process, build the structures to manage the VADs, fork descriptors, and force copy-on-write actions to all pages in any PAG_COPY sections.

Note that at no time during the processing of the VADs to build the PTEs in the child process was a PTE created in either the valid or transition state. By avoiding this condition no updates need occur to the working set child while attached to the parent process.

16. Working Set Management

Associated with every process is a working set. Two parameters control the size of the working set. The *WorkingSetMinimum* is the minimum number of pages guaranteed to be resident or made resident when any thread within the process is executing. The other parameter is the *WorkingSetMaximum* which is the maximum number of pages which the process is allowed to have resident. The *WorkingSetMaximum* can be exceeded if there is a large number of free pages available.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Each page fault requires a page to become resident to satisfy the request. The *working set list* is used to track the current number of pages a process has resident in memory. As pages are made valid, the page is added to the working set. When the working set reaches its limit, for every page added, a page is discarded. (Note, that due to cache flushes, etc. it may be the case that a set of pages is removed from the working set when a new page is added. This allows a number of page faults to occur without each one having to invalidate TBs and data caches.) The discarded page(s) are not removed from memory immediately, but rather, are placed on the modified or standby lists. By placing pages on the lists, if a fault occurs for the page while it is still on the list, the fault can be satisfied by removing the page from the list and placing it back into the working set.

A working set is an array of working set list entries (WSLE), with each WSLE describing one resident page of the address space. Associated with the working set list are two indexes. One index is for the start of the dynamic region (all working set list entries before that index are locked in the working set), and the other is the end of the working set list.

Replacement is performed by keeping an index into the dynamic portion of the list. When a page needs to be removed from the working set, the WSLE found by this index becomes the candidate for removal. If, after closer examination, this WLSE should be removed, then it is removed and the index is incremented. If it is not a proper candidate for replacement, the index is incremented and the next WSLE becomes the candidate for replacement. After the last WSLE in the array is examined, the index is changed to reference the first entry in the dynamic portion of the working set.

A WSLE which has been selected as a candidate for removal from the working set has the access bit in its corresponding PTE examined. If the access bit is currently set, it is cleared and the WSLE is not removed. If the access bit is clear, the WSLE is removed from the working set and placed on the modified list or standby list.

In order to limit the time to find a WSLE to replace, if a certain number, say 16, WSLEs are examined for replacement and a suitable candidate cannot be found, the first WSLE examined is the one removed from the working set and the index is adjusted accordingly.

A working set list entry is created for every valid pageable PTE. The working set list entry is 64 bits in size and consists of:

1. Virtual Page Number of this element (20 bits)

2. Attributes (locked in working set, valid, etc.) (12 bits)

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

3. Link words (16 bits each) to link working set entries together into a sorted list by virtual page number. If both link words are zero, the entry is directly indexed through the PFN working set index hint field and not in the tree.

Because the link words are 16 bits, the maximum size of a working set is limited to 65,535 entries. This means the largest amount of physical memory a process can consume is 512 MB. This does not limit the virtual size of a process. In addition, when this limit becomes a factor, the WSLE will be changed to support a larger working set.

17. Physical Page Management

The memory management subsystem maintains a working set for each process so that physical memory is shared equitably among all the active processes on the system. As the demands for physical memory increase, and the free and zeroed list becomes empty, free pages are obtained by the following methods:

- Removing pages from the standby list
- Writing pages on the modified list and placing those pages on the standby list
- Reducing the size of working sets thereby placing pages on the modified and standby list
- Eliminating processes from memory by reducing the size of their working sets to zero

17.1 Modified Page Writer

The modified page writer is a system thread created during system initialization. The modified page writer is responsible for limiting the size of the modified page list by writing pages to their backing store locations when the list becomes too big.

When invoked, the modified page writer attempts to write as many pages as possible to backing store with a single I/O request. This is accomplished by examining the original PTE field of the PFN database elements for pages on the modified page list to locate pages in contiguous locations in the backing store. Once a list is created, the pages are removed from the modified list, an I/O request is issued, and, at successful completion of the I/O request, the pages are placed at the tail of the standby list.

Pages which are in the process of being written may be faulted back into memory. This is accomplished by incrementing both the *ReferenceCount* and *ShareCount* for the physical page. When the I/O completes, the modified page writer notices that the *ShareCount* is no longer zero, and does not place the page on the standby list.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

17.2 Balance Set Manager

The balance set manager is created during system initialization and is responsible for trimming process working sets when physical memory becomes over-committed.

Each process has a minimum working set which is guaranteed to be available when any thread within the process is executing. When the balance set manager is invoked, it creates free pages by reducing working sets towards their minimum size by starting with those processes which have the lowest page fault rates.

If all working sets are reduced to their minimum, and physical memory is still over-committed, the balance set manager removes processes from the system. This is accomplished by selecting a process to eliminate, and calling the kernel function *KeExcludeProcess*. Once the kernel has prevented the process from executing, the balance set manager reduces the process's working set to zero causing pages to be placed on the modified and standby lists.

Once a process has been excluded from the balance set, it becomes eligible for inclusion when either ample physical memory exists for its minimum working set, or threads within the process become computable and the priority is such that the threads should be allowed to execute. In the later case, another process(es) may be removed from the balance set to free ample physical pages.

When the system reaches the state that physical memory is so severely over-committed that processes must be removed from the balance set to allow non-resident processes to be included in the balance set, the system performance suffers greatly. This case should be avoided in all but the most extreme cases, i.e. the amount of physical memory is inadequate for the workload.

Once memory again becomes plentiful the pager allows working sets to expand above their minimum. This is accomplished by satisfying page faults without removing pages from the working set list.

18. I/O Support

The memory management subsystem provides services for operating on virtual memory to support I/O operations. The following operations are available:

- Probing pages for access
- Locking pages in memory
- Unlocking pages from memory
- Mapping locked pages into the current address space

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Unmapping locked pages from the current address space
- o Getting the physical address of a locked page
- o Mapping I/O space
- o Unmapping I/O space

These operations are available from kernel mode at **IRQL 0** or **APC_LEVEL**.

18.1 Locking Pages in Memory

A range of virtual addresses for the current process are checked for proper accessibility and locked in physical memory with the **MmProbeAndLockPages** function:

VOID
MmProbeAndLockPages (
    IN PMDL *MemoryDescriptionList,*
    IN ULONG *AccessMode,*
    IN ULONG *Operation*
    );

Parameters:

MemoryDescriptionList - A pointer to a memory description list which contains the starting virtual address to lock, the size in bytes of the region to lock, and an array of elements that are to be filled with physical page numbers.

AccessMode - Specifies the access mode with which to probe the region (*UserMode* or *KernelMode*).

Operation - Specifies the type of the I/O operation (*IoWriteAccess, IoReadAccess* or *IoModifyAccess*).

This function probes the specified range for access and locks the pages in memory by making any nonresident pages resident and incrementing the *ReferenceCount* in the PFN database for the physical page. Incrementing the *ReferenceCount* prevents the physical page from being reused.

If any pages are found with improper access protection, or the processes working set is not sufficient to lock the range in memory, an exception is raised and none of the pages are locked in memory.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

18.2 Unlocking Pages from Memory

Pages which have been locked in memory are unlocked with the **MmUnlockPages** function:

VOID
MmUnlockPages (
    IN PMDL *MemoryDescriptionList*
    );

Parameters:

MemoryDescriptionList - A pointer to an memory definition list containing the information about locked pages.

This function analyzes the *MemoryDescriptionList* and unlocks any pages which have been locked. This function is callable within any process's context.

18.3 Mapping Locked Pages into the Current Address Space

Once a page has been locked into memory, the **MmMapLockedPages** function maps the physical pages into the address space of the current process. This provides a mechanism for system processes to virtually address the physical memory within another process.

PVOID
MmMapLockedPages (
    IN PMDL *MemoryDescriptionList*
    IN KPROCESSOR_MODE *AccessMode*
    );

Parameters:

MemoryDescriptorList - Supplies a valid Memory Descriptor List which has been updated by MmProbeAndLockPages.

AccessMode - Supplies an indicator of where to map the pages; KernelMode indicates that the pages should be mapped in the system part of the address space, UserMode indicates the pages should be mapped in the user part of the address space.

Returns the base address where the pages are mapped. The base address has the same offset as the virtual address in the MDL.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This routine will raise an exception if the processor mode is USER_MODE and quota limits or VM limits are exceeded.

18.4 Unmapping Locked Pages from the Current Address Space

Pages which have been mapped into the current process's address space are unmapped with the **MmUnmapLockedPages** function:

VOID
MmUnmapLockedPages (
    IN PVOID *BaseAddress,*
    IN PMDL *MemoryDescriptionList*
    );

Parameters:

BaseAddress - The base address where the pages are mapped.

MemoryDescriptionList - A pointer to an memory definition list containing the information about locked pages.

This function unmaps the pages which were previously mapped. Once the pages have been unmapped, they may be unlocked.

If the *MemoryDescriptionList* indicates the pages are not locked an exception is raised.

18.5 Mapping I/O Space

Physical addresses residing in the processors I/O space can be mapped to virtual addresses within the nonpagable portion of the system with the **MmMapIoSpace** function:

PVOID
MmMapIoSpace (
    IN PHYSICAL_ADDRESS *PhysicalAddress*,
    IN ULONG *NumberOfBytes*
    );

Parameters:

PhysicalAddress - The physical address within I/O space to map.

NumberOfBytes - The number of bytes to map.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

This function returns the base virtual address where the requested I/O space was mapped.

18.6 Unmapping I/O Space

Physical addresses residing in the processors I/O space can be unmapped from virtual addresses in the nonpagable portion of the system with the **MmUnmapIoSpace** function:

VOID
MmUnmapIoSpace (
    IN PVOID *BaseAddress*,
    IN ULONG *NumberOfBytes*
    );

Parameters:

BaseAddress - The virtual address within system space to unmap.

NumberOfBytes - The number of bytes to unmap.

18.7 Get Physical Address

The physical address mapped by a virtual address which has been locked in memory may be obtained with the **MmGetPhysicalAddress** function:

PHYSICAL_ADDRESS
    MmGetPhysicalAddress (
    IN PVOID *BaseAddress*
    );

Parameters:

BaseAddress - Specifies the virtual address of which to provide the physical address.

This function returns the physical address of the page mapped by the specified virtual address.

19. File System Caching Support

A 256MB region of system space is reserved for file system caching support. This region has the following characteristics:

- Not accessible from user-mode.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- o Pagable, but has a system-wide working set. This means that if a page is valid in the "system cache" it is valid in any address space. Note, however, that the page could be removed from the working set of the system cache at any time unless it has been explicitly "locked" in the system cache.
- o Only views of mapped files may reside in the cache.
- o Addresses within the system cache may not be used as arguments for the BaseAddress within NT memory management services, e.g., **NtLockVirtualMemory** supplying the base address argument as an address that resides in the cache.

The following routines are provided to the file systems and server for interacting with the system cache.

19.1 Mapping a View in the Cache

A view to a section can be mapped in the system cache with the **MmMapViewInSystemCache** function:

NTSTATUS
MmMapViewInSystemCache (
    IN PVOID *SectionObject*,
    OUT PVOID **CapturedBase,*
    IN OUT PLARGE_INTEGER *SectionOffset,*
    IN OUT PULONG *CapturedViewSize,*
    IN OUT ULONG *Protect*
    );

Parameters:

SectionObject - An pointer to a section object.

CaputuredAddress - A pointer to a variable that will receive the base address of the view.

SectionOffset - Supplies a pointer to the offset from the beginning of the section to the view in bytes. This value is rounded down to the next allocation granularity size boundary.

CapturedViewSize - A pointer to a variable that will receive the actual size in bytes of the view. If the value of this argument is zero, then a view of the section will be mapped starting at the specified section offset and continuing to the end of the section. Otherwise the initial value of this argument specifies the size of the view in bytes and is rounded up to the next host page size boundary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Protect - The protection desired for the region of initially committed pages.

19.2 Unmapping a View from the Cache

NTSTATUS
NtUnmapViewInSystemCache (
    IN PVOID *BaseAddress*
    );

Parameters:
    BaseAddress - A virtual address within the view which is to be unmapped.

18.3 Check and Lock Pages

A range of valid (or transition if the virtual address resides within the system cache) virtual addresses may be locked in memory with the check and lock pages function.

ULONG
MmCheckAndLockPages
    IN PEPROCESS *Process,*
    IN PVOID *BaseAddress,*
    IN ULONG *SizeToLock*
    );

Parameters:
    Process - Supplies a pointer to the process in which these pages are mapped.

    BaseAddress - A virtual address within the system cache to begin locking.

    SizeToLock - The number of bytes to attempt to lock in the system cache.

The returned value is the number of bytes that were actually locked.

This routine checks to see if the specified pages are resident and if so increments the reference count for the page. For addresses within the system cache, the virtual address is guaranteed to be valid until the pages are unlocked (the reference count for the page becomes zero), for pages not residing in the system cache, the physical page is resident, but a page-fault could occur when referencing this address (though no I/O operation will result from this page fault).

NOTE, this routine is not to be used for general locking of user addresses - use **MmProbeAndLockPages**. Rather, this routine is intended for well file system

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

caches which maps views of files into the address range and guarantees that the mapping will not be modified (deleted or changed) while the pages are mapped.

This routine may be called at DPC_LEVEL and below. If the base address is not within the system cache and the IRQL is at DPC_LEVEL, no pages will be locked and a zero will be returned.

19.3 Unlock Checked Pages

A range of addresses locked in memory with the **MmCheckAndLockPages** function is unlocked with the **MmUnlockCheckedPages**.

VOID
MmUnlockCheckedPages
    IN PEPROCESS *Process,*
    IN PVOID *BaseAddress,*
    IN ULONG *SizeToUnlock*
    );

Parameters:

Process - Supplies a pointer to the process in which these pages are mapped.

BaseAddress - A virtual address within the system cache to begin unlocking.

SizeToUnlock - The number of bytes to attempt to lock in the system cache, this was the function value returned when the range was locked.

If the base address is within the system cache, this routine may be called at DPC_LEVEL or below. If the base address is not within the system cache, it may be called at APC_LEVEL or below.

19.4 Read Mapped File

A range of virtual memory can be made valid with the **MmReadMappedFile** function:

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VOID
MmReadMappedFile
    IN PEPROCESS *Process,*
    IN PVOID *BaseAddress,*
    IN ULONG *Size*,
    IN PIOSTATUS_BLOCK *IoStatus*
    );

Parameters:

Process - Supplies a pointer to the process in which these pages are mapped.

BaseAddress - Supplies the virtual address within the system cache to begin reading.

Size - Supplies the number of bytes to read.

IoStatus - Supplies the I/O status value from the in-page operation.

This function checks the corresponding PTEs and makes the specified range valid with a minimum number of I/O operations. Any errors which occur during the in-page sequence are returned in the *IoStatus* argument.

19.5 Purge Section

To support file truncation, pages within a section can be cleared with the **MmPurgeSection** function:

BOOLEAN
MmPurgeSection (
    IN PVOID *SectionObject*,
    IN PLARGE_INTEGER *Offset*
    );

Parameters:

SectionObject - Supplies a pointer to the section object which to purge.

Offset - Supplies the offset into the file where the purge should begin.

This function examines the prototype PTEs beginning at the specified offset. If the PTE is active and valid, a bugcheck code is issued. If the PTE is transition, the page is put on the free list and the prototype PTE is loaded with the original contents from the PFN database.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

19.6 Force Section Closed

A section can be disassociated from a file object with the **MmForceSectionClosed** function:

```
BOOLEAN
MmForceSectionClosed
    IN POBJECT_FILE FileObject
    );
```

Parameters:

FileObject - Supplies a pointer to the file object to check for a section and attempt to close and remove the section reference.

If the section cannot be closed due to outstanding references or mapped view, the value FALSE is returned and no action is taken. If the section was successfully closed, the value TRUE is returned.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History:

Original Draft 1.0, January 6, 1989

Revision 1.1, January 20, 1989

1. Fix PTE format to include prototype PTEs.
2. Add virtual address descriptors.
3. General reorganization.

Revision 1.2, March 31, 1989

1. Add I/O routines.
2. Make PPTN database 20 bytes rather than 24.

Revision 1.3, May 3, 1989

1. Make PPTN database 24 bytes rather than 20, the PTE address field cannot overlay the Flink field.
2. Change size of last reserved page of user address space to 64k bytes.
3. Change name of I/O support routines.

Revision 1.5, August 10, 1898

1. Add description of fork structures and operation.
2. Change description of section to add description of segment object.

Revision 1.6, October 25, 1989

1. Clarify modified page writer.

Revision 1.7, July 10, 1990

1. PFN mutex is now an executive spinlock.
2. Add section detailing overview of how create and map file interact with image activiation.
3. Add section on extending mapped sections.
4. Add section on system wide cache.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision 1.8, July 25, 1990

1. At working set index hint field in PFN database.

[end of vmdesign.doc]

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License