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

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)

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.

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(

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:

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.

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:

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

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:

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

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.

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.

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.

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

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:

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.

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.

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.

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.

Return Value:

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.

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:

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.

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.

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

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.

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*

)

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:

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***,**

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.

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.

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.

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.

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.

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*,

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.

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*,

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

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**

)

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.

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.

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:

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.

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

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:

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
    )
```

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*
)

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(
```

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*
)

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*
)

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(

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.

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*

)

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:

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.

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.

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(

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.

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
```

)

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.

15.6 RtlInterlockedRemoveHeadList

PLIST_ENTRY
RtlInterlockedRemoveHeadList(
IN PLIST_ENTRY *ListHead***,**
IN PKSPIN_LOCK *Lock*
)

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.

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:

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.

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**
)

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.

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.

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.

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.

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:

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*

)

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.