Chapter 10 OS Support Routines

This chapter is a technical reference for auxiliary OS support routines that may be used by a CDM or HAM. These routines are not front-ended by the NWPA so their interfaces are subject to change as the OS changes. Generally, these OS support routines provide APIs that facilitate initialization of memory blocks, movement of memory blocks, and mapping of logical addresses to absolute addresses (and vice-versa) for initializing DMA channels. These routines also include APIs essential to making BIOS calls on EISA buses. The technical reference information is listed in alphabetical order according to routine names. The following is a list of the routines referenced in this chapter:

AllocateResourceTag

Purpose: Allocates OS resource tags for specific resource types.

Architecture Type: All

Thread Context: Blocking

Requirements: Must be called only from a blocking process level.

Syntax:

```
LONG AllocateResourceTag(
  LONG nlmHandle,
  void *resourceDescString,
  LONG resourceSignature);
```

Parameters:

Inputs:

nlmHandle The module handle (**LoadHandle**) passed to the driver when its NLM load routine was called.

resourceDescString Pointer to a null-terminated text string describing the resource, with a maximum total length of 16 bytes, including the null terminator. For example:

"NDCB Driver"

resourceSignature A value used to identify a specific resource type. The signatures the driver must pass (indicates to the OS the kind of resource tag to allocate, consequently do not change the following definitions or the OS will fail the driver's request to allocate a resource tag) to identify each resource tag type requested are defined as follows:

```
#define AESProcessSignature
0x50534541
#define AllocSignature
0x54524C41
#define CacheBelow16MegMemorySignature
0x36314243
#define EventSignature
0x544E5645
#define DiskDriverSignature
0x4B534444
#define InterruptSignature
0x50544E49
#define IORegistrationSignature
0x53524F49
#define SemiPermMemorySignature
0x454D5053 *
#define TimerSignature
0x524D4954
```

Outputs:

*v3.12 only

None

Return Value: Returns a resource tag identifying specified entry type.
0 if the call failed.

Description: Acquires a tracking identifier required by certain OS calls to track system resources (and recover them from NLM or driver failure). Typically, a driver or NLM must acquire a tag for each different type of resource it wants to allocate. However, under the NWPA driver architecture, the NWPA takes care of resource tags in behalf of CDMs and HAMs. The NWPA tracks allocated resources through each module's **NPAHandle**. The one exception to this rule is registration for event notification. Usuallly, CDMs and HAMs will not need to use this routine unless they intend to use **NPA_Register_For_Event_Notification()** to be aware of system events. Then, at load-time initialization, they must use this routine to allocate a resource tag using the `EventSignature` listed under the *ResourceSignature* parameter above.

CCmpB

Purpose: Performs a block comparison of two memory areas (BYTES).

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG CCmpB (
     BYTE *address1,
     BYTE *address2,
     LONG count)
```

Parameters:

Inputs:

address1 Address of the first block of memory to be compared.

address2 Address of the second block of memory to be compared.

count Number of BYTES to be compared.

Outputs: None

Return Value: -1 if the specified number of blocks match, or
Index of the first unmatched block pair.

Description: **CCmpB()** compares two memory blocks BYTE per BYTE. It returns either a -1 to indicate that the two blocks are identical or the block-index showing the position where the blocks first differ.

CCmpD

Purpose: Performs a block comparison of two memory areas (LONGS).

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG CCmpD (
     LONG *address1,
     LONG *address2,
     LONG count)
```

Parameters:

Inputs:

address1 Address of the first block of memory to be compared.

address2 Address of the second block of memory to be compared.

count Number of LONGS to be compared.

Outputs: None

Return Value: -1 if the specified number of blocks match, or
Index of the first unmatched block pair.

Description: **CCmpD()** compares two memory blocks LONG per LONG. It returns either a -1 to indicate that the two blocks are identical or the block-index showing the position where the blocks first differ.

CFindB

Purpose: Scans an array of BYTES for a particular value.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG CFindB(
     BYTE value,
     BYTE *address,
     LONG count);
```

Parameters:

Inputs:

value Target value being searched for.

address Pointer to the starting address of the array.

count Maximum number of BYTES to scan.

Outputs: None

Return Value: -1 if the target value is not found, or
Index from *address* where the target value was found in the array.

Description: **CFindB()** scans the array of BYTES pointed at by *address* either until *value* is found or the maximum number of BYTES specified in *count* are scanned.

CFindD

Purpose: Scans an array of LONGS for a particular value.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG CFindD(
     LONG value,
     LONG *address,
     LONG count);
```

Parameters:

Inputs:

value Target value being searched for.

address Pointer to the starting address of the array.

count Maximum number of LONGS to scan.

Outputs: None

Return Value: -1 if the target value is not found, or
Index from *address* where the target value was found in the array.

Description: **CFindD()** scans the array of LONGS pointed at by *address* either until *value* is found or the maximum number of LONGS specified in *count* are scanned.

CFindW

Purpose: Scans an array of WORDS for a particular value.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG CFindW(
     WORD value,
     WORD *address,
     LONG count);
```

Parameters:

Inputs:

value Target value being searched for.

address Pointer to the starting address of the array.

count Maximum number of WORDS to scan.

Outputs: None

Return Value: -1 if the target value is not found, or
Index from *address* where the target value was found in the array.

Description: **CFindW()** scans the array of WORDS pointed at by *address* either until *value* is found or the maximum number of WORDS specified in *count* are scanned.

CMovB

Purpose: Copies BYTES from one area to another.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
VOID CMovB(
     BYTE *source
     BYTE *destination,
     LONG count);
```

Parameters:

Inputs:

source Pointer to the starting BYTE of the block being copied.

destination Pointer to the starting BYTE where the block is being copied.

count Number of BYTES to copy.

Outputs: None.

Return Value: None.

Description: **CMovB()** copies data from the source area to the destination area.

CMovD

Purpose: Copies LONGS from one area to another.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
VOID CMovD(
     LONG *source
     LONG *destination,
     LONG count);
```

Parameters:

Inputs:

source Pointer to the starting LONG of the block being copied.

destination Pointer to the starting LONG where the block is being copied.

count Number of LONGS to copy.

Outputs: None.

Return Value: None.

Description: **CMovD()** copies data from the source area to the destination area.

CMovW

Purpose: Copies WORDS from one area to another.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
VOID CMovW(
     WORD *source
     WORD *destination,
     LONG count);
```

Parameters:

Inputs:

source Pointer to the starting WORD of the block being copied.

destination Pointer to the starting WORD where the block is being copied.

count Number of WORDS to move.

Outputs: None.

Return Value: None.

Description: **CMovW()** copies data from the source area to the destination area.

CPSemaphore

Purpose: Locks real mode workspace access.

Architecture Type: Intel

Thread Context: Blocking

Requirements: None.

Syntax: `VOID CPSemaphore(LONG workSpaceSemaphore);`

Parameters:

Inputs:

workSpaceSemaphore Handle to the workspace semaphore.

Outputs: None.

Return Value: None.

Description: **CPSemaphore()** is used to lock the real mode workspacer when making a BIOS call. This routine is called with interrupts disabled (**NPA_Interrupt_Control()**), and interrupts remain disabled.

Warning:	Do not use this call to handle critical sections local to the driver.

CSetB

Purpose: Initializes an area of memory to a value.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
void CSetB(
     BYTE value
     void *address,
     LONG count);
```

Parameters:

Inputs:

value Value to which the memory area is being set.

address Pointer to the starting address of the memory area.

count Number of BYTES in the memory area to be initialized to *value*.

Outputs: None.

Return Value: None.

Description: **CSetB()** initializes the number of BYTES specified in *count* of the memory area pointed at by *Address* to the value specified in *value*.

CSetD

Purpose: Initializes an area of memory to a value.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: The storage locations in the memory area must be on LONG boundaries.

Syntax:

```
void CSetD(
     LONG value
     void *address,
     LONG count);
```

Parameters:

Inputs:

value Value to which the memory area is being set.

address Pointer to the starting address of the memory area.

count Number of LONGS in the memory area to be initialized to *value*.

Outputs: None.

Return Value: None.

Description: **CSetD()** initializes the number of LONGS specified in *count* of the memory area pointed at by *address* to the value specified in *value*.

CSetW

Purpose: Initializes an area of memory to a value.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: The storage locations in the memory area must be on LONG boundaries.

Syntax:

```
void CSetW(
     WORD value
     void *address,
     LONG count);
```

Parameters:

Inputs:

value Value to which the memory area is being set.

address Pointer to the starting address of the memory area.

count Number of WORDS in the memory area to be initialized to *value*.

Outputs: None.

Return Value: None.

Description: **CSetW()** initializes the number of WORDS specified in *count* of the memory area pointed at by *address* to the value specified in *value*.

CVSemaphore

Purpose: Unlocks real mode workspace access.

Architecture Type: Intel

Thread Context: Blocking

Requirements: None.

Syntax: `VOID CVSemaphore(LONG workSpaceSemaphore);`

Parameters:

Inputs:

workSpaceSemaphore Handle to the workspace semaphore.

Outputs: None.

Return Value: None.

Description: **CVSemaphore()** is used to clear the semaphore that was set with **CPSemaphore()**. Normally, **CVSemaphore()** is used when the driver has finished making an EISA BIOS call so that other processes can be allowed to use the workspace. **CVSemaphore()** returns with interuupts enabled.

DisableAndRetFlags

Purpose: Saves the current state of the hardware's interrupt mask and disables all interrupts.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None

Syntax: `LONG DisableAndRetFlags(void);`

Parameters: None.

Return Value: Value of the saved interrupt mask.

Description: **DisableAndRetFlags()** saves the current state of the hardware's interrupt mask and then disables all interrupts. It returns the value of the saved interrupt mask.

DoRealModeInterrupt

Purpose: Does real mode interrupt during initialization.

Architecture Type: Intel

Thread Context: Blocking

Requirements: The input parameter structure (**InputParamStruct**) must already be initialized.

Syntax:

```
LONG DoRealModeInterrupt(
  struct InputParamStruct *inputParameters,
  struct *OutputParamStruct *outputParameters);
```

Parameters:

Inputs:

inputParameters Pointer to an initialized input parameter structure as defined below:

```
struct InputParamStruct
{
  WORD IAXRegister;
  WORD IBXRegister;
  WORD ICXRegister;
  WORD IDXRegister;
  WORD IBPRegister;
  WORD ISIRegister;
  WORD IDIRegister;
  WORD IDSRegister;
  WORD IESRegister;
  WORD IIntNumber;
};
```

Outputs:

outputParameters Pointer to a filled in output parameter structure as defined below:

```
struct OutputParamStruct
{
  WORD OAXRegister;
  WORD OBXRegister;
  WORD OCXRegister;
  WORD ODXRegister;
  WORD OBPRegister;
  WORD OSIRegister;
  WORD ODIRegister;
  WORD ODSRegister;
  WORD OESRegister;
  WORD OFlags;
};
```

Return Value: 0 if successful. The zero flag is set if the interrupt vector is called.
1 if unsuccessful. The zero flag is cleared if the interrupt vector is no longer available because DOS has been removed.

Description: **DoRealModeInterrupt()** is used to perform real mode interrupts, such as BIOS and DOS interrupts. This routine can only be called at process time, and it may enable interrupts and put the calling process to sleep. EISA boards will need to use **DoRealModeInterrupt()** to perform the INT 15h BIOS call that returns the board configuration.

> **Note:** For descriptions of the input/output parameter structures and information about making real mode BIOS calls on EISA boards, refer to the EISA specification.

DoRealModeInterrupt32

Purpose: Does 32 bit real mode interrupt during initialization.

Architecture Type: Intel

Thread Context: Blocking

Requirements: The input parameter structure (**InputParamStruct32**) must already be initialized.

Syntax:

```
LONG DoRealModeInterrupt32(
  struct InputParamStruct32 *inputParameters,
  struct *OutputParamStruct32 *outputParameters);
```

Parameters:

Inputs:

inputParameters Pointer to an initialized input parameter structure as defined below:

```
struct InputParamStruct32
{
  LONG IEAXRegister;
  LONG IEBXRegister;
  LONG IECXRegister;
  LONG IEDXRegister;
  LONG IEBPRegister;
  LONG IESIRegister;
  LONG IEDIRegister;
  WORD IDSRegister;
  WORD IESRegister;
  WORD IEFSRegister;
  WORD IEGSRegister
  BYTE IIntNumber;
 BYTE IDummy32[3];
};
```

Outputs:

outputParameters

Pointer to a filled in output parameter structure as defined below:

```
struct OutputParamStruct32
{
  LONG OEAXRegister;
  LONG OEAXRegister;
  LONG OECXRegister;
  LONG OEDXRegister;
  LONG OEBPRegister;
  LONG OESIRegister;
  LONG OEDIRegister;
  WORD ODSRegister;
  WORD OESRegister;
  WORD OFSRegister;
  WORD OGSRegister;
  LONG OFlags32;
};
```

Return Value: 0 if successful. The zero flag is set if the interrupt vector is called.
1 if unsuccessful. The zero flag is cleared if the interrupt vector is no longer available because DOS has been removed.

Description: See Purpose.

EnterDebugger

Purpose: Switches to the Novell internal debugger.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: This is for code development and debug purposes only. No **EnterDebugger()** calls are allowed in shipping code.

Syntax: `VOID EnterDebugger(void);`

Parameters:

Inputs: None

Outputs: None

Return Value: None.

Description: **EnterDebugger()** stops execution of the NetWare OS and enters the internal assembly language-oriented debugger.

GetCurrentTime

Purpose: Returns the current time in clock ticks since the server was loaded.

Architecture Type: Intel

Thread Context: Non-Blocking

Requirements: None

Syntax: `LONG GetCurrentTime(void);`

Parameters:

Inputs: None.

Outputs: None.

Return Value: The number of clock ticks (1/18th second) since the server was last loaded and began execution.

Description: This call is useful to determine the current relative time in order to determine the elapsed time between events. The current time value less the value returned at the beginning of an event is the elapsed time since the event occurred in 1/18th second clock ticks. It requires more than 2,761 days (over 7.5 years) of continuous server operation before this timer will roll over.

GetHighResolutionTimer

Purpose: Returns the current time in 100 microsecond increments.

Architecture Type: Intel

Thread Context: Non-Blocking

Requirements: Do not use this call within an interrupt service routine.

Syntax: `LONG GetHighResolutionTimer(void);`

Parameters:

Inputs: None

Outputs: None

Return Value: Time in approximately 100 microseconds per count.

Description: This timer combines the Current Time with the timer register to create a return value that has a resolution of approximately 100 microseconds per count.

Note: This call will enable interrupts. Do not make this call in a code path that requires interrupts to be disabled. Do not make this call within an interrupt service routine.

GetReadAfterWriteVerifyStatus

Purpose: Returns global Read-After-Write (RAW) verify status.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax: `LONG GetReadAfterWriteVerifyStatus (void);`

Parameters: None.

Return Value: 0 if RAW verification is off.
1 if RAW verification is on.

Description: **GetReadAfterWriteVerifyStatus()** is used to determine the current RAW verification status (on or off). This is an information call only. The driver cannot change the RAW status.

GetRealModeWorkSpace

Purpose: Used in conjunction with **DoRealModeInterrupt()** to allow the HAM access to memory in real mode.

Architecture Type: Intel

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
void GetRealModeWorkSpace(
    LONG *workSpaceSemaphore,
    LONG *protectedModeAddress,
    WORD *realModeSegment,
    WORD *realModeOffset,
    LONG *workSpaceSize);
```

Parameters:

Inputs:

workSpaceSemaphore	Address of a local variable of type LONG.
protectedModeAddress	Address of a local variable of type LONG.
realModeSegment	Address of a local variable of type WORD.
realModeOffset	Address of a local variable of type WORD.
workSpaceSize	Address of a local variable of type LONG.

Outputs:

workSpaceSemaphore	Receives a handle to the OS semaphore structure.
protectedModeAddress	Receives a 32-bit logical address of the workspace block from the OS.
realModeSegment	Receives the real mode segment of the workspace from the OS.
realModeOffset	Receives the real mode offset into the workspace segment from the OS.
workSpaceSize	Receives the size of the workspace in BYTES from the OS.

Return Value: None.

Description: **GetRealModeWorkSpace()** is used to provide a HAM with a real mode workspace. Used in conjunction with **DoRealModeInterrupt()**, the HAM has access to memory in real mode. Be aware that the HAM must provide the storage locations for the outputs it receives during this call.

Note: Since NetWare v4.x runs in protected mode, it does not allow direct access to BIOS information. **DoRealModeInterrupt()** allows the HAM to access the BIOS and get data from it. **DoRealModeInterrupt()** turns on the system interrupts and executes in a critical section; therefore, calls to **CPSemaphore()** and **CVSemaphore()** are necessary to keep other processes out of the workspace.

GetSectorsPerCacheBuffer

Purpose: Returns the number of sectors in server cache buffers.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax: `LONG GetSectorsPerCacheBuffer(void);`

Parameters: None.

Return Value: An integer (8, 16, or 32) indicating the number of sectors in a system cache buffer.

Description: **GetSectorsPerCacheBuffer()** returns to the caller the number of sectors in a server cache buffer. This value may allow drivers that allocate buffers in SRAM to allocate the optimal buffer size, thus providing better performance.

GetSuperHighResolutionTimer

Purpose: Returns the curent time in 838 nanosecond increments.

Architecture Type: Intel

Thread Context: Non-Blocking

Requirements: None.

Syntax: `LONG GetSuperHighResolutionTimer (void);`

Parameters:

Inputs: None

Outputs: None

Return Value: Time in approximately 838 nanoseconds per count.

Description: This is a high resolution timer that combines the lowest WORD of Current Time with the timer register to give a timer resolution of approximately 838 nanoseconds per count. This call does not allow for possible tick count rollover, so the programmer must take into consideration a "negative" time count. This rollover will occur approximately every hour.

InvertLong

Purpose: Reverses the byte order of a LONG.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax: `LONG InvertLong (LONG longValue);`

Parameters:

Inputs:

longValue The LONG that is to be inverted.

Outputs: None.

Return Value: Inverted LONG. See description.

Description: **InvertLong()** takes the input LONG value and reverses (inverts) the byte order as follows:

If the input *longValue* is *WWXXYYZZ*. the Return value will be *ZZYYXXWW*.

MapDataOffsetToAbsoluteAddress

Purpose: Converts a logical memory address to an absolute address.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax: `LONG MapDataOffsetToAbsoluteAddress(LONG dataOffset);`

Parameters:

Inputs:

dataOffset Logical NetWare 32-bit memory address.

Outputs: None.

Return Value: Real 32-bit absolute hardware memory address.

Description: **MapDataOffsetToAbsoluteAddress()** converts a logical NetWare address to an absolute hardware memory address. The absolute addresss can be used to initialize DMA channels and to validate hardware options.

OutputToScreen

Purpose: This routine outputs a message to a selected screen.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: **Syntax:**

```
VOID OutputToScreen(
  LONG screenHandle,
  BYTE *controlString,
  args...);
```

Parameters:

Inputs:

screenHandle

controlString

args

Outputs:

Return Value:

Handle of the screen where the message is to be displayed.

Pointer to a null-terminated, ASCII control string similar to that used with the C sprintf() function, including embedded returns, line feeds, tabs, bells, and % specifiers (except floating point specifiers).

Arguments as indicated by *controlString*.

None.

None **Description:**

See Purpose.

SetFlags

Purpose: Sets the interrupt flags to the specified value.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax: `VOID SetFlags(LONG flags);`

Parameters:

Inputs:

flags Value of the hardware's interrupt mask upon completion of this routine.

Outputs: None.

Return Value: None.

Description: **SetFlags()** sets the interrupt mask to the value specified in *flags*. *Flags* contains a value previously obtained from a call to a critical-code function.