Chapter 6
NIOS API

Note: The following registers are preserved for "C" calls: ebx, esi, edi, and ebp.

Note: Functions that do not explicitly mention that they are callable at interrupt time are only callable from foreground context.

Global Variables

NiosMemLockFlag

```
#include <nios.h>

UINT8  NiosMemLockFlag
```

Global variable which is set to non-zero when memory that is going to be accessed at interrupt time must be locked. If set to zero, the memory need not be explicitly locked before accessing at interrupt time.

This variable determines when to make calls to the **NiosPageLock** and **NiosPageUnlock** services. Its value can change dynamically, so it is necessary to check this flag on every request.

See also **NiosPageLock** and **NiosPageUnlock**.

NiosSystemFlags

```
UINT32  NiosSystemFlags

#include <nios.h>

#define        SF_IS_80486_BIT                 0x00000001
#define        SF_MCA_BIT                      0x00000002
#define        SF_EISA_BIT                     0x00000004
#define        SF_DBCS_PRESENT_BIT             0x00000008
#define        SF_CPUID_SUPPORTED_BIT          0x00000010
               // Supported on newer 486s and Pentiums.
#define        SF_DEBUGGER_PRESENT_BIT         0x00000020
#define        SF_PCI_BIT                      0x00000040
#define        SF_LOGGING_ENABLED_BIT          0x00000080
#define        SF_PCMCIA_BIT                   0x00000100
```

CheckHardwareInterrupt

Description	Determines if the specified IRQ is requesting service.
On Entry	ecx IRQ to check Interrupts are disabled
On Exit	Z flag Set if IRQ isn't pending eax, edx Destroyed All other registers preserved Interrupts are disabled
Remarks	This function is callable at interrupt time. This service reads the Programmable Interrupt Controller (PIC) Interrupt Request Register (IRR).
See Also	

DisableHardwareInterrupt

Description	Masks the specified hardware interrupt on the appropriate programmable interrupt controller.
On Entry	ecx IRQ to disable Interrupts are disabled
On Exit	eax, edx Destroyed All other registers preserved Interrupts are disabled
Remarks	This function is callable at interrupt time.
See Also	

DoEndOfInterrupt

Description	Issues the appropriate End-of-Interrupt (EOI) for the specified IRQ level.
On Entry	ecx Interrupt number (IRQ) Interrupts are disabled
On Exit	eax Destroyed All other registers preserved Interrupts are disabled
Remarks	This function is callable at interrupt time.
See Also	

EnableHardwareInterrupt

Description	Unmasks the specified hardware interrupt on the appropriate programmable interrupt controller.
On Entry	ecx IRQ to enable Interrupts are disabled
On Exit	eax, edx Destroyed All other registers preserved Interrupts are disabled
Remarks	This function is callable at interrupt time.
See Also	

NiosAddressToHandle

Description

Returns the handle associated with a 32-bit linear address.

Syntax

```
#include <handlmgr.h>

UINT32
NiosAddressToHandle (
        UINT32      clientID,
        void        *address,
        UINT32      *handle );
```

Parameters

clientID	Unique key that allows handle manager to identify caller
address	Address for which handle is desired
handle	Handle associated with address

Returns

SUCCESS_CODE
errINVALID_CLIENT_ID
errINVALID_HANDLE
errHANDLE_NOT_FOUND

Remarks

.

See Also

NiosRegisterHandleClient
NiosDeRegisterHandleClient
NiosGetHandle
NiosFreeHandle
NiosHandletoAddress
NiosListHandles

NiosBreak

Description

Executes an interrupt 01h instruction.

Syntax

```
#include <nios.h>

void
NiosBreak (
      void);
```

Parameters

None

Returns

Nothing

Remarks

NiosBreak is an in-line function.

See Also

NiosBreak3

Description

Executes an interrupt 03h instruction.

Syntax

```
#include <nios.h>

void
NiosBreak3 (
      void);
```

Parameters

None

Returns

Nothing

Remarks

NiosBreak3 is an in-line function.

See Also

NiosCancelAESEvent

Description

Cancels a previously scheduled AES event.

Syntax

```
#include <aes.h>

UINT32
NiosCancelAESEvent(
    NiosAESECB    *aesEcb);
```

Parameters

aesEcb	Pointer to AES ECB

Returns

AES_SUCCESS	Event successfully canceled
AES_ITEM_NOT_PRESENT	AES ECB is not scheduled

Remarks

Callable at interrupt time in all environments. Interrupts remain in the same state as when called.

If successful, the AESStatus field will be set to 0.

See Also

NiosCancelAllModuleAESEvents

Description

Cancels all outstanding AES events that have been scheduled by the specified module.

Syntax

```
#include <aes.h>

void
NiosCancelAllModuleAESEvents(
    modHandle       module);
```

Parameters

module	Caller's module handle

Returns

Nothing

Remarks

Callable at interrupt time in all environments. Interrupts are left in the same state as when called.

See Also

NiosCancelForegroundEvent

Description

Attempts to cancel an event previously scheduled using **NiosScheduleForegroundEvent**.

Syntax

```
#include <nios.h>

UINT32
NiosCancelForegroundEvent(
    FEB      *eventBlock);
```

Parameters

eventBlock

Returns

0	Event was cancelled successfully
!0	Event is not currently scheduled

If successful, the FEBStatus field will be set to 0.

Remarks

Callable at interrupt time in all environments. The interrupt flag is preserved and never enabled by this function.

See Also

NiosScheduleForegroundEvent
NiosPoll

NiosCfgRead

Description

Retrieves the parameter from the configuration database associated with a given keyword and performs, optionally, a type conversion.

Syntax

```
#include <config.h>

UINT32
NiosCfgRead(
    UINT8       *SectionName,
    UINT8       *KeywordName,
    void        *ParamValueBuf,
    UINT32      ParamValueBufLen,
    UINT32      ConvFlags);
```

Parameters

SectionName Name of the section to which the keyword belongs.

KeywordName Name of the keyword.

ParamValueBuf
Pointer to a buffer in which to put the parameter value. The actual format of the returned information is dependent on conversion information specified in the *Flags* parameter.

If the CFG_CONV_NONE flag option is specified, this parameter is ignored.

ParamValueBufLen
The length of *ParamValueBuf* (i.e., the maximum size the caller has allocated for the parameter value string). If the CFG_CONV_NONE flag option is specified, this parameter is ignored.

Flags Specifies how the keyword's parameter will be converted. The converted value is stored in *ParamValueBuf* on return. If multiple parameters can be specified for one keyword, the caller must use the CFG_CONV_STRING option and convert the individual parameters itself.

Only one CFG_CONV_xxxx can be specified. The following conversion types are available:

CFG_CONV_NONE
Keyword doesn't take parameters or the parser should ignore any parameters found. ParamValueBuf is ignored.

CFG_CONV_STRING
Parameter text is copied to *ParamValueBuf* unmodified. It is up to the caller to interpret the parameter.

CFG_CONV_DEC_UINT32
Parameter text is converted to a UINT32 value. The parameter is interpreted as decimal (Base 10).

CFG_CONV_HEX_UINT32
Parameter text is converted to a UINT32 value. The parameter is interpreted as hex (Base 16).

CFG_CONV_BOOLEAN
Parameter text is converted to a TRUE (-1) or FALSE (0) UINT32 value. The parameter is considered to be true if it is *true*, *yes*, or *on*, otherwise the parameter is false.

The following flag can be or'ed in with the CFG_CONV_xxxx value:

CFG_FLAG_ANYWHERE
The keyword will be recognized anywhere in the configuration file. By default the keyword is matched only if it is found in the DriverName section.

Returns

NC_OK The specified keyword was found.

NC_PARAM_NOT_FOUND
The configuration file was found but the specified parameter name does not exist.

NC_TRUNCATED
The parameter was found, but the logical line length was greater than NC_MAX_LINE_LEN. The parameter value string has been truncated. The length of the value buffer is NC_MAX_BUF_LEN-strlen(ParamName) - 1. This length includes the null terminator.

NC_OPEN_FAILED
No configuration file could be found.

NC_ALLOC_FAILED
Configuration file was found but there was not enough memory to allocate a parse buffer.

NC_READ_FAILED
Configuration file was found but there was an error reading the file into the parse buffer.

Remarks

This routine returns the parameter associated with the *first* instance of the keyword in the *first* section. Any other keywords or sections will be ignored.

If a specific section or keyword needs to be identified (that is, other than the first), or if wild card values for either the section name or the keyword name need to be located, use **NiosCfgReadSpecific**.

See Also

NiosCfgReadSpecific

Description

Retrieves the parameter associated with a given keyword and performs, optionally, a type conversion. Indices can be supplied to indicate how many sections and keywords should be skipped to retrieve the correct keyword.

Parameters

```
#include <config.h>

UINT32
NiosCfgReadSpecific(
    UINT8      *SectionName,
    UINT32     SectionIndex,
    UINT8      *KeywordName,
    UINT32     KeywordIndex,
    void       *ParamValueBuf,
    UINT32     ParamValueBufLen,
    UINT32     ConvFlags);
```

Parameters

SectionName	Name of the section to which the keyword belongs. If the *SectionName* string is "*" then only the *SectionIndex* count is used to identify which section header it will match.
SectionIndex	The number of matching headers to skip to reach the correct one. This is a zero-based value (that is, to find the second occurrence of a *SectionName*, use a 1.)
KeywordName	Name of the keyword associated with the desired parameter value. If the *KeywordName* string is "*" then only the *KeywordIndex* is used to identify which keyword to match.
KeywordIndex	The number of matching keywords to skip to reach the correct one. This is a zero-based value (that is, to find the second occurrence of a KeywordName a 1 should be used.)
ParamValueBuf	Pointer to a buffer to contain the parameter value. The actual format of the returned information is dependent on conversion information specified in *ConvFlags*.

If the CFG_CONV_NONE flag option is specified, this parameter is ignored.

ParamValueBufLen
The length of *ParamValueBuf*, i.e., the maximum size the caller has allocated for the parameter value string.

If the CFG_CONV_NONE conversion option is specified, this parameter is ignored.

ConvFlags Flag specifying how to convert the parameter. The converted value is stored in *ParamValueBuf* on return. If multiple parameters need to be specified for one keyword, the caller must use the CFG_CONV_STRING option and convert the individual parameters itself.

Only one CFG_CONV_xxxx parameter can be specified. The following conversion types are available:

CFG_CONV_NONE
Keyword doesn't take parameters or the parser should ignore any parameters found. *ParamValueBuf* is ignored.

CFG_CONV_STRING
Parameter text is copied to *ParamValueBuf* unmodified. It is up to the caller to interpret the parameter.

CFG_CONV_DEC_UINT32
Parameter text is converted to a UINT32 value. The parameter is interpreted as decimal (Base 10).

CFG_CONV_HEX_UINT32
Parameter text is converted to a UINT32 value. The parameter is interpreted as hex (Base 16).

CFG_CONV_BOOLEAN
Parameter text is converted to a TRUE (-1) or FALSE (0) UINT32 value. The parameter is considered to be true if it is *true*, *yes*, or *on*, otherwise the parameter is false.

The following flags can be or'ed in with the CFG_CONV_xxxx value.

CFG_FLAG_ANYWHERE
The keyword will be recognized anywhere in the configuration file. By default the keyword is matched only if it is found in the DriverName section.

Returns

NC_OK The specified keyword was found.

NC_PARAM_NOT_FOUND
The configuration file was found but the specified parameter name does not exist.

NC_TRUNCATED
The keyword was found, but the logical line length was greater than NC_MAX_LINE_LEN. The parameter value string has been truncated. The length of the value buffer is NC_MAX_BUF_LEN-strlen(ParamName) - 1. This length includes the null terminator. No attempt was made to convert the parameter.

NC_OPEN_FAILED
No configuration file could be found.

NC_ALLOC_FAILED
Configuration file was found but there was not enough memory to allocate a parse buffer.

NC_READ_FAILED
Configuration file was found but there was an error reading the file into the parse buffer.

Remarks

See Also

NiosCfgWrite

Description

Writes to the configuration database at the point of the first occurrence of the Section name.

Syntax

```
#include <config.h>

UINT32
NiosCfgWrite(
   UINT8        *SectionName,
   UINT8        *KeywordName,
   void       *ParamValue,
   UINT32  ConvFlags);
```

Parameters

DriverName	ASCIIZ string name of the driver to which this parameter belongs.
KeywordName	ASCIIZ string name of the keyword.
ParamValue	Optional keyword parameter. This is appended to the keyword when written to the configuration file. For example, if the CFG_CONV_DEC_UINT32 value is specified for the Flags parameter, then this parameter is converted to its equivalent ASCII string. Note that in this case the parameter is typecast to a UINT32, in other words the parameter must be passed by value, not by address. If CFG_CONV_STRING is specified, the parameter is typecast to a (UINT8 *). The maximum length of the specified string is MAX_BUF_LEN including the NULL byte. If CFG_CONV_NONE is specified for the Flags parameter, then the ParamValue parameter will be ignored.
Flags	Specifies the format of *ParamValue*. If multiple parameters need to be specified for one keyword, the caller must use the CFG_CONV_STRING option and convert the individual parameters into a string.

Only one CFG_CONV_???? can be specified. The following conversion types are available:

CFG_CONV_NONE
No parameter. *ParamValue* is ignored.

CFG_CONV_STRING
ParamValue points to an ASCIIZ string.

CFG_CONV_DEC_UINT32
ParamValue is interpreted as a UINT32 and is output as a base-10 ASCII string.

CFG_CONV_HEX_UINT32
ParamValue is interpreted as a UINT32 and is output as a base-16 ASCII string.

CFG_CONV_BOOLEAN
ParamValue is interpreted as a UINT32. A value of zero causes the string "OFF" to be output, any other value causes "ON" to be output.

CFG_CONV_DELETE
Delete the keyword from the section header. This will return NC_PARAM_NOT_FOUND if the keyword cannot be located, or NC_OK if it is located and deleted.

Returns

NC_OK The keyword and parameter were written successfully.

NC_LINE_OVERFLOW
The logical line length was greater than NC_MAX_LINE_LEN. The parameter value string was not written to the file.

NC_OPEN_FAILED
No configuration file could be found.

NC_ALLOC_FAILED
Configuration file was found but there was not enough memory to allocate a parse buffer.

NC_READ_FAILED
Configuration file was found, but there was an error reading the file into the parse buffer.

NC_WRITE_FAILED
Configuration file was found, but there was an error writing the new configuration file.

NC_PARAM_NOT_FOUND
The keyword specified for deletion was not found.

Remarks

See Also NiosCfgWriteSpecific

NiosCfgWriteSpecific

Description

Writes to the configuration database, allowing specific placement within the section and keyword listing.

Syntax

```
UINT32
NiosCfgWriteSpecific(
    UINT8       *SectionName,
    UINT32      SectionIndex,
    UINT8       *KeywordName,
    UINT32      KeywordIndex,
    void        *ParamValue,
    UINT32      ConvFlags);
```

Parameters

SectionName	ASCIIZ string name of the driver to which this parameter belongs.
SectionIndex	The number of matching headers to skip until the correct one is located. This is a zero-based value (that is, to find the second occurrence of a *SectionName*, use a 1 for the *SectionIndex*.)
KeywordName	Name of the keyword.
KeywordIndex	The number of matching keywords to skip until the correct one is located. This is a zero-based value (to find the second occurrence of a *KeywordName*, use a 1 here.)
ParamValue	Optional keyword parameter. This is appended to the keyword when written to the configuration file. For example, if the CFG_CONV_DEC_UINT32 value is specified for the *ConvFlags* parameter, then this parameter is converted to its equivalent ASCII string. Note that in this case the parameter is typecast to a UINT32. If CFG_CONV_STRING is specified, the parameter is typecast to a (UINT8 *). The maximum length of the specified string is MAX_BUF_LEN including the NULL byte. If CFG_CONV_NONE is specified for the ConvFlags parameter, then the ParamValue parameter will be ignored.

ConvFlags Option which specifies the format of the *ParamValue* parameter. If multiple parameters need to be specified for one keyword, the caller must use the CFG_CONV_STRING option and convert the individual parameters into a string.

Only one CFG_CONV_???? can be specified. The following conversion types are available:

CFG_CONV_NONE
No parameter. *ParamValue* is ignored.

CFG_CONV_STRING
ParamValue points to an ASCIIz string.

CFG_CONV_DEC_UINT32
ParamValue is interpreted as a UINT32 and is output as a base-10 ASCII string.

CFG_CONV_HEX_UINT32
ParamValue is interpreted as a UINT32 and is output as a base-16 ASCII string.

CFG_CONV_BOOLEAN
ParamValue is interpreted as a UINT32. A value of zero causes the string "OFF" to be output, any other value causes "ON" to be output.

CFG_CONV_DELETE
Delete the keyword from the section header. This will return NC_PARAM_NOT_FOUND if the keyword cannot be located, or NC_OK if it is located and deleted.

Returns

NC_OK The keyword and parameter were written successfully.

NC_LINE_OVERFLOW
The logical line length was greater than NC_MAX_LINE_LEN. The parameter value string was not written to the file.

NC_OPEN_FAILED
No configuration file could be found.

NC_ALLOC_FAILED

Configuration file was found but there was not enough memory to allocate a parse buffer.

NC_READ_FAILED
Configuration file was found, but there was an error reading the file into the parse buffer.

NC_WRITE_FAILED
Configuration file was found, but there was an error writing the new configuration file.

NC_CREATE_FAILED
The configuration file could not be created.

NC_DELETE_FAILED
The original configuration file could not be deleted. The new configuration file will be on disk named with the temporary filename.

NC_RENAME_FAILED
The temporary configuration file could not be renamed as the original file name. The new configuration file will be on disk named with the temporary filename.

NC_PARAM_NOT_FOUND
The keyword specified for deletion was not found.

Remarks

See Also NiosCfgWrite

NiosCharType

Description

Returns the size of the character pointed to by *String*.

Syntax

```
#include <nstdlib.h>

UINT32
NiosCharType(
    UINT8       *String);
```

Parameters

Returns

1	Single-byte character
2	Double-byte character

Remarks

See Also

NiosCli

Description

Executes a CLI instruction which disables interrupts.

Syntax

```
#include <nios.h>

void
NiosCli(
      void);
```

Parameters

None

Returns

Nothing

Remarks

This is an in-line function.

See Also

NiosPushfdCli
NiosPushfd
NiosPopfd
NiosSti

NiosCreateModuleHandle

Description

Allows a non-NLM module to get a valid NIOS-environment module handle which can then be used in calls to NLM services which require a module handle as a parameter.

Syntax

```
#include <module.h>

modHandle
NiosCreateModuleHandle(
      struct        VersionStampInfo *version,
      UINT8         *name,
      UINT8         *description,
      UINT8         *copyright,
      UINT32         options);
```

Parameters

version Pointer to a **VersionStampInfo** structure that contains the module's version information.

name Pointer to length-preceded NULL-terminated string which contains the module's short name. This is typically the module's filename including the extension. This can be a maximum of 14 bytes in length including the preceding length byte and terminating NULL byte.

description Pointer to length-preceded NULL-terminated string which contains a description of the module.

copyright Pointer to length-preceded NULL-terminated string which contains the module's copyright notice. This parameter can be NULL if no copyright information is available.

options Reserved for future use. Must be zero.

Returns

0	No memory or invalid parameters
!0	Pointer to created module handle

Remarks

A module handle is used to identify and track a module's resource allocations.

See Also NiosDestroyModuleHandle

NiosCreateSemaphore

Description

Allocates and initializes memory for a binary semaphore.

Syntax

```
#include <nios.h>

semHandle
NiosCreateSemaphore(
    modHandle      module,
    UINT32         reserved);
```

Parameters

module	Caller's module handle
reserved	Reserved for future use; must be zero

Returns

0	Error allocating semaphore
!0	Semaphore handle

Remarks

A binary semaphore is one created with a token count of one. This is used in cases where mutual exclusion is needed.

NIOS semaphores are used to protect an NLM module from multiple threads of execution. NLMs execute in a non-preemptive environment (that is, they run to completion).

An NLM should not be concerned with a task switch to another process within a procedure unless the NLM either directly yields or indirectly yields by invoking a yielding function.

Note: Semaphore protection will not help for data structures accessible both from foreground and at interrupt time. To protect yourself from this form of reentrancy, you must disable interrupts around critical sections of code.

See Also

NiosCreateSemaphoneEx, NiosDestroySemaphore
NiosExamineSemaphore, NiosSignalSemaphore
NiosWaitSemaphore

NiosCreateSemaphoreEx

Description

Allocates and initializes memory for a general semaphore.

Syntax

```
#include <nios.h>

semHandle
NiosCreateSemaphoreEx(
    modHandle    module,
    UINT32       tokenCount );
```

Parameters

module	Caller's module handle
tokenCount	Semaphore's initial token count. A token count of zero causes the task to block the first time NiosWaitSemaphore is invoked, likewise a token count of one causes the task to block when NiosWaitSemaphore is invoked the second time, and so forth.

Returns

Zero	Error allocating semaphore
Non-zero	Semaphore handle

Remarks

NLMs execute in a non-preemptive (run to completion) environment. NLMs should not be concerned with a task switch to another process within a procedure unless it either directly yields or indirectly yields by invoking a yielding function.

Note: Semaphore protection will not help for data structures accessible both from foreground and at interrupt time. To protect yourself from this form of re-entrancy, you must disable interrupts around critical sections of code.

See Also

NiosCreateSemaphore, NiosDestroySemaphore
NiosExamineSemaphore, NiosSignalSemaphore
NiosWaitSemaphore

NiosDebugCharInNoWait

Description Tests for user input from a debugger console.

Syntax

```
#include <nstdlib.h>

UINT32
NiosDebugCharInNoWait(
    void);
```

Parameters None

Returns

0	No debugger is present
0xFF	No character is present
Other	Character value

Remarks If an input character is present, it is read and returned. If an input character is not present, this function returns a value indicating that no character is present.

This function should only be used inside a "NIOS DEBUG QUERY" event consumer handler.

See Also NiosDebugCharInWait

NiosDebugCharInWait

Description	Waits for user input from a debugging terminal.

Syntax

```
#include <nstdlib.h>

UINT32
NiosDebugCharInWait(
            void);
```

Parameters

None

Returns

0	No debugger is present
0xFF	User pressed Control-C or ESCAPE
Other	Character value

Remarks

This function should only be used inside a "NIOS DEBUG QUERY" event consumer handler.

See Also

NiosDebugCharInNoWait

NiosDebugCharOut

Description

Displays the specified character on a debugging terminal screen.

Syntax

```
#include <nstdlib.h>

UINT32
NiosDebugCharOut(
    UINT8       charToPrint);
```

Parameters

charToPrint	Character to display on debugging terminal screen

Returns

zero	No debugger is present
non-zero	Character was displayed

Remarks

See Also

NiosDebugStringOut
NiosDprintf
NiosPrintf MT_DEBUG_OUT

NiosDebugStringOut

Description

Outputs the specified ASCIIZ string to a debugging terminal screen.

Syntax

```
#include <nstdlib.h>

UINT32
NiosDebugStringOut(
      UINT8       *string);
```

Parameters

string	Pointer to an ASCIIZ string.

Returns

zero	No debugger is present
non-zero	String was displayed

Remarks

See Also

NiosDebugCharOut
NiosDprintf
NiosPrintf MT_DEBUG_OUT

NiosDeportNlmApi

Description

Deletes an anonymous reference to the specified NLM API function.

Syntax

```
#include <module.h>

void
NiosDeportNlmApi(
      void       *apiFunctionAddr);
```

Parameters

apiFunctionAddr	Address of API function returned by **NiosImportNlmApi**.

Returns

Nothing

Remarks

See Also

NiosImportNlmApi

NiosDeRegisterHandleClient

Description

Deregisters handle manager client.

Syntax

```
#include <handlmgr.h>

UINT32
NiosDeRegisterHandleClient (
    UINT32      clientID );
```

Parameters

clientID	Unique key that allows handle manager to identify caller

Returns

SUCCESS_CODE
errINVALID_CLIENT_ID
errINVALID_PARAMETER

Remarks

See Also

NiosRegisterHandleClient
NiosListHandles
NiosGetHandle
NiosFreeHandle
NiosHandletoAddress
NiosAddressToHandle

NiosDeRegisterStdOutHandler

Description

Deregisters a previously registered STDOUT handler.

Syntax

```
#include <nstdlib.h>

void
NiosDeRegisterStdOutHandler(
    modHandle     module,
    stdOutInfo  *stdOutBlock);
```

Parameters

module	Caller's module handle
stdOutBlock	Pointer to stdoutInfo structure used during the call to register the handler

Returns

0	Handler successfully deregistered
0xFFFFFFFF	Invalid stdOutInfo block

Remarks

When no handler is registered, STDOUT is displayed using a default built-in display service.

See Also

NiosRegisterStdOutHandler

NiosDestroyModuleHandle

Description

Destroys a module handle that was created using **NiosCreateModuleHandle**.

Syntax

```
#include <module.h>

UINT32
NiosDestroyModuleHandle(
    modHandle      module);
```

Parameters

module	Module handle to destroy

Returns

0	Function successful.
0xFFFFFFFF	Invalid module handle.
0xFFFFFFFE	Module did not free one or more resources; however, module handle was still destroyed.
0xFFFFFFFD	Another module in the system has refused to allow this module handle to be destroyed.

Remarks

When the module is destroyed, it is removed from the loaded module list and the memory for the module handle is deallocated.

See Also

NiosCreateModuleHandle

NiosDestroySemaphore

Description

Destroys a semaphore that was created using the **NiosCreateSemaphore** or **NiosCreateSemaphoreEx** services.

Syntax

```
#include <nios.h>

UINT32
NiosDestroySemaphore(
    modHandle       module,
    semHandle  handle);
```

Parameters

module	Caller's module handle
handle	The handle of the semaphore that is being deleted

Returns

0	Destroy was successful
!0	Invalid semaphore handle

Remarks

See Also

NiosCreateSemaphore
NiosCreateSemaphoreEx
NiosExamineSemaphore
NiosSignalSemaphore
NiosWaitSemaphore

NiosDFindNode

Description

Searches for a given node in a doubly linked queue.

Syntax

```
#include <niosq.h>

dlinkNode
*NiosDFindNode (
    void            *key,
    dlinkQueue      *queue );
```

Parameters

key	Pointer to node to locate
queue	Queue to search for node

Returns

Zero if node is found
Non-zero if node is not found

Remarks

See Also

NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDLinkLast
NiosDNext
NiosDNextNode
NiosDPreviousNode
NiosDUnlinkFirst
NiosDUnlinkLast
NiosDUnlinkNode
NiosDQueueInit

NiosDLinkFirst

Description	Inserts a node into the front of a doubly linked list.

Syntax

```
#include <niosq.h>

void
NiosDLinkFirst (
    void            *node,
    dlinkQueue      *queue );
```

Parameters

node	Pointer to the node that is to be placed in the queue
queue	Doubly linked queue into which the node is to be placed

Returns

Nothing

Remarks

The list is assumed to be null terminated at both the head and tail. The queue structure passed in points to the head and tail nodes of the linear linked list.

Queue nodes must include a forward and backward link field in sequence. The offset to the first of these two fields must be provided to the queueing routine.

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkLast
NiosDNext
NiosDNextNode
NiosDPreviousNode
NiosDUnlinkFirst
NiosDUnlinkLast
NiosDUnlinkNode
NiosDQueueInit

NiosDLinkLast

Description

Inserts a node at the end of the doubly linked queue specified.

Syntax

```
#include <niosq.h>

void
NiosDLinkLast (
      void              *node,
      dlinkQueue        *queue );
```

Parameters

node	Node to place in queue
queue	Doubly linked queue where the node is to be inserted

Returns

Nothing

Remarks

The list is assumed to be null terminated at both the head and tail. The queue structure passed points to the head and tail nodes of the linear linked list.

Queue nodes must include a forward and backward link field in sequence. The offset to the first of these two fields must be provided to the queueing routine.

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDNext
NiosDNextNode
NiosDPreviousNode
NiosDUnlinkFirst
NiosDUnlinkLast
NiosDUnlinkNode
NiosDQueueInit

NiosDLinkNext

Description

Inserts a note (*insertNode*) after the specified node (*afterNode*) in a doubly linked list.

Syntax

```
#include <niosq.h>

void
NiosDLinkNext (
     void            *insertNode,
     void            *afterNode,
     dlinkQueue      *queue );
```

Parameters

insertNode	Node to be placed in queue
afterNode	Node to place *insertNode* after
queue	Doubly linked queue that insert operation will effect

Returns

Nothing

Remarks

The list is assumed to be null terminated at both the head and tail. The queue structure passed points to the head and tail nodes of the linear linked list.

Queue nodes must include a forward and backward link field in sequence. The offset to the first of these two fields must be provided to the queueing routine.

See Also

NiosDFindNode, NiosDLinkPrevious
NiosDLinkFirst, NiosDLinkLast
NiosDNext, NiosDNextNode
NiosDPreviousNode, NiosDUnlinkFirst
NiosDUnlinkLast, NiosDUnlinkNode
NiosDQueueInit

NiosDLinkPrevious

Description

Inserts a node (*insertNode*) in front of the specified node (*beforeNode*) in a doubly linked list.

Syntax

```
#include <niosq.h>

void
NiosDLinkPrevious (
      void              *insertNode,
      void              *beforeNode,
      dlinkQueue        *queue );
```

Parameters

insertNode	Node to be placed in queue
beforeNode	Node to place *insertNode* before
queue	Doubly linked queue that insert operation will effect

Returns

Nothing

Remarks

The list is assumed to be null terminated at both the head and tail. The queue structure passed points to the head and tail nodes of the linear linked list.

Queue nodes must include a forward and backward link field in sequence. The offset to the first of these two fields must be provided to the queueing routine.

See Also

NiosDFindNode, NiosDLinkNext
NiosDLinkFirst, NiosDLinkLast
NiosDNext, NiosDNextNode
NiosDPreviousNode, NiosDUnlinkFirst
NiosDUnlinkLast, NiosDUnlinkNode
NiosDQueueInit

NiosDNext

Description

Returns the forward link for a specified node in a doubly linked list.

Syntax

```
#include <niosq.h>

dlinkNode
*NiosDNext (
      dlinkNode   *node );
```

Parameters

node	Doubly linked node

Returns

Forward link for input node
Zero if no backward link exists

Remarks

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDLinkLast
NiosDNextNode
NiosDPreviousNode
NiosDUnlinkFirst
NiosDUnlinkLast
NiosDUnlinkNode
NiosDQueueInit

NiosDNextNode

Description

Returns the forward link for the specified node in a doubly linked list. Zero is returned if no forward link exists.

Syntax

```
#include <niosq.h>

void
*NiosDNextNode (
      void        *node );
```

Parameters

node	Doubly linked node

Returns

Forward link for input node
Zero if no forward link exists

Remarks

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDLinkLast
NiosDNext
NiosDPreviousNode
NiosDUnlinkFirst
NiosDUnlinkLast
NiosDUnlinkNode
NiosDQueueInit

NiosDPrevNode

Description

Returns the backward link for a specified node in a doubly linked list.

Syntax

```
#include <niosq.h>

void
*NiosDPrevNode (
      void            *node );
```

Parameters

node	Doubly linked node

Returns

Back link for input node
Zero if no backward link exists

Remarks

Zero is returned if no previous link exists.

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDLinkLast
NiosDNext
NiosDNextNode
NiosDUnlinkFirst
NiosDUnlinkLast
NiosDUnlinkNode
NiosDQueueInit

NiosDprintf

Description

Provides a debug trace-out function. If a debugger is present, the (formatted) string is displayed on the debugger console.

Syntax

```
include <nstdlib.h>

UINT32
NiosDprintf(
    UINT8       *FormatStr,
    ...);
```

Parameters

FormatStr	Pointer to the formatted string
...	Other possible strings

Returns

Number of bytes (columns) output

0xFFFFFFFF	No debugger present
0xFFFFFFFD	Invalid format specifier
0xFFFFFFFC	User pressed ESCAPE during a display pause

Remarks

If a debugger is not present this function logs the message to the logfile unless it is interrupt time. If there is no debugger and it is interrupt time, this function has no effect.

Refer to the "Printf Usage Information" discussion under NiosPrintf for detailed information on how to use this function.

This function is callable at interrupt time. It runs with interrupts disabled and does not yield (unless a debugger is not present and logging is enabled, in which case this function runs with interrupts enabled and may yield).

See Also

NiosDprintfReset

NiosDprintfDisablePause

Description

Disables pausing while information is output to the debug terminal using the NiosDprintf function or NiosPrintf with the MT_DEBUG_OUT message type.

Syntax

```
#include <nstdlib.h>

void
NiosDprintfDisablePause(
      void);
```

Returns

Nothing

Parameters

None

Remarks

This function should be invoked when pausing is not desired.

See Also

NiosDprintfEnablePause
NiosDprintfGetPauseMode
NisDprintfReset

NiosDprintfEnablePause

Description

Enables pausing while information is output to the debug terminal using the NiosDprintf function or NiosPrintf with the MT_DEBUG_OUT message type.

Syntax

```
#include <nstdlib.h>

void
NiosDprintfEnablePause(
      void);
```

Parameters

None

Returns

Nothing

Remarks

This function should be invoked before displaying information on the debug terminal that may exceed one display page.

After displaying the information, the caller should call NiosDprintfDisablePause to disable pause mode.

See Also

NiosDprintfDisablePause
NiosDprintfGetPauseMode
NisDprintfReset

NiosDprintfGetPauseMode

Description

Returns the current pause mode setting.

Syntax

```
include <nstdlib.h>

UINT32
NiosDprintfGetPauseMode(
    void);
```

Parameters

None

Returns

0	Pause mode is disabled
!0	Pause mode is enabled

Remarks

See Also

NiosDprintfEnablePause
NiosDprintfDisablePause
NisDprintfReset

NiosDprintfReset

Description

Resets the internal line count variable to zero.

Syntax

```
#include <nstdlib.h>

void
NiosDprintfReset(
      void);
```

Parameters

None

Returns

Nothing

Remarks

The line count variable is used to determine when the output should be paused. This function should be called before displaying a group of information.

See Also

NiosDprintfEnablePause
NiosDprintfGetPauseMode
NisDprintfDisablePause

NiosDUnlinkFirst

Description

Removes a node from the front of a doubly linked list. If the queue is empty, zero is returned.

Syntax

```
#include <niosq.h>

void
*NiosDUnlinkFirst (
      dlinkQueue        *queue );
```

Parameters

queue	Doubly linked list to remove node

Returns

Removed node
Zero if queue is empty

Remarks

Queue nodes must include a forward and backward link field in sequence. The offset to the first of these two fields must be provided to the queueing routine.

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDLinkLast
NiosDNext
NiosDNextNode
NiosDPreviousNode
NiosDUnlinkLast
NiosDUnlinkNode
NiosDQueueInit

NiosDUnlinkLast

Description

Removes the last queue entry from a doubly linked queue. If the queue is empty, zero is returned.

Syntax

```
#include <niosq.h>

void
*NiosDUnlinkLast (
     dlinkQueue        *queue );
```

Parameters

queue	Doubly linked queue from which last node is to be removed

Returns

Unlinked node
Zero if queue is empty

Remarks

Queue nodes must include a forward and backward link field in sequence. The offset to the first of these two fields must be provided to the queueing routine.

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDLinkLast
NiosDNext
NiosDNextNode
NiosDPreviousNode
NiosDUnlinkFirst
NiosDUnlinkNode
NiosDQueueInit

NiosDUnlinkNode

Description

Removes specified node from a doubly linked queue.

Syntax

```
#include <niosq.h>

dlinkNode
*NiosDUnlinkNode (
    void            *node,
    dlinkQueue      *queue );
```

Parameters

node	Link to be removed
queue	Doubly linked queue that contains the link

Returns

Pointer to unlinked node
Zero if queue is empty

Remarks

Queue nodes must include a forward and backward link field in sequence. The offset to the first of these two fields must be provided to the queueing routine.

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDLinkLast
NiosDNext
NiosDNextNode
NiosDPreviousNode
NiosDUnlinkFirst
NiosDUnlinkLast
NiosDQueueInit

NiosDQueueInit

Description

Initializes a queue for a doubly linked linear list.

Syntax

```
#include <niosq.h>

void
NiosDQueueInit (
    dlinkQueue        *queue );
```

Parameters

queue	Pointer to structure used to maintain first and last pointers in queued list

Returns

Nothing

Remarks

See Also

NiosDFindNode
NiosDLinkAfter
NiosDLinkPrevious
NiosDLinkFirst
NiosDLinkLast
NiosDNext
NiosDNextNode
NiosDPreviousNode
NiosDUnlinkFirst
NiosDUnlinkLast
NiosDUnlinkNode

NiosEatWhite

Description

Eliminates leading white space characters from the string.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosEatWhite(
    UINT8        *string);
```

Parameters

string	Pointer to the string to service.

Returns

A pointer to a new string position

Remarks

The following character values are considered white space:
  01h - 20h
  , (Comma)

See Also

NiosEnableLogging

Description

Enables or disables logging

Syntax

```
UINT32
NiosEnableLogging (
    UINT32      loggingMode );
```

Parameters

loggingMode	Either NIOS_LOG_ENABLE or NIOS_LOG_DISABLE

Returns

Previous logging mode

Remarks

See Also

NiosEnumLoadedModules

Description

Allows the caller to enumerate the currently loaded NLM modules.

Syntax

```
#include <module.h>

UINT32
NiosEnumLoadedModules (
     void              **context,
     modHandle         retBuf,
     UINT32            sizeRetBuf);
```

Parameters

context	Pointer to void **context* variable. This variable is used by this service to establish the context for the next invocation of this function. To begin the enumeration **context* must be set to NULL.
retBuf	Pointer to *LoadedModuleStruct* structure which receives a copy of the current module's module information.
sizeRetBuf	Size of buffer pointed to by *retBuf.*

Returns

0	Enumeration successful. Return buffer has been filled out.
0xFFFFFFFF	No more modules.
0xFFFFFFFE	Enumeration context lost. One or more modules were added or deleted from the system between calls to **NiosEnumLoadedModules**. The caller should start the enumeration over.

Remarks

See Also

NiosExamineSemaphore

Description

Allows the caller to examine the current token count of the specified semaphore.

Syntax

```
#include <nios.h>

SINT32
NIOSExamineSemaphore(
        semHandle  handle);
```

Parameters

handle	Specifies a handle indentifying the semaphore to examine.

Returns

Current semaphore token count.

Remarks

This function allows the caller to selectively decide when to call semaphore "up" or "down" services.

When the token count is negative, a thread is currently blocked on the semaphore. When 0, a call to **NiosWaitSemaphore** will block the caller unless **NiosSignalSemaphore** is called by an interrupt procedure between the time **NiosExamineSemaphore** is called and **NiosWaitSemaphore** is called. To make the examine and wait autonomous simply disable interrupts around the two function calls. When above 0, the value represents how many times a wait operation on the semaphore can occur before the wait call would block.

See Also

NiosCreateSemaphore, NiosCreateSemaphoreEx,
NiosWaitSemaphore, NiosDestroySemaphore
NiosSignalSemaphore

NiosFindNode

Description

Tests if queue entry *key* is a member of the specified queue.

Syntax

```
#include <niosq.h>

UINT32
NiosFindNode (
    void            *key,
    slinkQueue  *queue );
```

Parameters

key	Pointer to search node
queue	Queue to search for key node

Returns

Zero if node is found
Non-zero if node is not located

Remarks

See Also

NiosLinkAfter
NiosLinkFirst
NiosNextNode
NiosUnlinkAfter
NiosUnlinkFirst
NiosUnlinkNode

NiosFree

Description

Frees a block of memory that was previously allocated using **NiosLongTermAlloc**, **NiosShortTermAlloc**, or **NiosPhysContigAlloc**.

Syntax

```
#include <nios.h>

UINT32
NiosFree(
    modHandle     module,
    void          *ptr);
```

Parameters

module	Caller's module handle. This handle must be the same handle used in allocating the block.
ptr	Linear address of block to free.

Returns

0	Invalid pointer
!0	Memory was freed

Remarks

This function can be passed a null *ptr* parameter in which case this service is effectively a NOP.

See Also

NiosFreeHandle

Description

Deallocates a handle for a given linear address.

Syntax

```
#include <handlmgr.h>

UINT32
NiosFreeHandle (
     UINT32     clientID,
     UINT32     handle );
```

Parameters

clientID	Unique key that allows handle manager to identify caller
handle	Handle to deallocate

Returns

SUCCESS_CODE
errINVALID_CLIENT_ID
errINVALID_HANDLE

Remarks

See Also

NiosRegisterHandleClient
NiosDeregisterHandleClient
NiosGetHandle
NiosListHandles
NiosHandletoAddress
NiosAddressToHandle

NiosGetCountryInfo

Description

Provides country-specific information.

Syntax

```
#include <nios.h>

UINT32
NiosGetCountryInfo (
    UINT32              revisionExpected,
    NiosCountryInfo     *infoBlock );
```

Parameters

revisionExpected	Revision of the NiosCountryInfo block that the caller expects to receive a copy of. This allows for future expansion of the structure. Currently this value should be set to one (1).
infoBlock	Pointer to *NiosCountryInfo* structure which will be filled out on return.

Returns

Revision of *NiosCountryInfo* structure.

Remarks

See Also

NiosGetCurrProcessGroupId

Description

Returns the ID assigned to the currently executing process group.

Syntax

```
#include <nios.h>

UINT32
NiosGetCurrProcessGroupId(
    void);
```

Parameters

None

Returns

Current Process Group ID

Remarks

This number is guaranteed to be between 1 and NIOS_MAX_PROCESS_GROUPS inclusive. Typically, an environment- independent NLM will use this number to track resources allocated to a group of processes that share common resources.

The system or first process group will be assigned a value of 1. Subsequent IDs are not guaranteed to be contiguous, so no assumptions about order of assignment should be made.

Process group IDs can be reused.

See Also

NIOS_MAX_PROCESS_GROUPS (NIOS.H)
NIOS Event "NIOS PROCESS GROUP CREATE"
NIOS Event "NIOS PROCESS GROUP DESTROY"
NiosGetCurrProcessId

NiosGetCurrProcessId

Description

Returns the ID assigned to the currently executing process.

Syntax

```
#include <nios.h>

UINT32
NiosGetCurrProcessId(
    void);
```

Parameters

None

Returns

Current Process ID

Remarks

The return value is environment specific, however it is guaranteed to be unique.

Trypically an environment independent NLM will use this value to track resources allocated to a process.

Process ID's can be reused.

See Also

NiosGetCurrProcessGroupId
NiosGetProcessName

NiosGetDateTime

Description

Returns the current date and time.

Syntax

```
#include <nios.h>

void
NiosGetDateTime (
     NDateTime  *dateTime);
```

Parameters

dateTime Pointer to NDateTime structure which will be set on return

Returns

Nothing

Remarks

Structures

NDateTime Structure used by **NiosGetDateTime** and **NiosSetDateTime**.

```
typedef struct NDateTimeStruc

UINT8 NDTHour;         // (0-23)
UINT8 NDTMinute;       // (0-59)
UINT8 NDTSecond;       // (0-59)
UINT8 NDTReserved;
UINT8 NDTDay;          // (1-31)
UINT8 NDTMonth;        // (1-12)
UINT16 NDTYear;        // (1980-2079)
}NDateTime;
```

See Also

NiosGetHandle

Description

Gets a handle for a given linear address.

Syntax

```
#include <handlmgr.h>

UINT32
NiosGetHandle (
    UINT32      clientID,
    void        *address
    UINT32      *handle );
```

Parameters

clientID	Unique key that allows handle manager to identify caller.
address	A 32-bit linear address for which client wishes to allocate a handle.
handle	Handle supplied by handle manager for given address. Ignore if return code does not indicate success.

Returns

SUCCESS_CODE
errINVALID_CLIENT_ID
errINVALID_PARAMETER
errOUT_OF_MEMORY

Remarks

See Also

NiosRegisterHandleClient
NiosDeregisterHandleClient
NiosListHandles
NiosFreeHandle
NiosHandletoAddress
NiosAddressToHandle

NiosGetHighResIntervalMarker

Description

Gets the current high-resolution interval marker which has a resolution of 838 nanoseconds.

Syntax

```
#include <aes.h>

UINT32
NiosGetHighResIntervalMarker(
      void);
```

Parameters

None

Returns

Elapsed time in 838 ns units
Interrupt state is preserved and not enabled
Standard "C" regs preserved

Remarks

The value wraps once an hour. Note that most applications using this function can treat the resolution as being that of 1 microsecond instead of 838 nanoseconds. The loss of precision is minimal.

This function is callable at interrupt time.

See Also

NiosGetIntervalMarker
NiosGetTickCount

NiosGetIntervalMarker

Description

Returns the current number of milliseconds (ms) elapsed since NIOS was loaded.

Syntax

```
#include <aes.h>

UINT32
NiosGetIntervalMarker(
      void);
```

Parameters

None

Returns

EAX	System time in milliseconds

All other registers are preserved
Interrupt states are preserved and never enabled

Remarks

This function is callable from interrupt context in all environments.

See Also

NiosGetTickCount
NiosGetHiResIntervalMarker

NiosGetMemInfo

Description

Returns information about the NIOS memory allocator.

Syntax

```
#include <nios.h>

void
NiosGetMemInfo(
    MemInfo *memInfoBlock);
```

Parameters

memInfoBlock	Pointer to MemInfo structure that will be filled with information on return

Returns

memInfoBlock filled

Remarks

Structures

```
typedef struct MemInfoStruc

UINT32 MITotalSysFree;
UINT32 MITotalSubFree;
UINT32 MILargestSubFreeBlock;
UINT32 MITotalAlloced;
UINT32 MIAllocOverhead;
UINT32 MIAvgAllocSize;
UINT32 MITotalPhysAlloced;
}MemInfo;
```

See Also

NiosGetModHandleFromName

Description

Locates the module handle for the specified named module.

Syntax

```
#include <module.h>

modHandle
NiosGetModHandleFromName(
     UINT8        *moduleName);
```

Parameters

moduleName Pointer to ASCIIz module name (e.g. "LSL.NLM")

Returns

0	Module is not loaded
!0	Module handle of module

Remarks

See Also

NiosGetPhysLinearStart

Description

Returns the base linear address of a linear range that maps directly to all physical memory in the system.

Syntax

```
#include <nios.h>

UINT32
NiosGetPhysLinearStart(
                void);
```

Parameters

None

Returns

Base linear address value

Remarks

The returned value can be used to convert from physical to logical (adding the value) and from logical to physical (subing the value). This function can only be used on memory that was allocated using the **NiosPhysContigAlloc** function, since memory allocated using the normal allocation calls may not be physically contiguous.

See Also

NiosGetProcessName

Description

Returns a displayable description of the specified process.

Syntax

```
#include <nios.h>

UINT32
NiosGetProcessName(
            UINT32     processGroupId,
            UINT32     processId,
            UINT8      *retBuf);
```

Parameters

processGroupId	Id of the process group that the specified process Id is part of. This parameter is typically set to a value obtained from the **NiosGetCurrProcessGroupId** service. If set to PROCESS_GROUP_NLM, the processId parameter must be a valid NLM module handle.
processId	Id of the process. This parameter is typically set to a value obtained from the **NiosGetCurrProcessId** service. This parameter must be a valid NLM module handle if processGroupId is set to PROCESS_GROUP_NLM.
retBuf	Pointer to a buffer whose length is MAX_PROCESS_NAME_LEN. This buffer will be set on return.

Returns

Zero	Function was successful.
Non-zero	Unable to determine process name. The *retBuf* parameter will be set to "UNKNOWN".

Remarks

This service supports both Ring 3 user level applications as well as NLMs.

There are cases when this service cannot determine the name of a process. In these cases *retBuf* will be set to "UNKNOWN".

See Also	NiosGetCurrProcessGroupId NiosGetCurrProcessId

NiosGetSystemDirectory

Description

Returns a copy of the fully qualified path specification of where NetWare-related files are stored.

Syntax

```
#include <nios.h>

UINT8
*NiosGetSystemDirectory(
    UINT8       *retStringBuf,
    UINT32      bufLength );
```

Parameters

retStringBuf	Pointer to a buffer that will receive a copy of the path.
bufLength	Size in bytes of the buffer pointed to by retStringBuf.

Returns

zero	Successful.
non-zero	Buffer is too small to hold the result. In this case the return value is the length of buffer needed to hold the result including the NULL terminator.

Remarks

The path returned is typically the path from which NIOS was loaded. The returned ASCIIZ string will be properly formed to allow simple filename concatenation.

Typically NLM modules and system configuration files are located in this directory.

See Also

NiosGetTickCount

Description

Gets the current tick count calculated at 18.2 ticks/second.

Syntax

```
#include <aes.h>

UINT32
NiosGetTickCount(
    void);
```

Parameters

None

Returns

EAX   System time in ticks
All other registers are preserved
Interrupt state preserved and never enabled

Remarks

NiosGetIntervalMarker is preferred over this function since it is typically faster than this function.

This function is callable at interrupt time in all environments.

See Also

NiosGetIntervalMarker
NiosGetHiResIntervalMarker

NiosGetVersion

Description

Returns version information about the NIOS interface.

Syntax

```
#include <nios.h>

UINT32
NiosGetVersion(
    void);
```

Parameters

None

Returns

Upper 16 bits of EAX	NIOS Type Value
AH	NIOS major version
AL	NIOS minor version

Remarks

The following values are returned by **NiosGetVersion**:

```
#include <nios.h>

#define     NIOS_FOR_DOSWIN_VMM      0
#define     NIOS_FOR_WIN4X_VMM       1
#define     NIOS_FOR_NETWARE_OS      2
```

See Also

NiosHandleToAddress

Description

For a given handle, the handle manager will return a linear address for the handle if one is allocated.

Syntax

```
#include <handlmgr.h>

UINT32
NiosHandleToAddress (
    UINT32      clientID,
    UINT32      handle,
    void        *address );
```

Parameters

clientID	Unique key that allows handle manager to identify caller
handle	Handle client wishes to identify 32-bit address for
address	Return value indicating 32-bit address associated with handle

Returns

SUCCESS_CODE
errINVALID_CLIENT_ID
errINVALID_HANDLE

Remarks

See Also

NiosRegisterHandleClient
NiosDeregisterHandleClient
NiosGetHandle
NiosFreeHandle
NiosListHandles
NiosAddressToHandle

NiosHexCharToByte

Description	Converts the specified hex alphanumeric character into a byte. The base of the alphanumeric character is always 16.

Syntax

```
#include <nstdlib.h>

UINT8
NiosHexCharToByte(
      UINT8        c);
```

Parameters	*c*	ASCII hex alphanumeric character ('0'-'9','A'-'F')

Returns	Byte representation of the char (0x0-0xF)

Remarks

See Also

NiosHookExportedApi

Description

Allows an NLM to intercept functions and data variables that have been exported by NIOS or other NLMs.

Syntax

```
#include <module.h>

UINT32
NiosHookExportedApi(
    modHandle      moduleHandle,
    UINT8          *apiName,
    UINT32         newApiAddress);
```

Parameters

moduleHandle	Caller's module handle
apiName	Name of API to hook
newApiAddress	New address for API

Returns

0	Export hooked successfully
!0	No memory to create new export

Remarks

This function also allows on-the-fly creation of new exported publics.

Note that an NLM loaded prior to making this call will not be modified by this call and will use the old API address. Thus, to globally intercept an API, an NLM must be loaded prior to any other NLM that would use the API.

This service can also be used to dynamically add an exported public to the system. If the export is not currently defined in the system, this function will create it.

See Also

NiosUnHookExportedApi

NiosHookHardwareInt

Description

Attempts to hook the specified hardware interrupt vector.

Syntax

```
#include <nios.h>

UINT32
NiosHookHardwareInt(
    modHandle     moduleHandle,
    UINT32        hardIntNumber,
    void          (*interruptProc)(void),
    UINT32        options,
    UINT32        refData);
```

Parameters

moduleHandle	Pointer to caller's module handle.
hardIntNumber	IRQ to hook (0-15).
interruptProc	A pointer to the routine that is invoked when the specified hardware interrupt occurs. Entry and exit conditions are: On entry: EDX *refData* Interrupts are disabled CLD has been executed On exit: Interrupts are disabled All registers can be destroyed CLD preserved Z flag set if int was serviced else pass int to next handler
options	Possible values are: HIOPT_SHAREABLE_BIT Interrupt can be shared with others that support sharing
refData	Reference data passed to **interruptProc** in register EDX

Returns

0	Interrupt was successfully hooked

0xFFFFFFFF	The specified interrupt is already hooked by a handler that doesn't support shareable interrupts
0xFFFFFFFE	The specified interrupt is already hooked by a handler that requires other handlers to support sharing
0xFFFFFFFD	Specified IRQ value is above 15
0xFFFFFFFC	Not enough free memory to hook the interrupt

Remarks

See Also

NiosUnHookHardwareInt
DoEndOfInterrupt
EnableHardwareInterrupt
DisableHardwareInterrupt
CheckHardwareInterrupt

NiosImportNlmApi

Description

Determines the linear address of the specified NLM API name. The returned address can then be used to access the API by either calling it, in the case of a function, or accessing it, in the case of a data structure.

Syntax

```
#include <module.h>

void *
NiosImportNlmApi(
    modHandle moduleHandle,
    UINT8 *apiName);
```

Parameters

moduleHandle	Caller's module handle. If NULL, an anonymous dependency is built.
apiName	Name of API to resolve (case insensitive).

Returns

0	API does not exist
!0	Linear address of API

Remarks

This function allows an NLM to late bind to an API instead of specifying it in its DEF file import statement. Using this function builds a dependency between the caller and the NLM that owns the specified *apiName*.

If the caller specifies an anonymous dependency then the caller must use the **NiosDeportNlmApi** service to remove the dependency when finished using *apiName*. If a real module handle is passed in, then **NiosDeportNlmApi** should not be used. In this case the dependency will be removed when the calling NLM unloads.

See Also

NiosDeportNlmApi

NiosIsPhysContig

Description

Determines whether the specified memory block is physically contiguous.

Syntax

```
#include <nios.h>

UINT32 NiosIsPhysContig(
      void          *memoryBlock,
      UINT32        length);
```

Parameters

memoryBlock	Linear address of memory block to check
length	Length of memory block in bytes

Returns

0	Memory is *not* contiguous
!0	Memory is contiguous

Remarks

If the NIOS "PHYS CONTIGUOUS MEM" configuration parameter is set to OFF, this service will always return a non-zero value.

See Also

NiosKeywordDeRegister

Description

Deregisters a keyword from the system.

Syntax

```
#include <nioscfg.h>

UINT32
NiosKeywordDeRegister (
    ModHdlP    modHandle,
    UINT32     cfgHandle);
```

Parameters

modHandle	Module handle of the calling NLM or 0xFFFFFFFF if permanent registration
cfgHandle	Config info handle returned during **NiosKeywordRegister**

Returns

SUCCESS_CODE
NC_INVALID_CFG_HANDLE

Remarks

See Also

NiosKeywordRegister
NiosKeywordResetValue
NiosKeywordEnumerate
NiosKeywordSetValue
NiosKeywordUpdateNetCfg

NiosKeywordEnumerate

Description

Retrieves configuration keyword information.

Syntax

```
#include <nioscfg.h>

UINT32
NiosKeywordEnumerate (
    UINT32              *searchIndex,
    FindKeywordInfo     *findInfo,
    FoundKeywordInfo    *foundInfo);
```

Parameters

searchIndex	Address of buffer to hold search index. This must be 0xFFFFFFFF to start and will be modified for subsequent calls to enumerate through all keywords.
findInfo	Address of buffer to retrieve the following:

ModHandle	modHandle or 0xFFFFFFFF for any
DataType	dataType or 0xFFFFFFFF for any
SectionNameLength	Reply buffer size for foundInfo.SectionName
**SectionName*	sectionName or NULL for any
KeywordNameLength	Reply buffer size for foundInfo.KeywordName
**KeywordName*	keywordName or NULL for any
HelpTextLength	Reply buffer size for foundInfo.HelpText
Reserved	Reserved for future use
DefaultStrLength	Reply buffer size for foundInfo.DefaultStr
CurrentStrLength	Reply buffer size for foundInfo.CurrentStr

foundInfo Address of buffer to store the following:

ModHandle	modHandle
DataType	dataType
SectionNameLength	Section name length
**SectionName*	Section name copied to
KeywordNameLength	Keyword name length
**KeywordName*	Keyword name copied to
HelpTextLength	Help text length
**HelpText*	Help text copied to

if a UINT32

MinValue	Minimum value
MaxValue	Maximum value
DefaultValue	Default value
CurrentValue	Current value

if a binary data type or string

MinStrLength	Minimum length of data
MaxStrLength	Maximum length of data
DefaultStrLength	Default data length
**DefaultStr*	Default data copied to
CurrentStrLength	Current data length
**CurrentStr*	Current data copied to

Returns

If successful, *foundInfo* filled out

SUCCESS_CODE	
NC_NO_MORE_ENTRIES	if search has been exhausted

Remarks

If any of the lengths returned are greater than the size for the reply buffers, the reply has been truncated to the length specified in the *foundInfo* fields. When this happens the caller can reallocate additional buffers space for the size indicated by the returned length and re-request using the previous *searchIndex*.

See Also

NiosKeywordRegister, NiosKeywordDeRegister
NiosKeywordResetValue, NiosKeywordSetValue
NiosKeywordUpdateNetCfg

NiosKeywordRegister

Description

Registers a callback that will be invoked when the specified keyword's value is changed in the configuration database.

Syntax

```
#include <nioscfg.h>

UINT32
NiosKeywordRegister (
    ModHdlP                 modHandle,
    RegisterKeywordInfo    *registerInfo,
    UINT32                 *cfgHandle );
```

Parameters

modHandle Module handle of the calling NLM.

registerInfo Address of buffer to retrieve the following:

dataType	Specifies type of keyword (string, int, etc.). See NIOSCFG.H CFG_??? for definitions.
attribute	Specifies READ or READ/WRITE value. See NIOSCFG.H KEYWORD_??? for definitions.
section	Address of section name (must be NULL terminated).
keyword	Address of keyword (must be NULL terminated).
currentValue	Address of the value to change during runtime, and should initially be the default value. If the configuration database specifies a different value, the "changed Call Back" procedure will be called with the address of the new value. If this procedure returns successfully, the new value will be copied into this current value address.
currentValueLength	Current Size of the space needed to hold the currentValue (i.e., UINT32 should have a 4 here).
minValue	Minimum value (minimum string length if a string value).

maxValue	Maximum value (maximum string length if a string value).
helpText	Address of help text for keyword (must be NULL terminated). Put a zero in this field if no help text is desired.
qualifyCallBack	Address to call before the value changes during runtime. If NULL, no callback is necessary; however, the currentValue is changed.

chgHandle	Address to store the config info handle to be used for the **NiosKeywordDeRegister**.

Returns

SUCCESS_CODE
GENERAL_ERROR
NC_OUT_OF_CLIENT_MEMORY
NC_INVALID_MODULE_HANDLE
NC_KEYWORD_ALREADY_REGISTERED
Keyword passed in was already in registry. The client was hooked to the keyword, but the keyword values specified did not override previous values.

Remarks

See Also

NiosKeywordResetValue
NiosKeywordDeRegister
NiosKeywordEnumerate
NiosKeywordSetValue
NiosKeywordUpdateNetCfg

NiosKeywordResetValue

Description

Resets a keyword value to the default.

Syntax

```
#include <nioscfg.h>

UINT32
NiosKeywordResetValue (
    UINT8       *section,
    UINT8       *keyword);
```

Parameters

section	Address of section name
keyword	Address of keyword

Returns

SUCCESS_CODE
NC_KEYWORD_NOT_FOUND
NC_KEYWORD_READ_ONLY

Remarks

See Also

NiosKeywordRegister
NiosKeywordDeRegister
NiosKeywordEnumerate
NiosKeywordSetValue
NiosKeywordUpdateNetCfg

NiosKeywordSetValue

Description

Sets a keyword value.

Syntax

```
#include <nioscfg.h>

UINT32
NiosKeywordResetValue (
     UINT8      *section,
     UINT8      *keyword,
     UINT32     newValueLength,
     void       *newValue);
```

Parameters

section	Address of section name
keyword	Address of keyword
newValueLength	String length or 4 for UINT32
newValue	Address of new value

Returns

SUCCESS_CODE
NC_KEYWORD_NOT_FOUND
NC_KEYWORD_READ_ONLY

Remarks

See Also

NiosKeywordRegister
NiosKeywordDeRegister
NiosKeywordEnumerate
NiosKeywordResetValue
NiosKeywordUpdateNetCfg

NiosKeywordUpdateNetCfg

Description

Flushes the registered keywords to the configuration file.

Syntax

```
#include <nioscfg.h>

UINT32
NiosKeywordUpdateNetCfg (
      UINT8       *sectionName,
      UINT8       *keywordName);
```

Parameters

sectionName	Address of section name (NULL for all)
keywordName	Address of keyword (NULL for all)

Returns

SUCCESS_CODE
NC_KEYWORD_NOT_FOUND

Remarks

See Also

NiosKeywordRegister
NiosKeywordDeRegister
NiosKeywordEnumerate
NiosKeywordSetValue
NiosKeywordResetValue

NiosLinkFirst

Description

Inserts a node into the front of a singly linked list.

Syntax

```
#include <niosq.h>

void
NiosLinkFirst (
        void            *insertNode,
        slinkQueue  *queue );
```

Parameters

insertNode	Object to be placed on queue
queue	Queue to insert item

Returns

Nothing

Remarks

The list is assumed to be null terminated. The queue structure passed points to the head and tail nodes of the linear linked list.

Queue nodes must include a forward link field. The offset to this field must be provided to the queueing routine.

See Also

NiosFindNode
NiosLinkNext
NiosNextNode
NiosUnlinkAfter
NiosUnlinkFirst
NiosUnlinkNode

NiosLinkLast

Description

Inserts a node at the end of a singly linked list.

Syntax

```
#include <niosq.h>

void
NiosLinkLast (
      void              *insertNode,
      slinkQueue  *queue );
```

Parameters

insertNode	Object to be placed on queue
queue	Queue to insert item

Returns

Nothing

Remarks

The list is assumed to be null terminated. The queue structure passed points to the head and tail nodes of the linear linked list.

Queue nodes must include a forward link field. The offset to this field must be provided to the queueing routine.

See Also

NiosFindNode
NiosLinkNext
NiosNextNode
NiosUnlinkAfter
NiosUnlinkFirst
NiosUnlinkNode

NiosLinkNext

Description

Inserts a node (*insertNode*) after the specified node (*afterNode*) in a singly linked list.

Syntax

```
#include <niosq.h>

void
NiosLinkNext (
    void            *insertNode,
    void            *afterNode,
    slinkQueue  *queue );
```

Parameters

insertNode	Object to be placed following *afterNode* in the queue
afterNode	Node that *insertNode* will be placed after
queue	Queue to insert item

Returns

Nothing

Remarks

The list is assumed to be null terminated. The queue structure parameter points to the head and tail nodes of the linear linked list.

Queue nodes must include a forward link field. The offset to this field must be provided to the queueing routine.

See Also

NiosFindNode
NiosLinkFirst
NiosNextNode
NiosUnlinkAfter
NiosUnlinkFirst
NiosUnlinkNode

NiosLinToPhys

Description

Returns the physical address of a specified linear address.

Syntax

```
#include <nios.h>

UINT32
NiosLinToPhys(
    void      *LinearAddress);
```

Parameters

LinearAddress	Memory address for which to return physical address

Returns

-1	Memory address is not present
else	Physical address

Remarks

This function returns an error if the specified memory address is not present.

The physical address of a piece of memory should not be used if the memory's pages have not been previously page-locked, since the linear range's physical address can change without notice.

If the memory is not physically contiguous, the returned physical address may be invalid if used to access beyond the end of the physical page mapped to *LinearAddress*.

See Also

NiosListHandles

Description

Enumerates on a given client's handles.

Syntax

```
#include <handlmgr.h>

UINT32
NiosListHandles (
    UINT32      clientID,
    UINT32      *handleIndex );
```

Parameters

clientID	Unique key that allows handle manager to identify caller.
handleIndex	Allocated client handle. This value on initial enumeration should be set to 0xFFFFFFFF. When handle manager finds no more handles, a zero value will be returned.

Returns

SUCCESS_CODE
NO_MORE_HANDLES
errINVALID_CLIENT_ID

Remarks

See Also

NiosRegisterHandleClient
NiosDeregisterHandleClient
NiosGetHandle
NiosFreeHandle
NiosHandletoAddress
NiosAddressToHandle

NiosLoadModule

Description

Loads and executes a client NLM.

Syntax

```
#include <module.h>

UINT32
NiosLoadModule(
     UINT32          loadOptions,
     UINT8           *modulePathSpec,
     UINT8           *commandLine,
     UINT32          nlmFileOffset,
     modHandle       *retModHandle);
```

Parameters

loadOptions	Bits defining loading styles. All undefined bits must be set to zero. LOPTION_DEBUG_INIT Executes an Int 1 before the loader invokes the module's init routine. LOPTION_ERROR_MSGS Stdout error messages are enabled. LOPTION_BANNER_MSGS Stdout signon messages are enabled.
modulePathSpec	[Path\]name of module to load (with extension).
commandLine	Pointer to an ASCIIZ string containing parameters that will be passed to the loading module.
nlmFileOffset	Offset from the start of the *modulePathSpec* file where the NLM image starts. Typically this will be zero for straight .NLM files.
retModHandle	Pointer to a modHandle that will be set to the newly loaded module's handle on success. If NULL, the module handle will not be returned.

Returns

LOADER_SUCCESS
Module was loaded successfully.

LOADER_NO_LOAD_FILE
Open load file failed.

LOADER_IO_ERROR
File I/O error during read.

LOADER_INSUFFICIENT_MEMORY
Not enough memory to load module.

LOADER_INVALID_MODULE
Invalid NLM module.

LOADER_UNDEFINED_EXTERN
Referenced undefined external item.

LOADER_DUPLICATE_PUBLIC
Exported public is already defined.

LOADER_NO_MSG_FILE
Open message file failed.

LOADER_INVALID_MSG_MODULE
Message file is malformed.

LOADER_MODULE_ALREADY_LOADED
Module cannot be loaded more than once.

LOADER_BAD_REENTRANT_MODULE
Reentrant load failed because the module is not the same version as the first module.

LOADER_MODULE_INIT_FAILED
Module failed to initialize.

LOADER_LOAD_REFUSED
A loaded NLM refuses to allow this NLM to load.

Remarks

NiosLongTermAlloc

Description

Allocates a block of memory of the specified size for long-term use by the caller.

Syntax

```
#include <nios.h>

void
*NiosLongTermAlloc(
    modHandle    module,
    UINT32       size);
```

Parameters

module	Caller's module handle
size	Number of bytes to allocate

Returns

0	Allocation failed
!0	Valid memory pointer

Remarks

Use **NiosLongTermAlloc** when the requested memory is going to be used for a relatively long period of time (e.g., for the lifetime of the module or connection).

Allocating memory using the appropriate function will reduce memory fragmentation.

All memory allocated using this function is locked (that is, is non-pageable, is always present, etc.). However, it is not guaranteed to be physically contiguous.

See Also

NiosShortTermAlloc
NiosPhysContigAlloc
NiosFree

NiosMapPhysMemory

Description

Allocates a linear address range that maps to the specified physical range.

Syntax

```
#include <nios.h>

UINT32
NiosMapPhysMemory(
     UINT32      physAddress,
     UINT32      length);
```

Parameters

physAddress	Physical address of memory to map
length	Length of memory to map

Returns

0	No resources available to create linear range
!0	Linear address of start of physAddress

Remarks

NiosMapPhysMemory allows access to physical memory outside of the normal range of system RAM addresses (typically some type of adapter RAM/ROM).

Linear addresses allocated using this function cannot be converted to physical addresses using **NiosLinToPhys**. If *physAddress* can be addressed using an existing linear range, then the existing range is used.

See Also

NiosMemCmp

Description

Performs case-sensitive compare of *len* bytes of memory from *ptr1* to *ptr2*.

Syntax

```
UINT32
NiosMemCmp(
    void        *ptr1,
    void        *ptr2,
    UINT32      len);
```

Parameters

ptr1	Linear ptr to buffer 1
ptr2	Linear ptr to buffer 2
len	Number of bytes to compare

Returns

0	Memory at *ptr1* == memory at *ptr2*
!0	Memory at *ptr1* != memory at *ptr2*

Remarks

See Also

NiosMemCmpi

Description

Performs case-insensitive compare of *len* bytes of memory from *ptr1* to *ptr2*.

Syntax

```
#include <nios.h>

UINT32
NiosMemCmpi(
      void          *ptr1,
      void          *ptr2,
      UINT32        len);
```

Parameters

ptr1	Linear pointer to buffer 1
ptr2	Linear pointer to buffer 2
len	Number of bytes to compare

Returns

0	Memory at *ptr1* == memory at *ptr2*
!0	Memory at *ptr1* != memory at *ptr2*

Remarks

See Also

NiosMemCpy

Description

Copies the contents of one memory buffer to another.

Syntax

```
#include <nios.h>

void
*NiosMemCpy(
    void        *dst,
    void        *src,
    UINT32      len);
```

Parameters

len	Number of bytes to copy
dst	Pointer to destination buffer
scr	Pointer to source buffer

Returns

Changes made to destination buffer.

Remarks

See Also

NiosMemPoolAlloc

Description

This routine is identical to **NiosMemPoolFindBlock** except that it assumes the block has not yet been allocated and avoids looking up the block.

Syntax

```
#include <mempool.h>

UINT32
NiosMemPoolAlloc (
    UINT32          mpID,
    UINT32          options,
    UINT32          key1,
    UINT32          key2,
    MemPoolHandle   **handle,
    void            **buffer );
```

Parameters

See NiosMemPoolFindBlock

Returns

See NiosMemPoolFindBlock

Remarks

See Also

NiosMemPoolFindBlock

NiosMemPoolCheckAvail

Description

Determines if a request should be cached or go direct. This function returns the number of blocks remaining in the pool which are not allocated to the requesting application.

Note: This does not account for blocks that are currently held.

Syntax

```
#include <mempool.h>

UINT32
NiosMemPoolCheckAvail (
    UINT32          mpID,
    UINT32          blocksNeeded,
    UINT32          *blocksAvail );
```

Parameters

mpID	(IN) Application memory pool ID
blocksNeeded	(IN) Number of blocks needed
*blocksAvai*l	(IN) Pointer to variable to receive blocks available (may be NULL)
blocksAvail	(OUT) Filled with number of blocks not allocated by the application.

Returns

zero	Successful (blocksAvail >= blocksNeeded)
non-zero	Failure (blocksAvail < blocksNeeded)

Remarks

See Also

NiosMemPoolGetSize

NiosMemPoolDeRegister

Description

An application should call this routine before exiting in order to release any resources (including the callback) that have been allocated.

This routine will clean up all memory block holds, etc. which the application may currently have.

Syntax

```
#include <mempool.h>

UINT32
NiosMemPoolDeRegister (
    modHandle       moduleHandle,
    UINT32          mpID );
```

Parameters

moduleHandle	(IN) Module handle of the registering NLM
mpID	(IN) Application memory pool ID to be altered
mpID	(OUT) Invalidated by the memory pool

Returns

zero	Successful
non-zero	Failure, may result from invalid modHandle or mpID

Remarks

See Also

NiosMemPoolRegister

NiosMemPoolEnum

Description

Enumerates all of the memory blocks held by a particular application. An application can use this when it is inconvenient to maintain its own list of blocks (i.e., a background flush, exit routine, etc.) The callback function will be called once for each memory block held by the application each time this routine is invoked.

The callback can perform any NiosMemPool... function desired on the handle, including free.

Syntax

```
#include <mempool.h>

void
NiosMemPoolEnum (
      UINT32          mpID,
      void            (*callback)(
                      void          *handle,
                      void          *buffer,
                      UINT32         holdCount));
```

Parameters

mpID Memory pool application ID

callback Pointer to function to be called for each memory block

handle	Pointer to the memory handle
buffer	Pointer to the memory buffer
holdCount	Number of holds placed on the memory block

Returns

Nothing

Remarks

For blocks allocated while the enumerate is active, this function is non-deterministic. In other words, it may or may not pass the new handle to the enumerate callback.

See Also

NiosMemPoolFindBlock

Description

This function has multiple uses. An application can use this function to look up a previously allocated block, or to allocate a new block.

Syntax

```
#include <mempool.h>

UINT32
NiosMemPoolFindBlock (
    UINT32              mpID,
    UINT32              options,
    UINT32              key1,
    UINT32              key2,
    MemPoolHandle       **handle,
    void                **buffer );
```

Parameters

mpID	(IN) Application memory pool ID
options	(IN) Set of flags 'ORed' together with the following options available: MP_CREATE - Create buffer if not found. MP_HOLD - Hold buffer if found (i.e. lock). MP_MAKE_MRU - Make buffer most recently used. MP_LOW_PRIORITY - Low-priority create (only valid with MP_CREATE, not valid with MP_MUST_CREATE). MP_MUST_CREATE - High-priority create. Must succeed if there are any clean, unheld buffers (only valid with MP_CREATE).
key1	(IN) Key to use to search for the block (i.e., sector/byte #, connection, etc.).
key2	(IN) Key to use to search for the block (i.e., volume, file handle, etc.).
handle	(IN) Pointer to variable to receive memory handle (must not be NULL).
buffer	(IN) Pointer to variable to receive memory pointer (must not be NULL).
handle	(OUT) Filled with memory handle (if return value == 0).

buffer	(OUT) Filled with memory pointer (if return value == 0).

Returns

zero	Successfully found or created
non-zero	Failure

Remarks

See Also

NiosMemPoolFreeBlock
NiosMemPoolMakeMRU
NiosMemPoolHold
NiosMemPoolAlloc

NiosMemPoolFreeBlock

Description

Releases the memory block. It is assumed that the application has performed any flush, unlink, etc., because the application's callback is not executed by this routine.

Syntax

```
#include <mempool.h>

void
NiosMemPoolFreeBlock (
      void      *handle );
```

Parameters

handle	(IN) Pointer to memory handle.
handle	(OUT) Memory handle removed from application's allocated list. LRU position unaltered.

Returns

Nothing

Remarks

See Also

NiosMemPoolFindBlock

NiosMemPoolGetSize

Description

Determines how many memory blocks are available to either the entire system (if mpID = 0) or to the application (if mpID != 0).

Syntax

```
#include <mempool.h>

void
NiosMemPoolGetSize (
     UINT32          mpID,
     UINT32          *blockCount );
```

Parameters

mpID	(IN) Application memory pool ID or zero
blockCount	(IN) Pointer to variable to receive the block count (must not be NULL)
blockCount	(OUT) If mpID = 0, number of blocks in system pool, else number of blocks available to the application (may be same as system)

Returns

Nothing

Remarks

See Also

NiosMemPoolCheckAvail

NiosMemPoolGetVersion

Description

Retrieves version information and memory option information.

Syntax

```
#include <mempool.h>

void
NiosMemPoolGetVersion (
     UINT32          *ver,
     UINT32          *options );
```

Parameters

ver	(IN) Pointer to variable to receive the version number (must not be NULL)
options	(IN) Pointer to variable to receive the options parameter (may be NULL)
ver	(OUT) Filled with version number (100h = 1.00)
options	(OUT) Reserved for future use (0)

Returns

Nothing

Remarks

See Also

NiosMemPoolHold

Description

Increments the hold count on a memory block. An application uses this function to prevent the system from recycling the memory. This function must be called before read or write access is made to the buffer.

Syntax

```
#include <mempool.h>

UINT32
NiosMemPoolHold (
     MemPoolHandle      *handle,
     void               *buffer);
```

Parameters

handle	(IN) Pointer to memory handle to be held
buffer	(IN) Pointer to memory buffer to be held
handle	(OUT) Hold count information is updated

Returns

zero	Error, no holds placed on the memory block
non-zero	Number of holds currently on the memory block

Remarks

On some systems, this will also need to lock the memory buffer to prevent it from being paged out.

See Also

NiosMemPoolFindBlock
NiosMemPoolTestHold
NiosMemPoolUnhold

NiosMemPoolMakeMRU

Description

An application can use this function to avoid the overhead of the FindBlock function with the MP_MAKE_MRU option.

Syntax

```
#include <mempool.h>

void
NiosMemPoolMakeMRU (
      void      *handle);
```

Parameters

handle	(IN) Pointer to a memory handle to be updated
handle	(OUT) The access information is updated

Returns

None

Remarks

See Also

NiosMemPoolFindBlock
NiosMemPoolMakeLRU

NiosMemPoolRegister

Description

Registers a module with the memory pool manager. The application provides the callback routine at this time and receives a memory pool reference ID to be used in future calls to the memory pool system.

Syntax

```
#include <mempool.h>

UINT32
NiosMemPoolRegister (
    modHandle       moduleHandle,
    UINT32              (*callback)(
                        UINT32              func,
                        MemPoolHandle       *handle,
                        void                *buffer),
    UINT32          *mpID);
```

Parameters

moduleHandle (IN) Module handle/pointer to as defined by NIOS. The module handle is used to track the resources allocated to an application and ensure that memory is not wasted if the application terminates abnormally.

callback (IN) Pointer to system callable function. The callback can return non-zero if a failure occurs, such as unable to flush buffer. It is recommended that the application return memory whenever possible to allow the system to perform properly. The callback may be called with interrupts disabled or enabled.

If the callback returns success (zero) then the **NiosMemPoolFreeBlock** function will be invoked in its behalf (the callback doesn't need to call it) (must not be NULL).

func	0 - Standard flush/release buffer
	1 - Critical flush/release buffer
	Others - reserved for future use
handle	Pointer to a memory handle
buffer	Memory buffer pointer

mpID (IN) Pointer to variable to receive the memory pool ID (must not be NULL)

mpID	(OUT) Filled with memory pool ID (if return value == 0)

Returns

zero	Successful.
non-zero	Unable to register with memory pool. May have insufficient memory or the maximum support has been exceeded.

Remarks

See Also NiosMemPoolDeRegister

NiosMemPoolTestHold

Description

Returns the number of holds placed on a memory lock.

Syntax

```
#include <mempool.h>

UINT32
NiosMemPoolTestHold (
      void      *handle);
```

Parameters

handle	(IN) Point to memory handle to be checked

Returns

zero	The memory block is not held
non-zero	The number of holds placed on the memory block

Remarks

See Also

NiosMemPoolHold
NiosMemPoolUnhold

NiosMemPoolUnhold

Description

Decrements the hold count on a memory block.

Syntax

```
#include <mempool.h>

UINT32
NiosMemPoolUnhold (
      void      *handle,
      void      *buffer);
```

Parameters

handle	(IN) Pointer to the memory handle to be released
buffer	(IN) Pointer to the memory buffer to be released
handle	(OUT) Hold count information is updated

Returns

zero	The last hold has been released
non-zero	The number of holds that still remain

Remarks

An application uses this function to allow the system to recycle the memory when it is through accessing it. This function should be called after read and write access to the buffer is complete.

On some systems, this may also need to unlock the memory buffer to make it available to be paged out.

See Also

NiosMemPoolHold
NiosMemPoolTestHold

NiosMemSet

Description

Initializes a memory buffer to a given value (*val*).

Syntax

```
#include <mempool.h>

#include <nios.h>

void
*NiosMemSet(
      void        *ptr,
      UINT8       val,
      UINT32      len);
```

Parameters

len	Number of bytes to set

Returns

Nothing

Remarks

See Also

NiosNextChar

Description

Advances the string pointer to the next character.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosNextChar(
     UINT8        *String);
```

Parameters

String	A NULL terminated string

Returns

Pointer to next character in the string.

Remarks

The pointer will not advance if the pointer is at the end of the string (that is, if it points to the NULL character).

See Also

NiosNextNode

Description

Takes a single linked queue entry and returns the next entry in the list.

Syntax

```
#include <niosq.h>

void
NiosNextNode (
    void      *node );
```

Parameters

node	Queue entry for which the next entry is desired

Returns

The next entry in the queue
Zero if none exists

Remarks

If no entries follow in the list, zero is returned.

See Also

NiosFindNode
NiosLinkNext
NiosLinkFirst
NiosUnlinkAfter
NiosUnlinkFirst
NiosUnlinkNode

NiosPageLock

Description

Locks the specified memory region so that the memory is present and fixed at constant physical addresses.

Syntax

```
#include <nios.h>

void
*NiosPageLock(
    void        *memPtr,
    UINT32      length,
    UINT32      flags);
```

Parameters

memPtr	Linear address of memory to lock. This function will abend if *memPtr* is invalid.
length	Number of bytes pointed to by *memPtr* to lock.
flags	Must be zero. Reserved for future use.

Returns

Pointer to linear address to use while accessing the memory pointed to by *memPtr* asynchronously. This value may equal *memPtr* in some cases. This value must be passed to the unlock function.

Remarks

The locked memory region cannot be demand-paged after this function returns successfully.

Memory which is accessed at interrupt must be page locked. If the current environment does not use demand paging, this function returns immediately.

See Also

NiosPageUnlock
NiosMemLockFlag

NiosPageUnlock

Description

Unlocks the specified memory region so that the memory can be demand-paged.

Syntax

```
#include <nios.h>

void
NiosPageUnlock(
    void        *memPtr,
    UINT32      length,
    UINT32      flags);
```

Parameters

memPtr	Linear address of memory to lock. THIS MUST BE THE ADDRESS RETURNED FROM **NiosPageLock**. The function will abend if *memPtr* is invalid.
length	Number of bytes pointed to by *memPtr* to unlock. This can be zero, in which case this function is a NOP.
flags	Must be zero. Reserved for future use.

Returns

Nothing

Remarks

The memory is actually unlocked when it has been *unlocked* the same number of times it was *locked* (that is, lock count goes to zero).

If the current environment does not support paging, this function returns immediately.

This function will abend if *memPtr* in invalid.

See Also

NiosPageLock
NiosMemLockFlag

NiosPhysContigAlloc

Description

Allocates the specified number of physically contiguous bytes of memory.

Syntax

```
#include <nios.h>

void
*NiosPhysContigAlloc(
     modHandle      module,
     UINT32         numBytes);
```

Parameters

module	Caller's module handle.
numBytes	Number of bytes to allocate. Bit 31 (0x80000000) can be set to indicate that memory does not need to be below the 16meg address boundary.

Returns

0	Allocation failed
!0	Valid memory pointer

Remarks

If this function is successful, the returned linear address is guaranteed to be physically contiguous and locked. The linear address is relative to the base linear address returned by **NiosGetPhysLinearStart**.

This function makes a best effort to allocate the requested memory. However, there are situations where the system may be unable to allocate the requested amount, particularly when the alloc request is greater than a page (4K) and the system is low on memory. Generally, requests for less than 4K will always succeed, unless the system has no free pages left.

Returned memory will always be aligned on a dword boundary, and is guaranteed to exist below the 16 megabyte address, unless bit 31 of the *numBytes* parameter is set or the user has configured NIOS to ignore the below16 meg restriction. See the "NIOS PHYS MEM BELOW 16 MEG" configuration parameter.

See Also

NiosPoll

Description

Yields (relinquishes) control of the processor to other foreground threads that need to run.

Syntax

```
#include <nios.h>

void
NiosPoll(
    void);
```

Parameters

None

Returns

Nothing
All registers are preserved
Interrupt states preserved, but interrupts may have been enabled

Remarks

NLMs that execute for extended periods of time without indirectly causing a yield to occur should invoke this function to allow other threads a chance to run.

This function is typically used while polling for some type of event to complete. This function can cause your current execution thread to block, potentially causing your code to be reentered. If called while in the context of a hardware interrupt, this function is effectively an NOP.

It is preferrable to use the **NiosThreadBlockOnId** services to wait for an event to complete instead of using this service. The reason is that this service allows the current process to still receive its time slice, which wastes CPU time. On the other hand, the block on Id services removes the process from the run queue until the event has completed, which gives more time to other system processes.

See Also

NiosScheduleForegroundEvent
NiosCancelForegroundEvent

NiosPopfd

Description

Sets the Eflags register equal to the contents of the passed-in parameter.

Syntax

```
#include <nios.h>

void
NiosPopfd(
    UINT32      Eflags);
```

Parameters

None

Returns

Nothing

Remarks

This is an in-line function.

See Also

NiosPushfdCli
NiosPushfd
NiosCli
NiosSti

NiosPrevChar

Description

Decrements the *CurrentPtr* by one character.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosPrevChar(
     UINT8       *StringStart,
     UINT8       *CurrentPtr);
```

Parameters

StringStart	Pointer to the beginning of the string
CurrentPtr	Pointer to the current character in the string

Returns

Pointer to previous character in string

Remarks

Generally, *StringStart* must point at the beginning of the string. However, this function will still work as long as *StringStart* does not point past *CurrentPtr*, and *StringStart* is on a character boundary. This function does not allow *CurrentPtr* to decrement below *StringStart*.

See Also

NiosPrintf

Description

A double-byte-character-aware printf function.

Syntax

```
#include <rstdlib.h>

UINT32
NiosPrintf(
    modHandle       moduleHandle,
    UINT32          msgType,
    UINT8           *formatStr,
    ...);
```

Parameters

moduleHandle	Caller's module handle
msgType	One of MT_xxxx values defined in nstdlib (.h/.inc)
formatStr	ASCIIz format string (standard printf style)
...	Variable number of arguments. This must match the number of format specifiers in formatStr. If the MTF_INDIRECT_ARGS option is used, then there must be one parameter here which points to the block of parameters to use.

Returns

Number of bytes (columns) output

FFFFFFFDh	Invalid format specifier

Remarks

The discussion below gives detailed information on using this function. The variable list of parameters given to this routine may have been reordered when this function returns.

Message Types
The defined *msgType* values are listed below:

MT_NOMSG	equ	0x00000000
MT_INFORM	equ	0x00000001
MT_INIT_FATAL	equ	0x00000002
MT_ALERT	equ	0x00000003
MT_ABEND	equ	0x00000006
MT_DEBUG_OUT	equ	0x00000008
MT_DEBUG_TRACE	equ	0x00000009

MT_NOMSG	Effectively an NOP. **NiosPrintf** ignores this message.
MT_INFORM	Used to display normal status information during an NLM's initialization routine. This message type cannot be used at interrupt time.
MT_INIT_FATAL	Used to display messages describing why a module was unable to initialize during a module's init routine. This message type cannot be used at interrupt time.
MT_ALERT	Displays messages describing abnormal events that affect the user. These messages are queued and displayed at a later time. The user must acknowledge the message before continuing.
MT_ABEND	Immediately displays the message and hangs the system. This should be used in cases where an unrecoverable event has occurred and system operation cannot continue reliably. Typically, this is used for the "never should happen" cases.
MT_DEBUG_OUT	Displays a message in the active debugger environment. If no debugger is loaded, the message is not seen.
MT_DEBUG_TRACE	Places message in NIOS's trace buffer (if active). *** Not implemented at this printing ***

Printf Usage Information
To signal a new line in a string you must use "\r\n" instead of "\n".

A format string contains text and optional format specifications. There are six fields in a format specification: some are required and others are optional.

formatString = "% [Flags] [Width] [.Precision] [Size] Type"

Flags Currently the only flag this code understands is "-". This causes the output to be left-justified.

Width Defines the number of columns the output will consume. (Double-byte characters consume two columns.) If output is less than Width, the output will be padded with blanks on the left; on the right if the "-" flag is specified.

If '*' the actual width value will be obtained as a parameter to this call. The value must be passed as a parameter immediatelly before the parameter used for the Type conversion.

.Precision

Defines the maximum number of columns the input will consume. (Double-byte characters consume two columns.)

For value conversions, if the length of input is less than *Precision*, the input is padded on the left with zeroes. If the length of input is more than *Precision*, no truncation will occur.

For value conversions, if *Precision* is explicitly set to zero (i.e., ".0") and the converted value is zero, no digits will be output.

For string conversions, if the length of input is more than *Precision*, the input will be truncated. This can cause a double-byte character to be split. This code detects this situation and substitutes a ' ' in place of the split double-byte character.

If '*' the actual precision value will be obtained as a parameter to this call. The value must be passed as a parameter immediatelly before the parameter used for the Type conversion.

Size This character defines the size of a value parameter. It does not affect types "s", "c" or "n". If *Size* is not defined, value is assumed to be a DWORD.

"h" Causes value to be treated as a BYTE. The high byte of value is ignored. However, the stack must contain a DWORD value.

"l" Causes value to be treated as a DWORD. The stack must contain a DWORD value.

"w" Causes value to be treated as a 16-bit WORD. However, the stack must contain a DWORD value.

Type This required character defines the type of formatting that must be done. This character can be one of the following:

"s" String. The stack contains a 32-bit near pointer to an ASCIIZ string.

"u" Decimal Value. The stack contains a DWORD value that will be converted to a base-10 ASCII string and then output. The value is treated as unsigned.

"x" Hex Value. The stack contains a DWORD value that will be converted to a base-16 ASCII string and then output. The value is treated as unsigned. Hex alpha characters will be lower case.

"X" Hex Value. The stack contains a DWORD value that will be converted to a base-16 ASCII string and then output. The value is treated as unsigned. Hex alpha characters will be upper case.

"c" Character. The stack contains a DWORD value which is four consecutive bytes of a string. If the low byte is the first byte of a double-byte character, both bytes are output; else only the low byte is output.

"n" Pointer to int. The stack contains a 32-bit near pointer to a DWORD variable. This variable will be set to the number of bytes (columns) that have been output so far.

Limitations
This function does not support types "d" or "i".

This is because the format for negative numbers is different in some countries than our standard "-1234". If your code needs to output a negative number, (unlikely in assembler programs) you should refer to the "C" LOCALE library "_LLOCALE.C" for details on how to do it.

This function does not support floating-point conversions.

NiosPushfd

Description

Saves and returns the current value of the Eflags register.

Syntax

```
#include <nios.h>

UINT32
NiosPushfd(
    void);
```

Parameters

None

Returns

Eflags value

Remarks

See Also

NiosPopfd
NiosPushfdCli
NiosCli
NiosSti

NiosPushfdCli

Description

Saves and returns the current value of the Eflags register and clears the interrupt flag.

Syntax

```
#include <nios.h>

UINT32
NiosPushfdCli(
    void);
```

Parameters

None

Returns

Interrupts disabled
Eflags value prior to doing a CLI

Remarks

Interrupts are disabled after obtaining the current value. This is an in-line function.

See Also

NiosPopfd
NiosPushfd
NiosCli
NiosSti

NiosQueueInit

Description

Initializes a queue for a singly linked linear list.

Syntax

```
#include <niosq.h>

void
NiosQueueInit (
      slinkQueue  *queue );
```

Parameters

queue	Queue to initialize

Returns

Nothing

Remarks

See Also

NiosRegisterHandleClient

Description	Registers handle-manager client.

Syntax

```
#include <handlmgr.h>

UINT32
NiosRegisterHandleClient (
     ModHdlP    modHandle,
     UINT32     *clientID,
     UINT32     handlesPerBlock,
     UINT32     numberOfBlocks );
```

Parameters

modHandle	Module handle of handle manager client. This will be used to track resources associated with this client.
clientID	Set by handle manager. Provides unique key that will be used by clients of handle manager to access other management functions.
handlesPerBlock	Optionally specifies allocation handles per memory block. If this is zero, the handle manager will specify default value (30).
numberOfBlocks	Optionally specifies the number of handle allocation blocks originally allocated. If this is zero, the handle manager default value (1) ANDed with (pBlock -> clientBlockQ).

Returns

SUCCESS_CODE
errOUT_OF_MEMORY
errOUT_OF_CLIENT_IDS
errINVALID_PARAMETER

Remarks

See Also

NiosListHandles, NiosDeRegisterHandleClient, NiosGetHandle
NiosFreeHandle, NiosHandletoAddress, NiosAddressToHandle

NiosRegisterStdOutHandler

Description

Registers a handler that will be invoked when an NLM calls NiosPrintf with a message type of MT_INFORM or MT_INIT_FATAL.

Syntax

```
#include <nstdlib.h>

UINT32
NiosRegisterStdOutHandler(
    modHandle      module,
    stdOutInfo  *stdOutBlock);
```

Parameters

module	Caller's module handle.
stdOutBlock	Pointer to stdOutInfo structure with the SOIHandler field set. The memory for this structure is owned by NIOS until the handler is deregistered.

The SOIHandler must return zero if it processed the message; else the message is passed to the next registered handler.

Returns

0	Handler successfully registered
!0	Error registering handler

Remarks

This handler is invoked when an NLM calls NiosPrintf with a message type of MT_INFORM or MT_INIT_FATAL in the process context that was active when this function was called. This allows different handlers to be active depending on the execution context.

See Also

NiosDeRegisterStdOutHandler

NiosScheduleAESEvent

Description

Schedules an event to occur after a specified number of milliseconds.

Syntax

```
#include <aes.h>

void
NiosScheduleAESEvent(
    modHandle    module,
    UINT32       ms,
    AESECB       *aesEcb);
```

Parameters

module	Caller's module handle.
ms	Number of milliseconds.
aesEcb	Pointer to AES ECB. *aesEcb.AESESR* must point to a valid function

Returns

Nothing

Remarks

This function is callable at interrupt time in all environments. When the timer expires, the ECB's ESR will be called. The ESR is called at foreground context, not interrupt time, and therefore has access to all system services.

This service can be called for an AES that is already scheduled. In this case, the scheduled AES is cancelled and then rescheduled for the new timeout.

While the AES is actively scheduled the AESStatus field in aesEcb will be set to a non-zero value. Prior to invoking the AESESR handler, AESStatus will be set to 0.

The accuracy of this service can vary greatly depending on the underlying OS's timing capabilities. The worst case resolution is approximately ±55ms.

If the ECB is already scheduled, it will be rescheduled for the specified timeout. The entry and exit conditions for an AESESR are:

On entry: aesEcb.AESStatus = 0
"C" AESESR: void esr(NiosAESECB *aesEcb)
asm AESESR: ESI -> aesEcb
CLD has been executed
Interrupts disabled

On return: Interrupts in any state
CLD must be preserved
All registers can be destroyed

Interrupts are in the same state as when called

See Also

NiosScheduleForegroundEvent

Description

Schedules an event that fires in a foreground context.

Syntax

```
#include <nios.h>

void
NiosScheduleForegroundEvent(
      FEB      *eventBlock);
```

Parameters

eventBlock A pointer to a Foreground Event Block (FEB). The *FEBReserved* and *FEBESR* fields are not modified by this function. The *FEBESR* field must contain a valid callback address. FEBStatus will be non-zero while the foreground event is active. Prior to invoking the FEBESR handler, FEBStatus will be set to 0.

The FEBESR routine is invoked thus:

```
assumes:   eventBlock.FEBStatus = 0
           For "C" ESRs:
                 void (*FEBESR)(FEB *eventBlock)
           For "asm" ESRs:
                 esi -> eventBlock
           CLD has been executed
           Interrupts are disabled
returns:   All registers can be destroyed
           Interrupts in any state
```

Returns

Nothing

Remarks

Events are serviced after the interrupt context is exited and as soon as control is relinquished by the currently executing Ring-0 thread, either directly or indirectly.

This function can be invoked while executing in foreground context, in which case the event will fire during the next yield (relinquish). This service should be used when a module gains control in the context of a hardware interrupt and needs to invoke functions that are not callable from an interrupt context.

This function is callable at interrupt time in all environments. The interrupt flag is preserved and never enabled by this function.

See Also

NiosCancelForegroundEvent
NiosPoll

NiosSetDateTime

Description

Sets the local machine's current date and time equal to the specified values.

Syntax

```
#include <nios.h>

UINT32
NiosSetDateTime(
    NDateTime      *dateTime);
```

Parameters

dateTime	Structure which holds new date and time values

Returns

0	Success
0xFFFFFFFF	Function cannot be performed in the current execution context
0xFFFFFFFE	Function failure

Structures

NDateTime	Structure used by NiosGetDateTime and NiosSetDateTime function

```
typedef struct NDateTimeStruc

UINT8 NDTHour;        // (0-23)
UINT8 NDTMinute;      // (0-59)
UINT8 NDTSecond;      // (0-59)
UINT8 NDTReserved;

UINT8 NDTDay;         // (1-31)
UINT8 NDTMonth;       // (1-12)
UINT16 NDTYear;       // (1980-2079)
}NDateTime;
```

Remarks

Note that this function can yield.

See Also

NiosShortTermAlloc

Description

Allocates a block of memory of the specified size for short-term use by the caller.

Syntax

```
#include <nios.h>

void
*NiosShortTermAlloc(
     modHandle      module,
     UINT32         size);
```

Parameters

module	Caller's module handle
size	Number of bytes to allocate

Returns

0	Allocation failed
!0	Valid memory pointer

Remarks

NiosShortTermAlloc is appropriate when memory is needed only for a short period of time (for example, a piece of memory that is allocated and freed in the context of the same routine). Allocating memory using the appropriate function will reduce memory fragmentation.

All memory allocated using this function is locked (that is, is non-pageable, always presents, etc.). However, it is not guaranteed to be physically contiguous.

See Also

NiosLongTermAlloc
NiosPhysContigAlloc
NiosFree

NiosSignalSemaphore

Description

Unblocks the next waiting process (if any) that was blocked with a call to **NiosWaitSemaphore**.

Syntax

```
#include <nios.h>

void
NiosSignalSemaphore(
      semHandle  handle);
```

Parameters

handle	The handle identifying the semaphore to signal

Returns

Nothing

Remarks

If called at interrupt time, this service does not cause a task switch, instead any unblock process is scheduled to run the next time the foreground yields.

See Also

NiosCreateSemaphore
NiosCreateSemaphoreEx
NiosDestroySemaphore
NiosExamineSemaphore
NiosWaitSemaphore

NiosSprintf

Description

A double-byte-character-aware *sprintf* function.

Syntax

```
#include <nstdlib.h>

UINT32
NiosSprintf(
     UINT8        *buf,
     UINT8        *FormatStr,
     ...);
```

Parameters

buf

FormatStr

Returns

Number of bytes (columns) output

FFFFFFFDh	Invalid format specifier

Remarks

The format string is passed in to this function. Refer to the "Printf Usage Information" discussion under **NiosPrintf** for detailed information on using this function.

The stack parameters given to this routine may have been reordered when this function returns.

See Also

NiosStatDeRegister

Description	Removes an entry from the registry.

Syntax

```
#include <niosstat.h>

UINT32
NiosStatDeRegister (
   modHandle              moduleHandle,
   NIOS_STAT_TABLE        *table );
```

Parameters

moduleHandle	Caller's module handle.
table	Pointer to table used in call to **NiosStatRegister**.

Returns

NIOS_STAT_SUCCESS_CODE
NIOS_STAT_NOT_REGISTERED
NIOS_STAT_INVALID_PARAMETER

Remarks

This will unregister the first occurance of <table>.

See Also

NiosStatRegister, NiosStatEnumerate,
NiosStatGetTable, NiosStatResetTable

NiosStatEnumerate

Description

Enumerates through available statistics tables.

Syntax

```
#include <niosstat.h>

UINT32
NiosStatEnumerate (
   UINT32       *search,
   UINT8     *name );
```

Parameters

search	INPUT: Last search returned by this function. 0xFFFFFFFF to start. OUTPUT: Receives new search value.
name	OUTPUT: Receives name of table, at most NIOS_STAT_MAX_NAME bytes

Returns

NIOS_STAT_SUCCESS_CODE
NIOS_STAT_NO_MORE_TABLES
NIOS_STAT_INVALID_PARAMETER

Remarks

To initiate an enumeration, pass 0xFFFFFFFF as the search value. This function will enumerate through LSL, protocol stack and MLID tables if they exist.

Example:

```
UINT32  ccode;
UINT32  search = 0xFFFFFFFF;

for ( ccode = NStatEnumerate ( &search, name );
         ccode == NIOS_STAT_SUCCESS_CODE;
         ccode = NStatEnumerate ( &search, name ) ) {

   NiosPrintf ( NlmHandle, MT_INFORM, "Table: %s\n", name );
}
```

See Also

NiosStatRegister,
NiosStatDeRegister,

NiosStatGetTable,
NiosStatResetTable

NiosStatGetTable

Description

Retrieves a specific statistics table in condensed form.

Syntax

```
#include <niosstat.h>

UINT32
NiosStatGetTable (
   UINT32        options,
   UINT8         *description,
   UINT32        bufferSize,
   UINT8         *buffer,
   UINT32        *actualSize);
```

Parameters

options	INPUT: see NIOS_STAT_GET_OPTION_? bit flags in the beginning of this chapter Unused bits must be zero.
description	INPUT: ASCIIZ string used to register the table (case insensitive).
buffersize	INPUT: Size of buffer in bytes.
buffer	OUTPUT: Receives statistics information in "condensed" form.
actualsize	OUTPUT: Receives total size in bytes of available statistics information.

Returns

NIOS_STAT_SUCCESS_CODE

NIOS_STAT_NOT_REGISTERED
The table was not found.

NIOS_STAT_BUFFER_TOO_SMALL
BufferSize < size of available data.

NIOS_STAT_INVALID_PARAMETER

Remarks

The description string is a well known string that is used by applications to register the table.

This function will use the description string to perform a case insensitive search for the desired table.

To retrieve the size necessary to store the available statistics information, pass 0 for bufferSize (buffer will be ignored).

The NIOS_STAT_GET_OPTION_REFRESH option will update only the counters in the table instead of the entire table, description strings, and so forth. It assumes that buffer is unchanged from the previous call.

Example:

```
UINT8   *buffer;
UINT32   actualSize;

NiosStatGetTable ( 0, "Wamco Statistics Table", 0, NULL, &actualSize );
buffer = ( UINT8 * ) NiosShortTermAlloc ( NlmHandle, actualSize );
for ( NiosStatGetTable ( 0,
            "Wamco Statistics Table",
             buffer,
             &actualSize );
      AD_INFINITUM;
      NiosStatGetTable ( NIOS_STAT_GET_OPTION_REFRESH,
             "Wamco Statistics Table",
              buffer,
              &actualSize ) ) {
      Do stuff with buffer
      Relinquish control for an interval
}

NiosFree ( buffer );
```

See Also

NiosStatRegister, NiosStatDeRegister, NiosStatEnumerate, NiosStatResetTable

NiosStatRegister

Description

Creates an entry in the registry.

Syntax

```
#include <niosstat.h>

UINT32
NiosStatRegister (
   modHandle              moduleHandle,
   NIOS_STAT_TABLE        *table );
```

Parameters

moduleHandle	Caller's module handle.
table	Statistics table.

Returns

NIOS_STAT_SUCCESS_CODE
NIOS_STAT_OUT_OF_CLIENT_MEMORY
NIOS_STAT_INVALID_PARAMETER

Remarks

The description string in <table> is a well known string that can be used by applications to locate a specific stat table. Also, the memory for the table and all its subcomponents must be long lived, that is, the registry maintains the pointer to the table and not a copy of the table itself.

All strings in the table should be language enabled.

Once a table has registered, it must not change in size. If a table changes in size while it is registered, registry behavior is undefined.

Example:

```
typedef struct _wamcoUntyped {
   UINT32 length;
   UINT8  Bob;
} WAMCO_UNTYPED;


   define stats
UINT32        WamcoStat1   = 1;
UINT64        WamcoStat2   = { 2, 0 };
UINT8         WamcoStat3[] = "Bob";
```

```
UINT32        WamcoStat4   = 4;
WAMCO_UNTYPED WamcoStat5   = { 1, 42 };


NIOS_STAT_ENTRY   WamcoStats [] = {
 {NIOS_STAT_UINT32 | NIOS_STAT_RESETTABLE, 1, &WamcoStat1, "UINT32 stat" },
 {NIOS_STAT_UINT64,                        2, &WamcoStat2, "UINT64 stat" },
 {NIOS_STAT_ASCIIZ,                        3, WamcoStat3, "ASCIIZ stat"  },
 {NIOS_STAT_NOT_USED,                      4, &WamcoStat4, "UNUSED stat" },
 {NIOS_STAT_UNTYPED,                       5, &WamcoStat5, "UNTYPED stat" }
};

NIOS_STAT_TABLE   WamcoTable = {
  0,
  "Wamco Statistics Table",
  "Wamco Statistics Table: Another Quality Wamco Product",
  { 0, 0, 0 },
  NIOS_STAT_TABLE_HAS_RESETTABLE,
  sizeof WamcoStats / sizeof ( NIOS_STAT_ENTRY ),
  WamcoStats
};

NiosStatRegister ( mh, &WamcoTable );
```

See Also NiosStatDeRegister, NiosStatEnumerate, NiosStatGetTable, NiosStatResetTable

NiosStatResetTable

Description

Sets all UINT32 and UINT64 counters to zero for the requested table.

Syntax

```
#include <niosstat.h>

UINT32
NiosStatResetTable (
  UINT8      *description);
```

Parameters

description	ASCIIZ string used to register the table (case insensitive)

Returns

NIOS_STAT_SUCCESS_CODE

NIOS_STAT_NOT_REGISTERED
The requested table has not registered.

NIOS_STAT_READ_ONLY
The module that registered the table does not allow its counters to be reset

NIOS_STAT_INVALID_PARAMETER

Remarks

The description string is a well known string that is used by applications to register the table.

This function will use the description string to perform a case insensitive search for the desired table.

The protocol stack and MLID tables are not registered but are also read-only, so they return NIOS_STAT_NOT_REGISTERED.

See Also

NiosStatRegister, NiosStatDeRegister, NiosStatEnumerate, NiosStatGetTable

NiosSti

Description

NiosSti executes an STI instruction that enables interrupts.

Syntax

```
#include <nios.h>

void
NiosSti(
      void);
```

Parameters

None

Returns

Nothing

Remarks

This is an in-line function.

See Also

NiosPushfdCli
NiosPushfd
NiosCli
NiosPopfd

NiosStrCat

Description

Appends the contents of *srcStr* to *destStr*.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosStrCat(
     UINT8       *destStr,
     UINT8       *srcStr);
```

Parameters

destStr	Pointer to destination string
scrStr	Pointer to source string

Returns

Source string is appended to destination string.

Remarks

destStr must be large enough to hold the resultant string.

See Also

NiosStrChr

Description

Scans a string for the first occurrence of a given character.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosStrChr(
    UINT8       *str,
    UINT8       chr);
```

Parameters

Returns

zero	*chr* does not occur in *str*
Pointer to the first occurrence of the character	

Remarks

See Also

NiosStrCmp

Description

Performs a case-sensitive string compare.

Syntax

```
#include <nstdlib.h>

UINT32
NiosStrCmp(
     UINT8       *string1,
     UINT8       *string2);
```

Parameters

string1	Points to source string
string2	Points to destination string

Returns

0	Strings are the same
!0	Strings are different

Remarks

See Also

NiosStrCmpi

Description

Performs a case-insensitive string comparison.

Syntax

```
#include <nstdlib.h>

UINT32
NiosStrCmpi(
    UINT8      *string1,
    UINT8      *string2);
```

Parameters

string1	Points to source string
string2	Points to destination string

Returns

0	Strings are the same
!0	Strings are different

Remarks

See Also

NiosStrCpy

Description

Copies the contents of one ASCIIZ string to another.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosStrCpy(
     UINT8        *destStr,
     UINT8        *srcStr);
```

Parameters

Returns

destStr unmodified

Remarks

destStr must be large enough to hold the contents of *srcStr*.

See Also

NiosStrLen

Description

Returns the number of bytes in *String*, not counting the NULL termination byte.

Syntax

```
#include <nstdlib.h>

UINT32
NiosStrLen(
    UINT8        *String);
```

Parameters

String	Pointer to a string

Returns

Number of bytes in string

Remarks

Double-byte characters count as two bytes.

See Also

NiosStrLwr

Description

Converts all uppercase characters in the specified string to lowercase.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosStrLwr(
    UINT8       *str);
```

Parameters

str	Pointer to string

Returns

Changes made to string

Remarks

Note that this function does not covert uppercase characters with values above 127.

See Also

NiosStrUpr
NiosToUpper
NiosToLower

NiosStrnCmp

Description

Case-sensitive string compare with maximum.

Syntax

```
#include <nstdlib.h>

UINT32
NiosStrnCmp (
      UINT8       *string1,
      UINT8       *string2,
      UINT32      maxLen);
```

Parameters

string1	Source string
string2	Destination string
maxLen	Maximum number of characters to compare

Returns

zero	Strings are the same
non-zero	Strings are different

Remarks

Compares two strings until either the NULL terminator is encountered, *maxLen* characters have been compared, or the strings mismatch.

If *maxLen* is zero, this function will return zero.

See Also

NiosStrnCmpi

Description

Case-insensitive string compare with maximum.

Syntax

```
#include <nstdlib.h>

UINT32
NiosStrnCmpi (
    UINT8       *string1,
    UINT8       *string2,
    UINT32      maxLen);
```

Parameters

string1	Source string
string2	Destination string
maxLen	Maximum number of characters to compare

Returns

zero	Strings are the same
non-zero	Strings are different

Remarks

Compares two strings until either the NULL terminator is encountered, maxLen characters have been compared, or the strings mismatch.

If maxLen is zero, this function will return zero.

See Also

NiosStrtoByteArray

Description

This function converts the specified ASCIIZ numeric string into a byte array.

Syntax

```
#include <nstdlib.h>

UINT32 NiosStrtoByteArray (
    UINT8     *String,
    UINT8     **StringStop,
    UINT8     *Dst,
    UINT32    DstSize,
    UINT32    Flags);
```

Parameters

String	ASCIIZ hex numeric string to convert into a byte array.
StringStop	Character that halted the conversion process. To be filled in on return, if not NULL.
Dst	Array of bytes to be filled in by the conversion.
DstSize	The length of the Dst in bytes.
Flags	Bitmap of options to do during conversion: NSTBA_RIGHT_JUSTIFY specifies that the array should be justified to the right. All undefined bits must be set to zero. NSTBA_RIGHT_JUSTIFY is defined as 0x00000001

Returns

Number of bytes in Dst which were converted
Dst has been filled with converted byte array
**StringStop* updated as described above

Remarks

The base of the numeric string is always 16. If *StringStop* is not NULL the variable pointed to by *StringStop* will be updated to point at the character that caused the conversion to stop.

The string may have white space (01h-20h & ',') before the numeric characters. Values will be converted until a non-numeric character is encountered. By default, the converted byte array is left-justified.

See Also

NiosStrtoul

Description

Converts the specified ASCIIZ numeric string into a UINT32 value.

Syntax

```
#include <nstdlib.h>

UINT32
NiosStrtoul
    (UINT8      *String,
    UINT8       **StringStop,
    UINT32      Radix);
```

Parameters

Radix	Specifies the base of the numeric string. This value must be between 2 and 36 inclusive.
StringStop	If not NULL, the variable pointed to by *StringStop* will be updated to point at the character that caused the conversion to stop.
String	Points to string to be converted. Allows white space (01h-20h and/or ',') before the numeric characters.

Returns

Converted value
**StringStop* updated, as described below

Remarks

If the numerical string begins with "0x", the *radix* will be forced to base 16 (hex). This function does not handle negative numbers.

Note: The format for negative numbers is different in some countries from the "-1234" used in the U.S. If your code needs to output a negative number (unlikely in assembler programs), you should refer to the "C" LOCALE library "_LLOCALE.C" for instruction.

Values will be converted until a non-numeric character is encountered. If the converted value is larger than 2^{32-1} or there were no characters to convert, EAX will be set to zero and **StringStop* will equal *String*. This allows the caller to detect an empty string or an overflow condition.

See Also

NiosStrUpr

Description

Converts all lowercase characters in the specified string to uppercase.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosStrUpr(
    UINT8        *str);
```

Parameters

str	Pointer to string

Returns

Changes made to string

Remarks

See Also

NiosStrLwr
NiosToUpper
NiosToLower

NiosTestCharBoundary

Description

Determines if the current string position is positioned on the second byte of a double-byte character.

Syntax

```
#include <nstdlib.h>

UINT32
NiosTestCharBoundary(
    UINT8      *StringStart,
    UINT8      *CurrentPtr);
```

Parameters

StringStart	Pointer to the beginning of the string
CurrentPtr	Pointer to the current character in the string

Returns

00000000h	*CurrentPtr* is on a character boundary
Nonzero	*CurrentPtr* is on the second byte of DBC

Remarks

See Also

NiosThreadArmId

Description

Initializes for a subsequent call to **NiosThreadBlockOnId** using the specified id parameter.

Syntax

```
#include <nios.h>

void
*NiosThreadArmId(
     UINT32      id);
```

Parameters

id	Specifies a unique value which is used to unblock the thread. This typically should be the address of some memory object owned by the caller. This method ensures that the id value is unique and not used by another module in the system. Note that only one thread can use the id at a time.

Returns

zero	Out of memory
non-zero	Context handle

Remarks

To avoid a deadlock condition, this function must be invoked prior to allowing any code to run which would signal the id using the **NiosThreadSignalId** service. Prior to starting an event or operation which will use the **NiosThreadSignalId** service, the caller should invoke this service to arm the id and obtain a context handle which can be used to call the **NiosThreadBlockOnId** service.

The **NiosThreadBlockOnI**d service is typically invoked after one or more asynchronous operations have been started. The asynchronous event would then use **NiosThreadSignalId** to unblock the thread.

See Also

NiosThreadBlockOnId
NiosThreadSignalId

NiosThreadBlockOnId

Description

Blocks the currently running thread of execution until the specified id is signalled using the **NiosThreadSignalId** service.

Syntax

```
#include <nios.h>

UINT32
NiosThreadBlockOnId(
      void          *contextHandle
      UINT32        reserved);
```

Parameters

contextHandle	Handle obtained using the **NiosThreadArmId** service.
reserved	Must be set to zero. Reserved for future use.

Returns

0	id was signalled before block attempt
1	id was signalled while blocked
2	Reserved for future use

Remarks

Use of this service to yield is preferred over use of the **NiosPoll** service because this service removes the thread from the kernel's run queue, therefore providing more execution time to other threads in the system. The **NiosPoll** service, on the other hand, allows the thread to still receive time.

See Also

NiosThreadArmId
NiosThreadSignalId

NiosThreadSignalId

Description

Unblocks the thread currently blocked on the specified id.

Syntax

```
#include <nios.h>

void
NiosThreadSignalId(
      UINT32      id);
```

Parameters

id	Id value used during call to **NiosThreadArmID**

Returns

Nothing. Interrupt state preserved and NOT enabled.

Remarks

If called from interrupt context, this function does not immediately yield. Instead the thread is unblocked and run at the earliest opportunity.

This function is callable at interrupt time in all environments.

See Also

NiosThreadArmId
NiosThreadBlockOnId

NiosToLower

Description	Converts an uppercase character to lowercase.

Syntax

```
#include <nstdlib.h>

UINT8
NiosToLower(
    UINT8        chr);
```

Parameters

Returns

chr lowercase if it was uppercase
else
chr is returned unmodified

Remarks

If the character is not uppercase then it is returned unmodified.

Note that this function does not convert uppercase characters with values above 127.

See Also

NiosToUpper
NiosStrUpr
NiosStrLwr

NiosToUpper

Description

Converts a lowercase character to uppercase.

Syntax

```
#include <nstdlib.h>

UINT8
NiosToUpper(
    UINT8       chr);
```

Parameters

Returns

chr uppercase if it was lowercase
else
chr is returned unmodified

Remarks

If the character is not lowercase, it is returned unmodified.

See Also

NiosToLower
NiosStrUpr
NiosStrLwr

NiosUltoa

Description

Converts the specified value to a displayable ASCIIZ string of the specified *radix*.

Syntax

```
#include <nstdlib.h>

UINT8
*NiosUltoa(
    UINT32      Value,
    UINT8       *StringBuf,
    UINT32      Radix);
```

Parameters

Returns

Pointer to *StringBuf*

Remarks

Radix must be between 2 and 36, inclusive.

StringBuf must be large enough to hold the resultant string. The largest *StringBuf* ever needs to be is 33 bytes. (Necessary to convert a UINT32 to binary (Radix 2). This conversion requires 32 bytes plus 1 byte for the zero terminator (assuming the value did not contain any leading zeros)).

This function treats <*Value*> as unsigned.

Note: The format for negative numbers is different in some countries than the "-1234" used in the US. If your code needs to output a negative number (unlikely in assembler programs), you should refer to the "C" LOCALE library "_LLOCALE.C" for instruction. The returned string will contain no leading zeros.

See Also

NiosUnHookExportedApi

Description

Removes a previously defined export.

Syntax

```
#include <module.h>

UINT32
NiosUnHookExportedApi(
      modHandle      moduleHandle,
      UINT32         hardIntNumber,
      void           (*interruptProc)(void),
      UINT32         options,
      UINT32         refData );
```

Parameters

moduleHandle Pointer to the caller's module structure.

hardIntNumber IRQ to hook (0-15)

interruptProc Pointer to routine that is invoked when the specified hardware interrupt occurs. Entry and exit conditions are as follows if the procedure is an assembly handler, in which case the HIOPT_C_HANDLER option is NOT specified:

```
On entry:   EDX    refData
            Interrupts are disabled.
            CLD has been executed.

On exit:    Interrupts are disabled.
            All registers can be destroyed.
            CLD preserved.

            Z flag set if int was serviced, else pass
            int to next handler.
```

Entry and exit conditions are as follows if the procedure is a "C" language handler, in which case the HIOPT_C_HANDER_BIT option must be specified.

```
On entry:   UINT32 (*interruptProc)(void *refData)
            Interrupts are disabled.
            CLD has been executed.
```

On exit: 0 Interrupt was serviced.
!0 Pass interrupt to next handler.
Interrupts are disabled.
"C" registers preserved.

options Possible values are:

HIOPT_SHAREABLE_BIT
Interrupt can be shared with others that support sharing.

HIOPT_C_HANDER_BIT
interruptProc is invoked with "C" compatible input and exit conditions.

refData Reference data passed to *interruptProc* in register EDX.

Returns

0	Export was successfully hooked
0xFFFFFFFF	The specified interrupt is already hooked by a handler that doesnot support shareable interrupts.
0xFFFFFFFE	The specified interrupt is already hook by a handler that requires other handlers to support sharing.
0xFFFFFFFD	Specified IRQ value is above 15.
0xFFFFFFFC	Not enough free memory to hook the interrupt.

Remarks

See Also

NiosHookExportedApi, NiosUnHookHardwareInt, DoEndOfInterrupt, EnableHardwareInterrupt, DisableHardwareInterrupt, CheckHardwareInterrupt

NiosUnHookHardwareInt

Description

Unhooks the caller from the specified hardware interrupt chain.

Syntax

```
#include <nios.h>

UINT32
NiosUnHookHardwareInt(
    modHandle     moduleHandle,
    UINT32        hardIntToUnHook,
    void          (*interruptProc)(void));
```

Parameters

moduleHandle	Pointer to caller's module structure
hardIntToUnHook	IRQ to unhook from (0-15)
interruptProc	A pointer to the caller's interrupt handler routine

Returns

0	Caller was unhooked successfully
0xFFFFFFFF	Caller was not hooking the interrupt

Remarks

See Also

NiosHookHardwareInt

NiosUnlinkFirst

Description

Unlinks the first queue entry from a singly linked queue.

Syntax

```
#include <niosq.h>

void
NiosUnlinkFirst (
      slinkQueue  *queue );
```

Parameters

queue	Queue to remove node from

Returns

Pointer to removed node
Zero if queue is empty

Remarks

Queue nodes must include a forward link field. The offset to this field must be provided to the queueing routine.

See Also

NiosFindNode
NiosLinkNext
NiosLinkFirst
NiosNextNode
NiosUnlinkAfter
NiosUnlinkNode

NiosUnlinkNext

Description

Removes a node into the front of a singly linked list.

Syntax

```
#include <niosq.h>

void
NiosUnlinkNext (
      void              *node,
      slinkQueue  *queue );
```

Parameters

node	Pointer to node to remove link after
queue	Queue in which the node is contained

Returns

Pointer to removed node

Remarks

The list is assumed to be null terminated. The queue structure passed points to the head and tail nodes of the linear linked list.

Queue nodes must include a forward link field. The offset to this field must be provided to the queueing routine.

This function always returns a pointer to the removed entry.

See Also

NiosFindNode
NiosLinkNext
NiosLinkFirst
NiosNextNode
NiosUnlinkFirst
NiosUnlinkNode

NiosUnlinkNode

Description

Removes input parameter node from the specified queue.

Syntax

```
#include <niosq.h>

void
NiosUnlinkNode (
    void            *node,
    slinkQueue  *queue );
```

Parameters

node	Pointer to node that needs to be unlinked
queue	Queue to search for link

Returns

Pointer to removed node
Zero if node is not found in queue

Remarks

Queue nodes must include a forward link field. The offset to this field must be provided to the queueing routine.

See Also

NiosFindNode
NiosLinkNext
NiosLinkFirst
NiosNextNode
NiosUnlinkAfter
NiosUnlinkFirst

NiosUnloadModule

Description

Removes an NLM module from memory.

Syntax

```
#include <module.h>

UINT32
NiosUnloadModule(
     modHandle        modHandle,
     UINT32           unloadOptions);
```

Parameters

modHandle Module to unload

unloadOptions Bits defining unload options.

UOPTION_ERROR_MSGS
Stdout error messages are enabled. If not specified, unload error messages will be displayed as alerts.

Returns

UNLOAD_SUCCESS	Module was unloaded successfully
UNLOAD_MODULE_FORBIDS_UNLOAD	Module does not allow unload
UNLOAD_MODULE_BEING_REFERENCED	Another module is using this module
UNLOAD_INVALID_MODULE_HANDLE	Module handle is invalid
UNLOAD_RESOURCES_NOT_FREED	Module did not free resources, however the module was still removed
UNLOAD_MODULE_CANT_UNLOAD_NOW	Module is temporarily unable to unload
UNLOAD_UNLOAD_REFUSED	A loaded NLM refuses to allow this NLM to unload

Remarks

See Also

NiosUnloadSelf

NiosUnloadSelf

Description

Allows an NLM to unload itself.

Syntax

```
#include <module.h>

void
NiosUnloadSelf(
        modHandle    modHand);
```

Parameters

modHand	Handle of module to unload.

Returns

Nothing.

Remarks

This service schedules a timed event that actually performs the unload operation. The specified NLM module must not yield either directly or indirectly after invoking this service. An NLM should invoke this service and return from any of its code without yielding.

This function is callable at interrupt time.

See Also

NiosUnloadModule

NiosValidateModuleHandle

Description

Verifies that the specified module handle is valid.

Syntax

```
#include <module.h>

UINT32
NiosValidateModuleHandle(
    modHandle       modHandle);
```

Parameters

modHandle	Module handle to validate

Returns

VALIDATE_SUCCESS
Module handle is okay

VALIDATE_INVALID
Module handle is bad

Remarks

See Also

NiosVidCreateDialogBox

Description	Creates a modeless dialog box (status box).

Syntax

```
UINT32
NiosVidCreateDialogBox (
    UINT8       *title,
    UINT8       *prompt );
```

Parameters

title	Pointer to ASCIIZ title string (MAX_STR_LEN length)
prompt	Pointer to ASCIIZ prompt string (MAX_STR_LEN length)

Returns

0	Failure
Non-zero	Success, handle for use in other routines

Remarks	Output only. No user input is allowed.

See Also

NiosVidUpdateDialogBox
NiosVidDestroyDialogBox

NiosVidDestroyDialogBox

Description

Destroys the previously created dialog box referenced by the handle parameter.

Syntax

```
UINT32
NiosVidDestroyDialogBox (
      void       *handle );
```

Parameters

handle	Handle returned from **NiosVidCreateDialogBox**

Returns

0	Successful
0xFFFFFFFF	Failure

Remarks

See Also

NiosVidCreateDialogBox
NiosVidUpdateDialogBox

NiosVidInputDialogBox

Description

Displays an input dialog and handles the user input.

Syntax

```
UINT32
NiosVidInputDialogBox (
    UINT8      *title,
    UINT8      *prompt,
    UINT8      *input,
    UINT32     length,
    UINT32     flags );
```

Parameters

title	Pointer to ASCIIZ title string (MAX_STR_LEN length)
prompt	Pointer to ASCIIZ prompt string (MAX_STR_LEN length)
input	Pointer to ASCIIZ input string (MAX_STR_LEN length)
length	Maximum length for input string (< MAX_STR_LEN)
flags	Control flags constructed by ORing the following options: 0x01 || Password (hide the input string)

Returns

IDOK
IDCANCEL
0xFFFFFFFF   Error displaying the input dialog box

Remarks

See Also

NiosVidMessageBox

NiosVidMessageBox

Description

Displays a message box and handles the user input.

Syntax

```
UINT32
NiosVidInputDialogBox (
      UINT8       *title,
      UINT8       *prompt,
      UINT32      buttons );
```

Parameters

title	Pointer to ASCIIZ title string (MAX_STR_LEN length)
prompt	Pointer to ASCIIZ prompt string (MAX_STR_LEN length)
buttons	Button selection list as defined by MS Windows

Returns

Determined by button selection list

0xFFFFFFFF	Error displaying the message box

Remarks

The caller can specify a title, prompt, and button (response).

See Also

NiosVidInputDialogBox

NiosVidUpdateDialogBox

Description

Updates the title and the prompt of the status dialog.

Syntax

```
UINT32
NiosVidUpdateDialogBox (
    void      *handle,
    UINT8     *title,
    UINT8     *prompt );
```

Parameters

handle	Handle returned from **NiosVidCreateDialogBox**
title	Pointer to ASCIIZ title string (MAX_STR_LEN = 0 if this parameter is to be ignored)
prompt	Pointer to ASCIIZ prompt string (MAX_STR_LEN = 0 if this parameter is to be ignored)

Returns

0	Successful
0xFFFFFFFF	Failure

Remarks

See Also

NiosVidCreateDialogBox
NiosVidDestroyDialogBox

NiosWaitSemaphore

Description

Blocks the currently running process if the semaphore token count goes negative.

Syntax

```
#include <nios.h>

void
NiosWaitSemaphore(
    semHandle  handle);
```

Parameters

handle	The handle that identifies the semaphore

Returns

Nothing

Remarks

The process will be unblocked and subsequently run when the semaphore is signaled the appropriate number of times using the **NiosSignalSemaphore()** service.

This service can be called multiple times using the same semHandle in the context of the same process. In this case, when the token count goes negative, the process will be blocked.

See Also

NiosCreateSemaphore
NiosCreateSemaphoreEx
NiosDestroySemaphore
NiosExamineSemaphore
NiosSignalSemaphore