This chapter provides an alphabetically ordered list of all the functions in the hardware abstraction layer (HAL) application program interface (API). Each function is listed with its C prototype and a short description. Each listing provides information about whether the function is thread-safe when running in a multi-threaded environment, and whether it can be called from an interrupt service routine (ISR).

This chapter only lists the functionality provided by the HAL. The complete newlib API is also available from within HAL systems. For example, newlib provides `printf()`, and other standard I/O functions, which are not described here.

☞ Each function description lists the C header file that your code must include to access the function. Because header files include other header files, the function prototype might not be defined in the listed header file. However, you must include the listed header file in order to include all definitions on which the function depends.

👣 For more details about the newlib API, refer to the newlib documentation. On the Windows **Start** menu, click **Programs** > **Altera** > **Nios II** *<version>* > **Nios II** *<version>* **Documentation**.

This chapter contains the following sections:

- "HAL API Functions" on page 14–1
- "HAL Standard Types" on page 14–79

HAL API Functions

The HAL API functions are shown on the following pages.

_exit()

Prototype:	`void _exit (int exit_code)`
Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<unistd.h>`
Description:	The newlib `exit()` function calls the `_exit()` function to terminate the current process. Typically, `exit()` calls this function when `main()` completes. Because there is only a single process in HAL systems, the HAL implementation blocks forever.
	Interrupts are not disabled, so ISRs continue to execute.
	The input argument, `exit_code`, is ignored.
Return:	–
See also:	newlib documentation

_rename()

Prototype:	`int _rename(char *existing, char* new)`
Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<stdio.h>`
Description:	The `_rename()` function is provided for newlib compatibility.
Return:	It always fails with return code −1, and with `errno` set to `ENOSYS`.
See also:	newlib documentation

alt_alarm_start()

Prototype:	```int alt_alarm_start```
	 ``` (alt_alarm* alarm,```
	 ``` alt_u32 nticks,```
	 ``` alt_u32 (*callback) (void* context),```
	 ``` void* context)```

Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_alarm.h>`
Description:	The `alt_alarm_start()` function schedules an alarm callback. Refer to "Using Timer Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. The HAL waits `nticks` system clock ticks before calling the `callback()` function. When the HAL calls `callback()`, it passes it the input argument context. The `alarm` argument is a pointer to a structure that represents this alarm. You must create it, and it must have a lifetime that is at least as long as that of the alarm. However, you are not responsible for initializing the contents of the structure pointed to by `alarm`. This action is done by the call to `alt_alarm_start()`.
Return:	The return value for `alt_alarm_start()` is zero on success, and negative otherwise. This function fails if there is no system clock available.
See also:	alt_alarm_stop() alt_nticks() alt_sysclk_init() alt_tick() alt_ticks_per_second() gettimeofday() settimeofday() times() usleep()

alt_alarm_stop()

Prototype:	`void alt_alarm_stop (alt_alarm* alarm)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_alarm.h>`
Description:	You can call the `alt_alarm_stop()` function to cancel an alarm previously registered by a call to `alt_alarm_start()`. The input argument is a pointer to the alarm structure in the previous call to `alt_alarm_start()`.
	On return the alarm is canceled, if it is still active.
Return:	–
See also:	alt_alarm_start()
	alt_nticks()
	alt_sysclk_init()
	alt_tick()
	alt_ticks_per_second()
	gettimeofday()
	settimeofday()
	times()
	usleep()

alt_dcache_flush()

Prototype:	`void alt_dcache_flush (void* start, alt_u32 len)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_cache.h>`
Description:	The `alt_dcache_flush()` function flushes the data cache for a memory region of length len bytes, starting at address start. Flushing the cache consists of writing back dirty data and then invalidating the cache.
	In processors without data caches, it has no effect.
Return:	–
See also:	alt_dcache_flush_all()
	alt_icache_flush()
	alt_icache_flush_all()
	alt_remap_cached()
	alt_remap_uncached()
	alt_uncached_free()
	alt_uncached_malloc()

alt_dcache_flush_all()

Prototype:	`void alt_dcache_flush_all (void)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_cache.h>`
Description:	The `alt_dcache_flush_all()` function flushes, that is, writes back dirty data and then invalidates, the entire contents of the data cache.
	In processors without data caches, it has no effect.
Return:	–
See also:	alt_dcache_flush()
	alt_icache_flush()
	alt_icache_flush_all()
	alt_remap_cached()
	alt_remap_uncached()
	alt_uncached_free()
	alt_uncached_malloc()

alt_dev_reg()

Prototype:	`int alt_dev_reg(alt_dev* dev)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_dev.h>`
Description:	The `alt_dev_reg()` function registers a device with the system. After it is registered, you can access a device using the standard I/O functions. Refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

The system behavior is undefined in the event that a device is registered with a name that conflicts with an existing device or file system.

The `alt_dev_reg()` function is not thread-safe in the sense that no other thread can use the device list at the time that `alt_dev_reg()` is called. Call `alt_dev_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return:	The return value is zero upon success. A negative return value indicates failure.
See also:	alt_fs_reg()

alt_dma_rxchan_close()

Prototype:	`int alt_dma_rxchan_close (alt_dma_rxchan rxchan)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_rxchan_close()` function notifies the system that the application has finished using the direct memory access (DMA) receive channel, `rxchan`. The current implementation always succeeds.
Return:	The return value is zero on success and negative otherwise.
See also:	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

alt_dma_rxchan_depth()

Prototype:	`alt_u32 alt_dma_rxchan_depth(alt_dma_rxchan dma)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_rxchan_depth()` function returns the maximum number of receive requests that can be posted to the specified DMA transmit channel, `dma`.
	Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.
Return:	Returns the maximum number of receive requests that can be posted.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

alt_dma_rxchan_ioctl()

Prototype:	`int alt_dma_rxchan_ioctl (alt_dma_rxchan dma,`
	`                            int            req,`
	`                            void*          arg)`

Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	See description.
Available from ISR:	See description.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_rxchan_ioctl()` function performs DMA I/O operations on the DMA receive channel, dma. The I/O operations are device specific. For example, some DMA drivers support options to control the width of the transfer operations. The input argument, req, is an enumeration of the requested operation; `arg` is an additional argument for the request. The interpretation of `arg` is request dependent.
	Table 14–1 shows generic requests defined in **alt_dma.h**, which a DMA device might support.
	Whether a call to `alt_dma_rxchan_ioctl()` is thread-safe, or can be called from an ISR, is device dependent. In general it safest to assume that it is not thread-safe.
	Do not call the `alt_dma_rxchan_ioctl()` function while DMA transfers are pending, or unpredictable behavior could result.
	For device-specific information about the Altera® DMA controller core, refer to the *DMA Controller Core* chapter in the *Embedded Peripherals IP User Guide*.
Return:	A negative return value indicates failure. The interpretation of nonnegative return values is request specific.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

Table 14–1. Generic Requests

Request	Meaning
ALT_DMA_SET_MODE_8	Transfer data in units of 8 bits. The value of `arg` is ignored.
ALT_DMA_SET_MODE_16	Transfer data in units of 16 bits. The value of `arg` is ignored.
ALT_DMA_SET_MODE_32	Transfer data in units of 32 bits. The value of `arg` is ignored.
ALT_DMA_SET_MODE_64	Transfer data in units of 64 bits. The value of `arg` is ignored.
ALT_DMA_SET_MODE_128	Transfer data in units of 128 bits. The value of `arg` is ignored.
ALT_DMA_GET_MODE	Return the transfer width. The value of `arg` is ignored.

Table 14–1. Generic Requests

Request	Meaning
ALT_DMA_TX_ONLY_ON	The ALT_DMA_TX_ONLY_ON request causes a DMA channel to operate in a mode in which only the transmitter is under software control. The other side writes continuously from a single location. The address to which to write is the argument to this request.
ALT_DMA_TX_ONLY_OFF	Return to the default mode, in which both the receive and transmit sides of the DMA can be under software control.
ALT_DMA_RX_ONLY_ON	The ALT_DMA_RX_ONLY_ON request causes a DMA channel to operate in a mode in which only the receiver is under software control. The other side reads continuously from a single location. The address to read is the argument to this request.
ALT_DMA_RX_ONLY_OFF	Return to the default mode, in which both the receive and transmit sides of the DMA can be under software control.

alt_dma_rxchan_open()

Prototype:	`alt_dma_rxchan alt_dma_rxchan_open (const char* name)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_rxchan_open()` function obtains an `alt_dma_rxchan` descriptor for a DMA receive channel. The input argument, name, is the name of the associated physical device, for example, `/dev/dma_0`.
Return:	The return value is null on failure and non-null otherwise. If an error occurs, `errno` is set to `ENODEV`.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

alt_dma_rxchan_prepare()

Prototype:	`int alt_dma_rxchan_prepare (alt_dma_rxchan dma,`
	`                            void*          data,`
	`                            alt_u32        length,`
	`                            alt_rxchan_done* done,`
	`                            void*          handle)`

Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	See description.
Available from ISR:	See description.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_rxchan_prepare()` posts a receive request to a DMA receive channel. The input arguments are: `dma`, the channel to use; `data`, a pointer to the location that data is to be received to; `length`, the maximum length of the data to receive in bytes; `done`, callback function that is called after the data is received; `handle`, an opaque value passed to `done`.
	Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.
Return:	The return value is zero upon success. A negative return value indicates that the request cannot be posted.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

alt_dma_rxchan_reg()

Prototype:	`int alt_dma_rxchan_reg (alt_dma_rxchan_dev* dev)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_dma_dev.h>`
Description:	The `alt_dma_rxchan_reg()` function registers a DMA receive channel with the system. After it is registered, a device can be accessed using the functions described in "Using DMA Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

System behavior is undefined in the event that a channel is registered with a name that conflicts with an existing channel.

The `alt_dma_rxchan_reg()` function is not thread-safe if other threads are using the channel list at the time that `alt_dma_rxchan_reg()` is called. Call `alt_dma_rxchan_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return:	The return value is zero upon success. A negative return value indicates failure.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

alt_dma_txchan_close()

Prototype:	`int alt_dma_txchan_close (alt_dma_txchan txchan)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_txchan_close` function notifies the system that the application has finished using the DMA transmit channel, `txchan`. The current implementation always succeeds.
Return:	The return value is zero on success and negative otherwise.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

alt_dma_txchan_ioctl()

Prototype:	`int alt_dma_txchan_ioctl (alt_dma_txchan dma,`
	`                          int             req,`
	`                          void*           arg)`

Commonly called by:	C/C++ programs Device drivers
Thread-safe:	See description.
Available from ISR:	See description.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_txchan_ioctl()` function performs device specific I/O operations on the DMA transmit channel, `dma`. For example, some drivers support options to control the width of the transfer operations. The input argument, `req`, is an enumeration of the requested operation; `arg` is an additional argument for the request. The interpretation of `arg` is request dependent. Refer to Table 14–1 on page 14–11 for the generic requests a device might support. Whether a call to `alt_dma_txchan_ioctl()` is thread-safe, or can be called from an ISR, is device dependent. In general it safest to assume that it is not thread-safe. Do not call the `alt_dma_txchan_ioctl()` function while DMA transfers are pending, or unpredictable behavior could result.
Return:	A negative return value indicates failure; otherwise the interpretation of the return value is request specific.
See also:	alt_dma_rxchan_close() alt_dma_rxchan_depth() alt_dma_rxchan_ioctl() alt_dma_rxchan_open() alt_dma_rxchan_prepare() alt_dma_rxchan_reg() alt_dma_txchan_close() alt_dma_txchan_open() alt_dma_txchan_reg() alt_dma_txchan_send() alt_dma_txchan_space()

alt_dma_txchan_open()

Prototype:	`alt_dma_txchan alt_dma_txchan_open (const char* name)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_txchan_open()` function obtains an `alt_dma_txchan()` descriptor for a DMA transmit channel. The input argument, `name`, is the name of the associated physical device, for example, `/dev/dma_0`.
Return:	The return value is null on failure and non-null otherwise. If an error occurs, `errno` is set to `ENODEV`.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

alt_dma_txchan_reg()

Prototype:	`int alt_dma_txchan_reg (alt_dma_txchan_dev* dev)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_dma_dev.h>`
Description:	The `alt_dma_txchan_reg()` function registers a DMA transmit channel with the system. After it is registered, a device can be accessed using the functions described in "Using DMA Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.
	System behavior is undefined in the event that a channel is registered with a name that conflicts with an existing channel.
	The `alt_dma_txchan_reg()` function is not thread-safe if other threads are using the channel list at the time that `alt_dma_txchan_reg()` is called. Call `alt_dma_txchan_reg()` only in the following circumstances:
	■ When running in single-threaded mode.
	■ From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.
Return:	The return value is zero upon success. A negative return value indicates failure.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_send()
	alt_dma_txchan_space()

alt_dma_txchan_send()

Prototype:	`int alt_dma_txchan_send (alt_dma_txchan   dma,`
	`                         const void*      from,`
	`                         alt_u32          length,`
	`                         alt_txchan_done* done,`
	`                         void*            handle)`

Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	See description.
Available from ISR:	See description.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_txchan_send()` function posts a transmit request to a DMA transmit channel. The input arguments are: `dma`, the channel to use; from, a pointer to the start of the data to send; `length`, the length of the data to send in bytes; `done`, a callback function that is called after the data is sent; and `handle`, an opaque value passed to done.
	Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.
Return:	The return value is negative if the request cannot be posted, and zero otherwise.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_space()

alt_dma_txchan_space()

Prototype:	`int alt_dma_txchan_space (alt_dma_txchan dma)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	See description.
Available from ISR:	See description.
Include:	`<sys/alt_dma.h>`
Description:	The `alt_dma_txchan_space()` function returns the number of transmit requests that can be posted to the specified DMA transmit channel, `dma`. A negative value indicates that the value cannot be determined.
	Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.
Return:	Returns the number of transmit requests that can be posted.
See also:	alt_dma_rxchan_close()
	alt_dma_rxchan_depth()
	alt_dma_rxchan_ioctl()
	alt_dma_rxchan_open()
	alt_dma_rxchan_prepare()
	alt_dma_rxchan_reg()
	alt_dma_txchan_close()
	alt_dma_txchan_ioctl()
	alt_dma_txchan_open()
	alt_dma_txchan_reg()
	alt_dma_txchan_send()

alt_erase_flash_block()

Prototype:	`int alt_erase_flash_block(alt_flash_fd* fd,`
	`                          int        offset,`
	`                          int        length)`

Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_flash.h>`
Description:	The `alt_erase_flash_block()` function erases an individual flash erase block. The parameter `fd` specifies the flash device; `offset` is the offset within the flash of the block to erase; `length` is the size of the block to erase. No error checking is performed to check that this is a valid block, or that the length is correct. Refer to "Using Flash Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.
	Call the `alt_erase_flash_block()` function only when operating in single-threaded mode.
	The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.
Return:	The return value is zero upon success. A negative return value indicates failure.
See also:	alt_flash_close_dev()
	alt_flash_open_dev()
	alt_get_flash_info()
	alt_read_flash()
	alt_write_flash()
	alt_write_flash_block()

alt_exception_cause_generated_bad_addr()

Prototype:	`int alt_exception_cause_generated_bad_addr`
	`( alt_exception_cause cause)`
Commonly called by:	Instruction-related exception handlers
Thread-safe:	
Available from ISR:	
Include:	`<sys/alt_exceptions.h>`
Description:	This function validates the `bad_addr` argument to an instruction-related exception handler. The function parses the handler's `cause` argument to determine whether the `bad_addr` register contains the exception-causing address.
	If the exception is of a type that generates a valid address in `bad_addr`, this function returns a nonzero value. Otherwise, it returns zero.
	If the `cause` register is unimplemented in the Nios® II processor core, this function always returns zero.
Return:	A nonzero value means `bad_addr` contains the exception-causing address.
	Zero means the value of `bad_addr` is to be ignored.
See also:	alt_instruction_exception_register()

alt_flash_close_dev()

Prototype:	`void alt_flash_close_dev(alt_flash_fd* fd)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_flash.h>`
Description:	The `alt_flash_close_dev()` function closes a flash device. All subsequent calls to `alt_write_flash()`, `alt_read_flash()`, `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()` for this flash device fail.
	Call the `alt_flash_close_dev()` function only when operating in single-threaded mode.
	The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.
Return:	–
See also:	alt_erase_flash_block()
	alt_flash_open_dev()
	alt_get_flash_info()
	alt_read_flash()
	alt_write_flash()
	alt_write_flash_block()

alt_flash_open_dev()

Prototype:	`alt_flash_fd* alt_flash_open_dev(const char* name)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_flash.h>`
Description:	The `alt_flash_open_dev()` function opens a flash device. After it is opened, you can perform the following operations:

- Write to a flash device using `alt_write_flash()`
- Read from a flash device using `alt_read_flash()`
- Control individual flash blocks using `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()`.

Call the `alt_flash_open_dev` function only when operating in single-threaded mode.

Return:	The return value is zero upon failure. Any other value indicates success.
See also:	alt_erase_flash_block()
	alt_flash_close_dev()
	alt_get_flash_info()
	alt_read_flash()
	alt_write_flash()
	alt_write_flash_block()

alt_fs_reg()

Prototype:	`int alt_fs_reg (alt_dev* dev)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_dev.h>`
Description:	The `alt_fs_reg()` function registers a file system with the HAL. After it is registered, a file system can be accessed using the standard I/O functions. Refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

System behavior is undefined in the event that a file system is registered with a name that conflicts with an existing device or file system.

`alt_fs_reg()` is not thread-safe if other threads are using the device list at the time that `alt_fs_reg()` is called. Call `alt_fs_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return:	The return value is zero upon success. A negative return value indicates failure.
See also:	alt_dev_reg()

alt_get_flash_info()

Prototype:	`int alt_get_flash_info(alt_flash_fd*  fd,`
	 `                        flash_region** info,`
	 `                        int*           number_of_regions)`

Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_flash.h>`
Description:	The `alt_get_flash_info()` function gets the details of the erase region of a flash part. The flash part is specified by the descriptor `fd`, a pointer to the start of the `flash_region` structures is returned in the `info` parameter, and the number of flash regions are returned in number of regions. Call this function only when operating in single-threaded mode. The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.
Return:	The return value is zero upon success. A negative return value indicates failure.
See also:	alt_erase_flash_block() alt_flash_close_dev() alt_flash_open_dev() alt_read_flash() alt_write_flash() alt_write_flash_block()

alt_ic_irq_disable()

Prototype:	`int alt_ic_irq_disable (alt_u32 ic_id, alt_u32 irq)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_irq.h>`
Description:	The `alt_ic_irq_disable()` function disables a single interrupt.

The function arguments are as follows:

- `ic_id` is the interrupt controller identifier (ID) as defined in **system.h**, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.

- irq is the interrupt request (IRQ) number, as defined in system.h, identifying the interrupt to enable.

- A driver for an external interrupt controller (EIC) must implement this function.

Return:	This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the external interrupt controller.
See also:	alt_irq_disable_all()
	alt_irq_enable()
	alt_irq_enable_all()
	alt_irq_enabled()
	alt_irq_register()
	alt_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_register()

alt_ic_irq_enable()

Prototype:	`int alt_ic_irq_enable (alt_u32 ic_id, alt_u32 irq)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_irq.h>`
Description:	The `alt_ic_irq_enable()` function enables a single interrupt.

The function arguments are as follows:

- `ic_id` is the interrupt controller ID as defined in **system.h**, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.

- `irq` is the IRQ number, as defined in **system.h**, identifying the interrupt to enable.

- A driver for an EIC must implement this function.

Return:	This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the external interrupt controller.
See also:	alt_irq_disable()
	alt_irq_disable_all()
	alt_irq_enable_all()
	alt_irq_enabled()
	alt_irq_register()
	alt_irq_enable()
	alt_ic_irq_disable()
	alt_ic_irq_enabled()
	alt_ic_isr_register()

alt_ic_irq_enabled()

Prototype:	`int alt_ic_irq_enabled (alt_u32 ic_id, alt_u32 irq)`
Commonly called by:	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_irq.h>`
Description:	This function determines whether a specified interrupt is enabled.

The function arguments are as follows:

- `ic_id` is the interrupt controller ID as defined in **system.h**, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.

- `irq` is the IRQ number, as defined in **system.h**, identifying the interrupt to enable.

- A driver for an EIC must implement this function.

Return:	Returns zero if the specified interrupt is disabled, and nonzero otherwise.
See also:	alt_irq_disable()
	alt_irq_disable_all()
	alt_irq_enable()
	alt_irq_enable_all()
	alt_irq_register()
	alt_irq_enabled()
	alt_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_isr_register()

alt_ic_isr_register()

Prototype:	`int alt_ic_isr_register (alt_u32 ic_id,`
	`                          alt_u32 irq,`
	`                          alt_isr_func isr,`
	`                          void*   isr_context,`
	`                          void*   flags)`

Commonly called by:	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_irq.h>`
Description:	The `alt_ic_isr_register()` function registers an ISR. If the function is successful, the requested interrupt is enabled on return, and `isr` and `isr_context` are inserted in the vector table.

The function arguments are as follows:

- `ic_id` is the interrupt controller ID as defined in **system.h**, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.

- `irq` is the IRQ number, as defined in **system.h**, identifying the interrupt to register.

- `isr` is the function that is called when the interrupt is accepted.

- `isr_context` is the input argument to `isr`. `isr_context` points to a data structure associated with the device driver instance.

- `flags` is reserved.

The ISR function prototype is defined as follows:

```
typedef void (*alt_isr_func) (void* isr_context);
```

Calls to `alt_ic_isr_register()` replace previously registered handlers for interrupt `irq`.

If `isr` is set to null, the interrupt is disabled.

- A driver for an EIC must implement this function.

Return:	This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the external interrupt controller.
See also:	alt_irq_disable() alt_irq_disable_all() alt_irq_enable() alt_irq_enable_all() alt_irq_enabled() alt_irq_register() alt_ic_irq_disable() alt_ic_irq_enable() alt_ic_irq_enabled()

alt_icache_flush()

Prototype:	`void alt_icache_flush (void* start, alt_u32 len)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_cache.h>`
Description:	The `alt_icache_flush()` function invalidates the instruction cache for a memory region of length `len` bytes, starting at address `start`.
	In processors without instruction caches, it has no effect.
Return:	–
See also:	alt_dcache_flush()
	alt_dcache_flush_all()
	alt_icache_flush_all()
	alt_remap_cached()
	alt_remap_uncached()
	alt_uncached_free()
	alt_uncached_malloc()

alt_icache_flush_all()

Prototype:	`void alt_icache_flush_all (void)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_cache.h>`
Description:	The `alt_icache_flush_all()` function invalidates the entire contents of the instruction cache.
	In processors without instruction caches, it has no effect.
Return:	–
See also:	alt_dcache_flush()
	alt_dcache_flush_all()
	alt_icache_flush()
	alt_remap_cached()
	alt_remap_uncached()
	alt_uncached_free()
	alt_uncached_malloc()

alt_instruction_exception_register()

Prototype:	```void alt_instruction_exception_register (```
	``` alt_exception_result (*handler)```
	``` (alt_exception_cause cause,```
	``` alt_u32 exception_pc,```
	``` alt_u32 bad_addr))```

Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	Yes.
Include:	`<sys/alt_exceptions.h>`
Description:	The HAL API function `alt_instruction_exception_register()` registers an instruction-related exception handler. The `handler` argument is a pointer to the instruction-related exception handler.
	You can only use this API function if you have enabled the `hal.enable_instruction_related_exceptions_api` setting in the board support package (BSP). For details, refer to "Settings Managed by the Software Build Tools" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.
	Register the instruction-related exception handler as early as possible in function `main()`. This allows you to handle abnormal conditions during startup.
	You can register an exception handler from the `alt_main()` function.
	A call to `alt_instruction_exception_register()` replaces the previously registered exception handler, if any. If `handler` is set to null, the instruction-related exception handler is removed.
	For further usage details, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.
Return:	—
See also:	alt_irq_register()
	alt_exception_cause_generated_bad_addr()

alt_irq_disable()

Prototype:	`int alt_irq_disable (alt_u32 id)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_irq.h>`
Description:	The `alt_irq_disable()` function disables a single interrupt.
	☞ This function is part of the legacy HAL interrupt API, which is deprecated. Altera recommends using the enhanced HAL interrupt API.
	For details about using the enhanced HAL interrupt API, refer to "Nios II Interrupt Service Routines" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.
Return:	The return value is zero.
See also:	alt_irq_disable_all()
	alt_irq_enable()
	alt_irq_enable_all()
	alt_irq_enabled()
	alt_irq_register()
	alt_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_register()

alt_irq_disable_all()

Prototype:	`alt_irq_context alt_irq_disable_all (void)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_irq.h>`
Description:	The `alt_irq_disable_all()` function disables all maskable interrupts. Nonmaskable interrupts (NMIs) are unaffected.
Return:	Pass the return value as the input argument to a subsequent call to `alt_irq_enable_all()`.
See also:	alt_irq_disable()
	alt_irq_enable()
	alt_irq_enable_all()
	alt_irq_enabled()
	alt_irq_register()
	alt_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_register()

alt_irq_enable()

Prototype:	`int alt_irq_enable (alt_u32 id)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_irq.h>`
Description:	The `alt_irq_enable()` function enables a single interrupt.
Return:	The return value is zero.
See also:	alt_irq_disable()
	alt_irq_disable_all()
	alt_irq_enable_all()
	alt_irq_enabled()
	alt_irq_register()
	alt_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_register()

alt_irq_enable_all()

Prototype:	`void alt_irq_enable_all (alt_irq_context context)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_irq.h>`
Description:	The `alt_irq_enable_all()` function enables all interrupts that were previously disabled by `alt_irq_disable_all()`. The input argument, `context`, is the value returned by a previous call to `alt_irq_disable_all()`. Using `context` allows nested calls to `alt_irq_disable_all()` and `alt_irq_enable_all()`. As a result, `alt_irq_enable_all()` does not necessarily enable all interrupts, such as interrupts explicitly disabled by `alt_irq_disable()`.
Return:	–
See also:	alt_irq_disable()
	alt_irq_disable_all()
	alt_irq_enable()
	alt_irq_enabled()
	alt_irq_register()
	alt_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_register()

alt_irq_enabled()

Prototype:	`int alt_irq_enabled (void)`
Commonly called by:	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_irq.h>`
Description:	Determines whether maskable exceptions (`status.PIE`) are enabled.
	☞ This function is part of the legacy HAL interrupt API, which is deprecated. Altera recommends using the enhanced HAL interrupt API.
	For details about using the enhanced HAL interrupt API, refer to "Nios II Interrupt Service Routines" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.
Return:	Returns zero if interrupts are disabled, and non-zero otherwise.
See also:	alt_irq_disable()
	alt_irq_disable_all()
	alt_irq_enable()
	alt_irq_enable_all()
	alt_irq_register()
	alt_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_register()

alt_irq_register()

Prototype:	`int alt_irq_register (alt_u32 id,` `void*   context,` `void   (*isr)(void*, alt_u32))`
Commonly called by:	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_irq.h>`
Description:	The `alt_irq_register()` function registers an ISR. If the function is successful, the requested interrupt is enabled on return.
	The input argument `id` is the interrupt to enable. `isr` is the function that is called when the interrupt is active. `context` and `id` are the two input arguments to `isr`.
	Calls to `alt_irq_register()` replace previously registered handlers for interrupt `id`.
	If `irq_handler` is set to null, the interrupt is disabled.
	☞ This function is part of the legacy HAL interrupt API, which is deprecated. Altera recommends using the enhanced HAL interrupt API.
	For details about using the enhanced HAL interrupt API, refer to "Nios II Interrupt Service Routines" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.
Return:	The `alt_irq_register()` function returns zero if successful, or non-zero otherwise.
See also:	alt_irq_disable()
	alt_irq_disable_all()
	alt_irq_enable()
	alt_irq_enable_all()
	alt_irq_enabled()
	alt_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_register()

alt_llist_insert()

Prototype:	`void alt_llist_insert(alt_llist* list,` `                       alt_llist* entry)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	Yes.
Include:	`<sys/alt_llist.h>`
Description:	The `alt_llist_insert()` function inserts the doubly linked list entry `entry` in the list `list`. This operation is not reentrant. For example, if a list can be manipulated from different threads, or from within both application code and an ISR, some mechanism is required to protect access to the list. Interrupts can be locked, or in MicroC/OS-II, a `mutex` can be used.
Return:	–
See also:	alt_llist_remove()

alt_llist_remove()

Prototype:	`void alt_llist_remove(alt_llist* entry)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	Yes.
Include:	`<sys/alt_llist.h>`
Description:	The `alt_llist_remove()` function removes the doubly linked list entry `entry` from the list it is currently a member of. This operation is not reentrant. For example if a list can be manipulated from different threads, or from within both application code and an ISR, some mechanism is required to protect access to the list. Interrupts can be locked, or in MicroC/OS-II, a `mutex` can be used.
Return:	–
See also:	alt_llist_insert()

alt_load_section()

Prototype:	`void alt_load_section(alt_u32* from,`
	`                      alt_u32* to,`
	`                      alt_u32* end)`

Commonly called by:	C/C++ programs
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_load.h>`
Description:	When operating in run-from-flash mode, the sections `.exceptions`, `.rodata`, and `.rwdata` are automatically loaded from the boot device to RAM at boot time. However, if there are any additional sections that require loading, the `alt_load_section()` function loads them manually before these sections are used.
	The input argument `from` is the start address in the boot device of the section; `to` is the start address in RAM of the section, and `end` is the end address in RAM of the section.
	To load one of the additional memory sections provided by the default linker script, use the macro `ALT_LOAD_SECTION_BY_NAME` rather than calling `alt_load_section()` directly. For example, to load the section `.onchip_ram`, use the following code:
	`ALT_LOAD_SECTION_BY_NAME(onchip_ram);`
	The leading '.' is omitted in the section name. This macro is defined in the header **sys/alt_load.h**.
Return:	–
See also:	–

alt_nticks()

Prototype:	`alt_u32 alt_nticks (void)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_alarm.h>`
Description:	The `alt_nticks()` function.
Return:	Returns the number of elapsed system clock tick since reset. It returns zero if there is no system clock available.
See also:	alt_alarm_start()
	alt_alarm_stop()
	alt_sysclk_init()
	alt_tick()
	alt_ticks_per_second()
	gettimeofday()
	settimeofday()
	times()
	usleep()

alt_read_flash()

Prototype:	```int alt_read_flash(alt_flash_fd* fd,```
	``` int offset,```
	``` void* dest_addr,```
	``` int length)```

Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_flash.h>`
Description:	The `alt_read_flash()` function reads data from flash. `length` bytes are read from the flash `fd`, starting `offset` bytes from the beginning of the flash and are written to the location `dest_addr`. Call this function only when operating in single-threaded mode. The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.
Return:	The return value is zero on success and nonzero otherwise.
See also:	alt_erase_flash_block() alt_flash_close_dev() alt_flash_open_dev() alt_get_flash_info() alt_write_flash() alt_write_flash_block()

alt_remap_cached()

Prototype:	`void* alt_remap_cached (volatile void* ptr,` `                         alt_u32       len);`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_cache.h>`
Description:	The `alt_remap_cached()` function remaps a region of memory for cached access. The memory to map is `len` bytes, starting at address `ptr`.
	Processors that do not have a data cache return uncached memory.
Return:	The return value for this function is the remapped memory region.
See also:	alt_dcache_flush()
	alt_dcache_flush_all()
	alt_icache_flush()
	alt_icache_flush_all()
	alt_remap_uncached()
	alt_uncached_free()
	alt_uncached_malloc()

alt_remap_uncached()

Prototype:	`volatile void* alt_remap_uncached (void*  ptr,` `                                    alt_u32 len);`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_cache.h>`
Description:	The `alt_remap_uncached()` function remaps a region of memory for uncached access. The memory to map is `len` bytes, starting at address `ptr`. Processors that do not have a data cache return uncached memory.
Return:	The return value for this function is the remapped memory region.
See also:	alt_dcache_flush() alt_dcache_flush_all() alt_icache_flush() alt_icache_flush_all() alt_remap_cached() alt_uncached_free() alt_uncached_malloc()

alt_sysclk_init()

Prototype:	`int alt_sysclk_init (alt_u32 nticks)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_alarm.h>`
Description:	The `alt_sysclk_init()` function registers the presence of a system clock driver. The input argument is the number of ticks per second at which the system clock is run.
	The expectation is that this function is only called from within `alt_sys_init()`, that is, while the system is running in single-threaded mode. Concurrent calls to this function might lead to unpredictable results.
Return:	This function returns zero on success; otherwise it returns a negative value. The call can fail if a system clock driver is already registered, or if no system clock device is available.
See also:	alt_alarm_start()
	alt_alarm_stop()
	alt_nticks()
	alt_tick()
	alt_ticks_per_second()
	gettimeofday()
	settimeofday()
	times()
	usleep()

alt_tick()

Prototype:	`void alt_tick (void)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	Yes.
Include:	`<sys/alt_alarm.h>`
Description:	Only the system clock driver may call the `alt_tick()` function. The driver is responsible for making periodic calls to this function at the rate specified in the call to `alt_sysclk_init()`. This function provides notification to the system that a system clock tick has occurred. This function runs as a part of the ISR for the system clock driver.
Return:	–
See also:	alt_alarm_start()
	alt_alarm_stop()
	alt_nticks()
	alt_sysclk_init()
	alt_ticks_per_second()
	gettimeofday()
	settimeofday()
	times()
	usleep()

alt_ticks_per_second()

Prototype:	`alt_u32 alt_ticks_per_second (void)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_alarm.h>`
Description:	The `alt_ticks_per_second()` function returns the number of system clock ticks that elapse per second. If there is no system clock available, the return value is zero.
Return:	Returns the number of system clock ticks that elapse per second.
See also:	alt_alarm_start()
	alt_alarm_stop()
	alt_nticks()
	alt_sysclk_init()
	alt_tick()
	gettimeofday()
	settimeofday()
	times()
	usleep()

alt_timestamp()

Prototype:	`alt_u32 alt_timestamp (void)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	See description.
Include:	`<sys/alt_timestamp.h>`
Description:	The `alt_timestamp()` function returns the current value of the timestamp counter. Refer to "Using Timer Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level depends on the underlying driver.
	Always call the `alt_timestamp_start()` function before any calls to `alt_timestamp()`. Otherwise the behavior of `alt_timestamp()` is undefined.
Return:	Returns the current value of the timestamp counter.
See also:	alt_timestamp_freq()
	alt_timestamp_start()

alt_timestamp_freq()

Prototype:	`alt_u32 alt_timestamp_freq (void)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	See description.
Include:	`<sys/alt_timestamp.h>`
Description:	The `alt_timestamp_freq()` function returns the rate at which the timestamp counter increments. Refer to "Using Timer Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level depends on the underlying driver.
Return:	The returned value is the number of counter ticks per second.
See also:	alt_timestamp()
	alt_timestamp_start()

alt_timestamp_start()

Prototype:	`int alt_timestamp_start (void)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	See description.
Include:	`<sys/alt_timestamp.h>`
Description:	The `alt_timestamp_start()` function starts the system timestamp counter. Refer to "Using Timer Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level depends on the underlying driver. This function resets the counter to zero, and starts the counter running.
Return:	The return value is zero on success and nonzero otherwise.
See also:	alt_timestamp() alt_timestamp_freq()

alt_uncached_free()

Prototype:	`void alt_uncached_free (volatile void* ptr)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_cache.h>`
Description:	The `alt_uncached_free()` function causes the memory pointed to by `ptr` to be deallocated, that is, made available for future allocation through a call to `alt_uncached_malloc()`. The input pointer, `ptr`, points to a region of memory previously allocated through a call to `alt_uncached_malloc()`. Behavior is undefined if this is not the case.
Return:	–
See also:	alt_dcache_flush()
	alt_dcache_flush_all()
	alt_icache_flush()
	alt_icache_flush_all()
	alt_remap_cached()
	alt_remap_uncached()
	alt_uncached_malloc()

alt_uncached_malloc()

Prototype:	`volatile void* alt_uncached_malloc (size_t size)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<sys/alt_cache.h>`
Description:	The `alt_uncached_malloc()` function allocates a region of uncached memory of length `size` bytes. Regions of memory allocated in this way can be released using the `alt_uncached_free()` function.
	Processors that do not have a data cache return uncached memory.
Return:	If sufficient memory cannot be allocated, this function returns null, otherwise a pointer to the allocated space is returned.
See also:	alt_dcache_flush()
	alt_dcache_flush_all()
	alt_icache_flush()
	alt_icache_flush_all()
	alt_remap_cached()
	alt_remap_uncached()
	alt_uncached_free()

alt_write_flash()

Prototype:	`int alt_write_flash(alt_flash_fd* fd,`
	`                    int         offset,`
	`                    const void*  src_addr,`
	`                    int          length)`

Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_flash.h>`
Description:	The `alt_write_flash()` function writes data to flash. The data to be written is at address `src_addr`. `length` bytes are written to the flash `fd`, `offset` bytes from the beginning of the flash device address space.
	Call this function only when operating in single-threaded mode. This function does not preserve any unwritten areas of any flash sectors affected by this write. Refer to "Using Flash Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.
	The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.
Return:	The return value is zero on success and nonzero otherwise.
See also:	alt_erase_flash_block()
	alt_flash_close_dev()
	alt_flash_open_dev()
	alt_get_flash_info()
	alt_read_flash()
	alt_write_flash_block()

alt_write_flash_block()

Prototype:	```int alt_write_flash_block(alt_flash_fd* fd,```
	``` int block_offset,```
	``` int data_offset,```
	``` const void *data,```
	``` int length)```

Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	`<sys/alt_flash.h>`
Description:	The `alt_write_flash_block()` function writes one block of data of flash. The data to be written is at address `data`. `length` bytes are written to the flash `fd`, into the block starting at offset `block_offset` from the beginning of the flash address space. The data starts at offset `data_offset` from the beginning of the flash address space.
	No check is performed on any of the parameters. Refer to "Using Flash Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.
	Call this function only when operating in single-threaded mode.
	The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.
Return:	The return value is zero on success and nonzero otherwise.
See also:	alt_erase_flash_block()
	alt_flash_close_dev()
	alt_flash_open_dev()
	alt_get_flash_info()
	alt_read_flash()
	alt_write_flash()

close()

Prototype:	`int close (int fd)`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	`<unistd.h>`
Description:	The `close()` function is the standard UNIX-style `close()` function, which closes the file descriptor `fd`.
	Calls to `close()` are thread-safe only if the implementation of `close()` provided by the driver that is manipulated is thread-safe.
	Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.
Return:	The return value is zero on success, and −1 otherwise. If an error occurs, `errno` is set to indicate the cause.
See also:	fcntl()
	fstat()
	ioctl()
	isatty()
	lseek()
	open()
	read()
	stat()
	write()
	newlib documentation

execve()

Prototype:	`int execve(const char   *path,`
	`             char *const  argv[],`
	`             char *const  envp[])`

Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<unistd.h>`
Description:	The `execve()` function is only provided for compatibility with newlib.
Return:	Calls to `execve()` always fail with the return code −1 and `errno` set to `ENOSYS`.
See also:	newlib documentation

fcntl()

Prototype:	`int fcntl(int fd, int cmd)`
Commonly called by:	C/C++ programs
Thread-safe:	No.
Available from ISR:	No.
Include:	`<unistd.h>` `<fcntl.h>`
Description:	The `fcntl()` function is a limited implementation of the standard `fcntl()` system call, which can change the state of the flags associated with an open file descriptor. Normally these flags are set during the call to `open()`. The main use of this function is to change the state of a device from blocking to nonblocking (for device drivers that support this feature). The input argument `fd` is the file descriptor to be manipulated. `cmd` is the command to execute, which can be either `F_GETFL` (return the current value of the flags) or `F_SETFL` (set the value of the flags).
Return:	If `cmd` is `F_SETFL`, the argument `arg` is the new value of flags, otherwise `arg` is ignored. Only the flags `O_APPEND` and `O_NONBLOCK` can be updated by a call to `fcntl()`. All other flags remain unchanged. The return value is zero on success, or –1 otherwise. If `cmd` is `F_GETFL`, the return value is the current value of the flags. If an error occurs, –1 is returned. In the event of an error, `errno` is set to indicate the cause.
See also:	close() fstat() ioctl() isatty() lseek() read() stat() write() newlib documentation

fork()

Prototype:	`pid_t fork (void)`
Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	no
Include:	`<unistd.h>`
Description:	The `fork()` function is only provided for compatibility with newlib.
Return:	Calls to `fork()` always fails with the return code –1 and `errno` set to `ENOSYS`.
See also:	newlib documentation

fstat()

Prototype:	`int fstat (int fd, struct stat *st)`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	`<sys/stat.h>`
Description:	The `fstat()` function obtains information about the capabilities of an open file descriptor. The underlying device driver fills in the input `st` structure with a description of its functionality. Refer to the header file **sys/stat.h** provided with the compiler for the available options.
	By default, file descriptors are marked as character devices, unless the underlying driver provides its own implementation of the `fstat()` function.
	Calls to `fstat()` are thread-safe only if the implementation of `fstat()` provided by the driver that is manipulated is thread-safe.
	Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.
Return:	The return value is zero on success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.
See also:	close()
	fcntl()
	ioctl()
	isatty()
	lseek()
	open()
	read()
	stat()
	write()
	newlib documentation

getpid()

Prototype:	`pid_t getpid (void)`
Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<unistd.h>`
Description:	The `getpid()` function is provided for newlib compatibility and obtains the current process ID.
Return:	Because HAL systems cannot contain multiple processes, `getpid()` always returns the same ID number.
See also:	newlib documentation

gettimeofday()

Prototype:	`int gettimeofday(struct timeval *ptimeval,` `                  struct timezone *ptimezone)`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	See description.
Available from ISR:	Yes.
Include:	`<sys/time.h>`
Description:	The `gettimeofday()` function obtains a time structure that indicates the current time. This time is calculated using the elapsed number of system clock ticks, and the current time value set by the most recent call to `settimeofday()`.
	If this function is called concurrently with a call to `settimeofday()`, the value returned by `gettimeofday()` is unreliable; however, concurrent calls to `gettimeofday()` are legal.
Return:	The return value is zero on success. If no system clock is available, the return value is `-ENOTSUP`.
See also:	alt_alarm_start()
	alt_alarm_stop()
	alt_nticks()
	alt_sysclk_init()
	alt_tick()
	alt_ticks_per_second()
	settimeofday()
	times()
	usleep()
	newlib documentation

ioctl()

Prototype:	`int ioctl (int fd, int req, void* arg)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	No.
Include:	`<sys/ioctl.h>`
Description:	The `ioctl()` function allows application code to manipulate the I/O capabilities of a device driver in driver-specific ways. This function is equivalent to the standard UNIX `ioctl()` function. The input argument `fd` is an open file descriptor for the device to manipulate, `req` is an enumeration defining the operation request, and the interpretation of `arg` is request specific.
	For file subsystems, `ioctl()` is wrapper function that passes control directly to the appropriate device driver's `ioctl()` function (as registered in the driver's `alt_dev` structure).
	For devices, `ioctl()` handles `TIOCEXCL` and `TIOCNXCL` requests internally, without calling the device driver. These requests lock and release a device for exclusive access. For requests other than `TIOCEXCL` and `TIOCNXCL`, `ioctl()` passes control to the device driver's `ioctl()` function.
	Calls to `ioctl()` are thread-safe only if the implementation of `ioctl()` provided by the driver that is manipulated is thread-safe.
	Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.
Return:	The interpretation of the return value is request specific. If the call fails, `errno` is set to indicate the cause of the error.
See also:	close()
	fcntl()
	fstat()
	isatty()
	lseek()
	open()
	read()
	stat()
	write()
	newlib documentation

isatty()

Prototype:	`int isatty(int fd)`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	`<unistd.h>`
Description:	The `isatty()` function determines whether the device associated with the open file descriptor `fd` is a terminal device. This implementation uses the driver's `fstat()` function to determine its reply.
	Calls to `isatty()` are thread-safe only if the implementation of `fstat()` provided by the driver that is manipulated is thread-safe.
Return:	The return value is 1 if the device is a character device, and zero otherwise. If an error occurs, `errno` is set to indicate the cause.
See also:	close()
	fcntl()
	fstat()
	ioctl()
	lseek()
	open()
	read()
	stat()
	write()
	newlib documentation

kill()

Prototype:	`int kill(int pid, int sig)`
Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<signal.h>`
Description:	The `kill()` function is used by newlib to send signals to processes. The input argument `pid` is the ID of the process to signal, and `sig` is the signal to send. As there is only a single process in the HAL, the only valid values for `pid` are either the current process ID, as returned by `getpid()`, or the broadcast values, that is, `pid` must be less than or equal to zero.

The following signals result in an immediate shutdown of the system, without call to `exit()`: `SIGABRT`, `SIGALRM`, `SIGFPE`, `SIGILL`, `SIGKILL`, `SIGPIPE`, `SIGQUIT`, `SIGSEGV`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGBUS`, `SIGPOLL`, `SIGPROF`, `SIGSYS`, `SIGTRAP`, `SIGVTALRM`, `SIGXCPU`, and `SIGXFSZ`.

The following signals are ignored: `SIGCHLD` and `SIGURG`.

All the remaining signals are treated as errors. |
| Return: | The return value is zero on success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error. |
| See also: | newlib documentation |

link()

Prototype:	`int link(const char *_path1,` `           const char *_path2)`
Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<unistd.h>`
Description:	The `link()` function is only provided for compatibility with newlib.
Return:	Calls to `link()` always fails with the return code –1 and `errno` set to `ENOSYS`.
See also:	newlib documentation

lseek()

Prototype:	`off_t lseek(int fd, off_t ptr, int whence)`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	`<unistd.h>`
Description:	The `lseek()` function moves the read/write pointer associated with the file descriptor `fd`. `lseek()` is wrapper function that passes control directly to the `lseek()` function registered for the driver associated with the file descriptor. If the driver does not provide an implementation of `lseek()`, an error is reported.
	`lseek()` corresponds to the standard UNIX `lseek()` function.
	You can use the following values for the input parameter, `whence`:
	■ `SEEK_SET`—The offset is set to `ptr` bytes.
	■ `SEEK_CUR`—The offset is incremented by `ptr` bytes.
	■ `SEEK_END`—The offset is set to the end of the file plus `ptr` bytes.
	Calls to `lseek()` are thread-safe only if the implementation of `lseek()` provided by the driver that is manipulated is thread-safe.
	Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.
Return:	On success, the return value is a nonnegative file pointer. The return value is –1 in the event of an error. If the call fails, `errno` is set to indicate the cause of the error.
See also:	close()
	fcntl()
	fstat()
	ioctl()
	isatty()
	open()
	read()
	stat()
	write()
	newlib documentation

open()

Prototype:	`int open (const char* pathname, int flags, mode_t mode)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	No.
Include:	`<unistd.h>` `<fcntl.h>`
Description:	The `open()` function opens a file or device and returns a file descriptor (a small, nonnegative integer for use in read, write, etc.) `flags` is one of: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, which request opening the file in read-only, write-only, or read/write mode, respectively. You can also bitwise-OR `flags` with `O_NONBLOCK`, which causes the file to be opened in nonblocking mode. Neither `open()` nor any subsequent operation on the returned file descriptor causes the calling process to wait. Not all file systems/devices recognize this option. `mode` specifies the permissions to use, if a new file is created. It is unused by current file systems, but is maintained for compatibility. Calls to `open()` are thread-safe only if the implementation of `open()` provided by the driver that is manipulated is thread-safe.
Return:	The return value is the new file descriptor, and –1 otherwise. If an error occurs, `errno` is set to indicate the cause.
See also:	close() fcntl() fstat() ioctl() isatty() lseek() read() stat() write() newlib documentation

read()

Prototype:	`int read(int fd, void *ptr, size_t len)`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	`<unistd.h>`
Description:	The `read()` function reads a block of data from a file or device. `read()` is wrapper function that passes control directly to the `read()` function registered for the device driver associated with the open file descriptor `fd`. The input argument, `ptr`, is the location to place the data read and `len` is the length of the data to read in bytes.
	Calls to `read()` are thread-safe only if the implementation of `read()` provided by the driver that is manipulated is thread-safe.
	Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.
Return:	The return argument is the number of bytes read, which might be less than the requested length
	The return value is –1 upon an error. In the event of an error, `errno` is set to indicate the cause.
See also:	close()
	fcntl()
	fstat()
	ioctl()
	isatty()
	lseek()
	open()
	stat()
	write()
	newlib documentation

sbrk()

Prototype:	`caddr_t sbrk(int incr)`
Commonly called by:	newlib C library
Thread-safe:	No.
Available from ISR:	No.
Include:	`<unistd.h>`
Description:	The `sbrk()` function dynamically extends the data segment for the application. The input argument `incr` is the size of the block to allocate. Do not call `sbrk()` directly. If you wish to dynamically allocate memory, use the newlib `malloc()` function.
Return:	–
See also:	newlib documentation

settimeofday()

Prototype:	`int settimeofday (const struct timeval  *t,` `                   const struct timezone *tz)`
Commonly called by:	C/C++ programs
Thread-safe:	No.
Available from ISR:	Yes.
Include:	`<sys/time.h>`
Description:	If the `settimeofday()` function is called concurrently with a call to `gettimeofday()`, the value returned by `gettimeofday()` is unreliable.
Return:	The return value is zero on success. If no system clock is available, the return value is -1, and `errno` is set to `ENOSYS`.
See also:	alt_alarm_start()
	alt_alarm_stop()
	alt_nticks()
	alt_sysclk_init()
	alt_tick()
	alt_ticks_per_second()
	gettimeofday()
	times()
	usleep()

stat()

Prototype:	`int stat(const char *file_name,` `            struct stat *buf);`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	`<sys/stat.h>`
Description:	The `stat()` function is similar to the `fstat()` function—It obtains status information about a file. Instead of using an open file descriptor, like `fstat()`, `stat()` takes the name of a file as an input argument.
	Calls to `stat()` are thread-safe only if the implementation of `stat()` provided by the driver that is manipulated is thread-safe.
	Internally, the `stat()` function is implemented as a call to `fstat()`. Refer to "fstat()" on page 14–62.
Return:	–
See also:	close()
	fcntl()
	fstat()
	ioctl()
	isatty()
	lseek()
	open()
	read()
	write()
	newlib documentation

times()

Prototype:	`clock_t times (struct tms *buf)`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/times.h>`
Description:	This `times()` function is provided for compatibility with newlib. It returns the number of clock ticks since reset. It also fills in the structure pointed to by the input parameter `buf` with time accounting information. The definition of the `tms` structure is:

```
typedef struct
{
  clock_t tms_utime;
  clock_t tms_stime;
  clock_t tms_cutime;
  clock_t tms_cstime;
};
```

The structure has the following elements:

- `tms_utime`: the processor time charged for the execution of user instructions
- `tms_stime`: the processor time charged for execution by the system on behalf of the process
- `tms_cutime`: the sum of the values of `tms_utime` and `tms_cutime` for all child processes
- `tms_cstime`: the sum of the values of `tms_stime` and `tms_cstime` for all child processes

In practice, all elapsed time is accounted as system time. No time is ever attributed as user time. In addition, no time is allocated to child processes, as child processes cannot be spawned by the HAL.

Return:	If there is no system clock available, the return value is zero, and `errno` is set to `ENOSYS`.
See also:	alt_alarm_start()
	alt_alarm_stop()
	alt_nticks()
	alt_sysclk_init()
	alt_tick()
	alt_ticks_per_second()
	gettimeofday()
	settimeofday()
	usleep()
	newlib documentation

unlink()

Prototype:	`int unlink(char *name)`
Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<unistd.h>`
Description:	The `unlink()` function is only provided for compatibility with newlib.
Return:	Calls to `unlink()` always fails with the return code −1 and `errno` set to `ENOSYS`.
See also:	newlib documentation

usleep()

Prototype:	`int usleep (unsigned int us)`
Commonly called by:	C/C++ programs
	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<unistd.h>`
Description:	The `usleep()` function blocks until at least `us` microseconds have elapsed.
Return:	The `usleep()` function returns zero on success, or –1 otherwise. If an error occurs, `errno` is set to indicate the cause. The current implementation always succeeds.
See also:	alt_alarm_start()
	alt_alarm_stop()
	alt_nticks()
	alt_sysclk_init()
	alt_tick()
	alt_ticks_per_second()
	gettimeofday()
	settimeofday()
	times()

wait()

Prototype:	`int wait(int *status)`
Commonly called by:	newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/wait.h>`
Description:	newlib uses the `wait()` function to wait for all child processes to exit. Because the HAL does not support spawning child processes, this function returns immediately.
Return:	On return, the content of `status` is set to zero, which indicates there is no child processes.
	The return value is always −1 and `errno` is set to `ECHILD`, which indicates that there are no child processes to wait for.
See also:	newlib documentation

write()

Prototype:	`int write(int fd, const void *ptr, size_t len)`
Commonly called by:	C/C++ programs
	newlib C library
Thread-safe:	See description.
Available from ISR:	no
Include:	`<unistd.h>`
Description:	The `write()` function writes a block of data to a file or device. `write()` is wrapper function that passes control directly to the `write()` function registered for the device driver associated with the file descriptor `fd`. The input argument `ptr` is the data to write and `len` is the length of the data in bytes.
	Calls to `write()` are thread-safe only if the implementation of `write()` provided by the driver that is manipulated is thread-safe.
	Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.
Return:	The return argument is the number of bytes written, which might be less than the requested length.
	The return value is –1 upon an error. In the event of an error, `errno` is set to indicate the cause.
See also:	close()
	fcntl()
	fstat()
	ioctl()
	isatty()
	lseek()
	open()
	read()
	stat()
	newlib documentation

HAL Standard Types

In the interest of portability, the HAL uses a set of standard type definitions in place of the ANSI C built-in types. Table 14–2 describes these types, which are defined in the header file **alt_types.h**.

Table 14–2. HAL Standard Types

Type	Description
alt_8	Signed 8-bit integer.
alt_u8	Unsigned 8-bit integer.
alt_16	Signed 16-bit integer.
alt_u16	Unsigned 16-bit integer.
alt_32	Signed 32-bit integer.
alt_u32	Unsigned 32-bit integer.

Table 14–2. HAL Standard Types

Type	Description
alt_64	Signed 64-bit integer.
alt_u64	Unsigned 64-bit integer.

Document Revision History

Table 14–3 shows the revision history for this document.

Table 14–3. Document Revision History

Date	Version	Changes
May 2011	11.0.0	No change
February 2011	10.1.0	Removed "Referenced Documents" section.
July 2010	10.0.0	■ Clarify purpose of listed C header file for functions. ■ Correction: alt_irq_enabled() is not a legacy function.
November 2009	9.1.0	■ Document new enhanced HAL interrupt API functions: alt_ic_irq_disable(), alt_ic_irq_enable(), alt_ic_irq_enabled(), and alt_ic_isr_register(). ■ Deprecate legacy HAL interrupt API functions alt_irq_disable(), alt_irq_enable(), alt_irq_enabled(), and alt_irq_register().
March 2009	9.0.0	■ Corrected minor typographical errors.
May 2008	8.0.0	■ Advanced exceptions added to Nios II core. ■ Instruction-related exception handling added to HAL. ■ Added alt_instruction_exception_register() and alt_exception_cause_generated_bad_addr() for instruction-related exception handlers.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Added table of contents to "Introduction" section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Function open() requires fcntl.h.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Added API entries for "alt_irq_disable()" and "alt_irq_enable()", which were previously omitted by error.
May 2005	5.0.0	■ Added alt_load_section() function. ■ Added fcntl() function.
December 2004	1.2	Updated names of DMA generic requests.
September 2004	1.1	■ Added open(). ■ Added ERRNO information to alt_dma_txchan_open(). ■ Corrected ALT_DMA_TX_STREAM_ON definition. ■ Corrected ALT_DMA_RX_STREAM_ON definition. ■ Added information to alt_dma_rxchan_ioctl() and alt_dma_txchan_ioctl().
May 2004	1.0	Initial release.