*HUNT ENGINEERING*
**Chestnut Court, Burton Row,**
**Brent Knoll, Somerset, TA9 4BP, UK**
**Tel: (+44) (0)1278 760188,**
**Fax: (+44) (0)1278 760199,**
**Email: sales@hunteng.co.uk**
**http://www.hunteng.co.uk**
**http://www.hunt-dsp.com**

# *HERON-API*

# *Hardware Access Library for HUNT ENGINEERING C6000 HERON processing modules*

# *USER MANUAL*

***Version 3.9***
***Document Rev M***
***P.Warnes & R.Williams 30/7/04***

## COPYRIGHT



## WARRANTIES LIABILITY and INDEMNITIES

HUNT ENGINEERING warrants the hardware to be free from defects in the material and workmanship for 12 months from the date of purchase. Product returned under the terms of the warranty must be returned carriage paid to the main offices of HUNT ENGINEERING situated at BRENT KNOLL Somerset UK, the product will be repaired or replaced at the discretion of HUNT ENGINEERING.

Exclusions - If HUNT ENGINEERING decides that there is any evidence of electrical or mechanical abuse to the hardware, then the customer shall have no recourse to HUNT ENGINEERING or its agents. In such circumstances HUNT ENGINEERING may at its discretion offer to repair the hardware and charge for that repair.

Limitations of Liability - HUNT ENGINEERING makes no warranty as to the fitness of the product for any particular purpose. In no event shall HUNT ENGINEERING'S liability related to the product exceed the purchase fee actually paid by you for the product. Neither HUNT ENGINEERING nor its suppliers shall in any event be liable for any indirect, consequential or financial damages caused by the delivery, use or performance of this product.

Because some states do not allow the exclusion or limitation of incidental or consequential damages or limitation on how long an implied warranty lasts, the above limitations may not apply to you.

## TECHNICAL SUPPORT

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/index.htm on the HUNT ENGINEERING web site. This includes FAQs, latest product, software and documentation updates etc. Or contact your local supplier - if you are unsure of details please refer to www.hunteng.co.uk for the list of current re-sellers.

HUNT ENGINEERING technical support can be contacted by emailing support@hunteng.demon.co.uk, calling the direct support telephone number +44 (0)1278 760775, or by calling the general number +44 (0)1278 760188 and choosing the technical support option.

**Version Control**

Version 1.0 – First version

Software v2.0 Major rewrite to remove design problems. Document rev. C refers.

Software v2.1 FIFO performance enhanced, two new functions. Document rev. D refers.

Software v2.2 Support added for dedicated DMA. Document rev. E refers.

Software v2.3 Some function names changed to match xDAIS guidelines (see section Migrating from Previous Versions of the HERON-API). Changed FIFO Handle type to be "HeronFIFO" to match xDAIS guidelines. New functions added: HeronReadWord,HeronWriteWord, HeronCancelWithFlush, HeronInstallIsr, HeronUmiOut, HeronUmiIn and HeronModId. Some functions made inline to increase efficiency.

Software v2.4 FIFO performance enhanced. Functions HeronTestIo and HeronWaitIo rewritten as inline functions. HeronRead and HeronWrite made partially inline. Functions HeronRestartRead and HeronRestartWrite added.

Software v2.5 Supported added for the HERON4 module. HERON1 code also improved. New function HeronSwiOpenFifo added.

Software v2.6 Correction made to the HERON4 FIFO addresses.

Software v2.7 UMI functions added for the HERON4. Some changes made to the way functions are inlined. Corrected HeronWriteWord for the HERON1.

Software v3.0 Streaming I/O (SIO) added for both the HERON1 and HERON4. New functions added: HeronSioOpenFifo, HeronSioClose, HeronSioRead, HeronSioWrite and HeronSioReadAlign.

Software v3.1 HeronSerial Bus (HSB) functions added for the HERON1 and HERON4. New functions added: HeronHsbOpen, HeronHsbClose, HeronHsbSendMessage, HeronHsbReceiveMessage, HeronHsbStartSendMessage, HeronHsbSendMessageData, HeronHsbEndOfSendMessage, HeronHsbStartReceiveMessage, HeronHsbReceiveMessageData, and HeronHsbEndOfReceiveMessage.

Software v3.2 Dedicated DMA functions for HERON4 and HERON1 had bugs. These have been fixed at this version.

Software v3.3 first version of HERON API to support HEART (HEPC9 like) carriers. HEPC8 is now supported as a legacy carrier.

Documentation change to clarify use of DMA reload register

Software v3.9 added HeronHsbSendMessageEx function to account for latency between EM2 connected HSBs on different boards.

# TABLE OF CONTENTS

## FIFO Accesses

The idea of HERON-API is to have a standard set of functions to access the communications FIFOs of a HERON processor module. These FIFOs are provided by the HERON module carrier boards, which will determine how they are used to interconnect the HERON modules. The way that the HERON processor module accesses the FIFOs through its module connectors is actually defined by the HERON module design.

What the HERON-API provides is a standard set of functions to access the FIFOs, regardless of the actual HERON module type. In this way a user program can be written to work with all HERON module types, present and future. As new HERON modules are introduced that access the FIFOs in a different manner, or that use newer 'C6000 family members, the user program can be easily re-used by recompiling with the new HERON-API version that supports those modules.

## User DMA Management

Because the HERON-API will use DMA where possible to perform FIFO accesses, it is important that if the user wishes to use DMAs themselves, that they claim a DMA channel from the HERON-API.

This prevents the HERON-API from disturbing the user's DMA and vice-versa.

## HERON Serial Bus Functions

The HERON-API includes functions that enable messages to be sent and received using the HERON Serial Bus (HSB). The HERON Serial Bus is a relatively slow bus in comparison to the fast FIFO interface. The serial bus is typically used to retrieve module information or to configure devices in the system.

## Hardware Specific Functions

There are also some other functions that allow the user access to certain other HERON module functions, so that they also can be accessed through a function that removes the specific hardware details that may change from module design to module design. An example of this is access to the Digital I/O.

The HERON-API is a 'C6000 library, and as such is provided as several `.lib` files. There are four libraries that are provided. Each library supports a different combination of data access model and code access model. In order to select the appropriate library for use with your application, please refer to the section 'Selecting the Right Library for your Application'.

In your source code you include the HERON-API functions by having a header file included. The name of this header file defines the type of HERON module that you are using. For example,

```
#include heron4.h
```

will include the HERON-API functions for the HERON4 module.

If you later wish to run your application on another HERON module type, this is the only part of your source file that you will need to change.

The HERON-API has two code section names associated with it. These are `heronapi_code` and `heronapi_data`. These section names are provided to allow some control over where these sections are placed. They should be included in the linker command file (.cmd) that you use to control the 'C6000 linker.

The section `heronapi_code` contains the instructions of the HERON-API functions. The section `heronapi_data` contains the variable data structures used by the HERON-API functions.

The placement of these sections is discussed in the section 'Building your Application'.

## Migrating from Previous Versions of the HERON-API

Version 3.3 supports HEART devices like the HEPC9, but has to maintain support for the legacy HEPC8. The same HERON-API cannot be used for both board types due to the differing FIFO flag positions. Version 3.3 provides separate (but almost identical) source trees for the two board types as it is anticipated that future versions of HERON-API will be further optimised in its use of the HEART FIFOs, and will add some new functions to support new hardware features.

The differing carrier types are supported in the same way as the differing module type, i.e. through the #include statement. Now three #includes are supported:-

#include heron4.h this is HEART based carrier support for the HERON4

#include heron4_pc8.h this is HEPC8 support for the HERON4

#include heron1_pc8.h this is HEPC8 support for the HERON1

At Version 2.3, there were several unfortunate but necessary changes made to the HERON-API definitions. These changes were made in order to comply with `xDAIS` guidelines. These are detailed as follows:-

1. The FIFO handle type definition '`FIFO`' is replaced by the new type name '`HeronFIFO`'.

2. The `config_off()` and `config_on()` functions are replaced by the `HeronConfigOff()` and `HeronConfigOn()` functions which are just name changes to

the functions.

3. The `digio_out()` and `digio_in()` functions are replaced by the `HeronDigioOut()` and `HeronDigioIn()` functions which are just name changes to the functions.

4. The `dma_claim()` and `dma_free()` functions are replaced by the `HeronDmaClaim()` and `HeronDmaFree()` functions which are just name changes to the functions.

As a result of these changes, the HERON-API now supports a standard naming scheme for all function names, global variables, types and definitions.

All HERON-API function names and type definitions start with the text '`Heron`', and all global variables defined by the HERON-API library begin with the text '`HRN`'. Any definitions supplied in the user application should therefore avoid text that starts with '`Heron`' or '`HRN`'. Doing so will enable the user to avoid conflicts between objects and function names when the application is built.

## FIFO Accessing

The HERON-API provides many functions for accessing the communications FIFO of a HERON system. These functions support different programming models and enable the use of key features of DSP/BIOS in transferring data between a user application and a FIFO.

Each FIFO is treated as a resource, that must first be "opened". Once a FIFO has been successfully opened, the user is given a FIFO handle with which to operate on that FIFO. Any subsequent attempts to open that FIFO will fail, ensuring that a program cannot mistakenly use the same FIFO twice and cause program errors.

A FIFO can be opened to read, or opened to write. Although an application may read and write the same FIFO, this must be done using two separate handles. One handle must be obtained by opening the FIFO with a read flag, and a separate handle must be obtained by opening the FIFO with a write flag.

After successfully opening a FIFO, a program can transfer FIFO data using one of several methods. Which method is used will depend on the requirements of each individual application, but all must involve initiating a transfer to or from a FIFO, and later detecting that the transfer has completed.

The following section discusses each transfer method and explains when each method would typically be used.

When all data transfer to or from a FIFO has been completed, that FIFO should then be "closed". The operation of closing the FIFO returns it to a state where another process may re-open that FIFO and perform a new task.

## FIFO Access Models

The HERON-API is intended for use as part of a DSP/BIOS application, and as such provides functions that use many of the key features of DSP/BIOS.

DSP/BIOS enables an application to be constructed as collection of threads, each of which carries out a modularised function. DSP/BIOS enables a multithreaded program to run on a single processor by allowing higher priority threads to preempt lower priority threads, and by allowing various types of communication between threads.

DSP/BIOS provides several types of threads, which include hardware interrupts (HWIs), software interrupts (SWIs), and tasks (TSKs).

The HERON-API provides FIFO access functions that can be called within the context of a software interrupt or task. It also uses hardware interrupts internally to maintain the operation of each FIFO transfer.

Depending on the application you are developing you may wish to perform a FIFO transfer as part of a SWI or as part of a task. For each, there is a different programming model that should be used. These models are explained below.

For each model, a more detailed description and example of the functions used can be found in the section 'HERON-API Functions'. Examples of the programming models can also be found on the HUNT ENGINEERING CD, under the '`heron_api_examples`' directory of the '`examples`' directory tree.

### The WaitIo Model

The FIFO transfer functions `HeronRead` and `HeronWrite` are asynchronous. This means that calling the read or write function only initiates the transfer. The read or write itself proceeds in the background.

When using `HeronRead` or `HeronWrite`, another function call is required to detect the completion of the transfer. For the `WaitIo` model, this is done by calling the function `HeronWaitIo`.

`HeronWaitIo` should be called after a transfer has been started using `HeronRead` or `HeronWrite` and the program needs to do no other work than wait for the completion of that I/O. When `HeronWaitIo` is called, it will simply not return until the I/O has completed.

`HeronWaitIo` must only be used within the context of a task. As `HeronWaitIo` implements a blocking mechanism that waits for the completion of the I/O, it cannot be used within the context of a SWI. This is because a SWI should always run to completion without blocking (the use of SWIs is further described below in the section 'The SWI Model').

When using `HeronWaitIo`, the FIFO must first be opened with `HeronOpenFifo`. Once processing is complete for that FIFO, the FIFO may be closed with `HeronClose`.

### The TestIo Model

The `TestIo` model is similar to the `WaitIo` model, in that it is used as part of an asynchronous transfer. For the `TestIo` model, the completion of a previously started transfer is detected by a call to `HeronTestIo`, but unlike `HeronWaitIo` this function

does not block.

When called, `HeronTestIo` will immediately return a status about the I/O specified by the handle. The status will be "in-progress" or "complete".

In comparison to `HeronWaitIo`, this function allows useful processing to be done by the processor while I/O completion is still awaited.

Following the start of a transfer, a certain amount of processing can be performed before returning to check the state of the transfer. Each time the call returns with the status "in-progress", further processing can again be performed. This loop can be repeated several times until the transfer finally completes and the status is returned "complete".

`HeronTestIo` can be used within the context of both a task or a SWI, as it does not block.

When using `HeronTestIo`, the FIFO must first be opened with `HeronOpenFifo`. Once processing is complete for that FIFO, the FIFO may be closed with `HeronClose`.

### The SWI Model

A SWI, or software interrupt is modelled after the operation of hardware interrupts. A SWI however, is triggered by calling SWI functions from within the program. When a SWI function runs, it must run to completion. This is because the priority of SWIs is higher than that of tasks. If a SWI were to block, it would also block the operation of all tasks (as well as any other SWIs of the same or lower priority).

The HERON-API supports the use of SWIs in two ways. Firstly, the functions `HeronRead` and `HeronWrite` may be called within the context of a SWI. These functions are asynchronous, in that each will initiate either a read or write and will then return. As such, these functions can never block waiting for the completion of a transfer.

Secondly, the HERON-API can be used to post a SWI on the completion of a transfer.

By opening a FIFO handle using the function `HeronSwiOpenFifo`, a software interrupt can be attached to the completion of a transfer for that FIFO. When opening a FIFO in this way, `HeronTestIo` and `HeronWaitIo` are no longer required. Instead, a user function specified through a DSP/BIOS SWI object is attached to the FIFO handle. Each time a transfer completes for that FIFO, the specified function is posted for execution.

### The SEM Model

The SEM model is provided as an extension to the use of `HeronWaitIo`. When `HeronWaitIo` is used to test for the completion of a transfer, it will block the task within which it is called. When `HeronWaitIo` blocks it will also not cause a task switch, and will therefore block all other tasks until the transfer completes.

When you need to block waiting for the completion of a transfer, but still want all other tasks to be able to run, you should use the SEM model.

By opening a FIFO handle using the function `HeronSemOpenFifo`, a semaphore can be attached to the completion of a transfer for that FIFO. When opening a FIFO in this way, `HeronTestIo` and `HeronWaitIo` are no longer required. Instead, a user supplied DSP/BIOS semaphore object is attached to the FIFO handle. After each call to `HeronRead` or `HeronWrite` a SEM_pend function call should be used to block the user task. When the transfer completes for that FIFO, the semaphore is posted, unblocking the previously blocked task.

### The SIO Model

The Streaming I/O model, or 'SIO' model is provided for the transfer of data to or from high speed I/O modules.

The SIO model is an extension of the SEM model, in that the transfer is data is performed using semaphores. However, the Streaming I/O functions further separate the user application from the read or write process by implementing queues of data buffers between the application and the opened FIFO.

Each call to a Streaming I/O function involves the transfer of a used buffer into one queue, and the return of a second new buffer from a second queue.

By opening a FIFO handle using `HeronSioOpenFifo`, the asynchronous Streaming I/O (SIO) functions `HeronSioRead` and `HeronSioWrite` can be used for the transfer of data.

Unlike the functions `HeronRead` and `HeronWrite`, these SIO functions will block until the transfer is complete, by using a DSP/BIOS semaphore. The SIO functions are used on their own in that they do not require the use of `HeronTestIo` or `HeronWaitIo`.

As the SIO functions use a semaphore to block waiting for buffers to be available from the SIO queues, they must be used only with the context of a task.

## An Example of FIFO Access with the Waitlo Model

The `HeronRead` and `HeronWrite` functions of the HERON-API are asynchronous which means that after the read or write has been started, the transfer will continue in the background enabling the user application to perform processing in the foreground before returning to check whether the I/O has completed.

The HERON-API FIFO Access functions also allow transfers from multiple FIFOs to be performed in parallel to each other and to the processing of the user application.

The following code example shows how this can be achieved with the HERON-API. In this example, three transfers are being performed in parallel. Two transfers are receiving data from FIFOs 1 and 2, and one transfer is sending data to FIFO 2. Once the transfers have all been started, some user processing can begin while the transfers complete in the background. Once the processing is complete, the transfers can be checked to wait for completion through the use of the function `HeronWaitIo`.

In the example, if an error occurs at any stage, the program simply exits.

```
/* Open the FIFOs */
handle_a = HeronOpenFifo(1, "r");
handle_b = HeronOpenFifo(2, "w");
handle_c = HeronOpenFifo(2, "r");

/* Check the HeronOpenFifo calls were successful */
if (handle_a == NULL) exit(0);
if (handle_b == NULL) exit(0);
if (handle_c == NULL) exit(0);

/* Start three transfers (two reads and one write) */
status_a = HeronRead(handle_a,buffer_a,size_a);
status_b = HeronWrite(handle_b,buffer_b,size_b);
status_c = HeronRead(handle_c,buffer_c,size_c);

/* Check that the transfers were started successfully */
if (status_a != HERON_IO_IN_PROGRESS) exit(0);
if (status_b != HERON_IO_IN_PROGRESS) exit(0);
if (status_c != HERON_IO_IN_PROGRESS) exit(0);

/* Do some processing here */

/* Wait for each transfer to complete */
status_a = HeronWaitIo(handle_a);
status_b = HeronWaitIo(handle_b);
status_c = HeronWaitIo(handle_c);

/* Check there were no problems */
if (status_a != HERON_OK) exit(0);
if (status_b != HERON_OK) exit(0);
if (status_c != HERON_OK) exit(0);
```

## An Example of FIFO Access with the TestIo Model

The `TestIo` model is used in a very similar way to the previous example of `WaitIo`, the difference being that it is possible to use a loop that tests for completion each time a portion of processing is done.

For an example of this, look at the HUNT ENGINEERING CD, choose Getting Started and then HERON-API examples where you will see examples of all the models.

## An Example of FIFO Access with the SWI Model

To use the SWI model, a function must be created that will run each time a transfer completes. In addition, a SWI object must be created. The properties of the SWI object should be set so the new SWI function is run each time that SWI is posted.

With this done, the SWI object is attached to the appropriate FIFO through a call to the function `HeronSwiOpenFifo`.

The SWI function that is set up should then contain a call to `HeronRead` or `HeronWrite`, so that as each transfer is completed, the next begins.

For an example of this, look at the HUNT ENGINEERING CD, choose Getting Started and then HERON-API examples where you will see examples of all the models.

## An Example of FIFO Access with the SEM Model

To use the SEM model, a SEM object must be created. With this done, the SEM object must be attached to the appropriate FIFO through a call to the function `HeronSemOpenFifo`.

Then a task is required that calls `HeronRead` or `HeronWrite`, and then calls the DSP/BIOS function, `SEM_pend`. The call to `SEM_pend` will ensure that the task will block until the transfer completes. When `SEM_pend` returns, the task can then start the next transfer.

For an example of this, look at the HUNT ENGINEERING CD, choose Getting Started and then HERON-API examples where you will see examples of all the models.

## An Example of FIFO Access with the Streaming I/O Model

The `HeronSioRead` and `HeronSioWrite` functions of the HERON-API are provided to transfer data to and from the HERON FIFOs using Streaming I/O (SIO).

These functions are asynchronous, in that the transfer continues in the background, while processing continues in the foreground.

However, rather than being used to directly transfer a buffer to or from a HERON FIFO, the SIO functions are used to pass data between two queues managed by the HERON-API. By using buffer queues, the transfer of data between the FIFOs and the user application is further de-coupled in comparison to the asynchronous FIFO access functions.

The HERON-API is used to manage the transfer of data between the queues and the FIFOs. It works by repeatedly transferring many buffers of the same size. As soon as one buffer transfer is complete, the next is started.

This leaves the user application free to pass buffers to and from the other end of the two buffer queues at its own rate.

For each FIFO opened to use SIO, two queues are created. One for buffers to be passed to the HERON-API, and one for buffers passed from the HERON-API to the user application.

If the FIFO is opened for reading, the first queue is used to pass empty buffers to the HERON-API, and the second queue is used to receive full buffers from the HERON-API.

Conversely, if the FIFO is opened for writing, the first queue is used to pass full buffers to the HERON-API, and the second queue is used to receive empty buffers back from the HERON-API.

The following code example shows how SIO can be used to read data from a FIFO. In this example, repeated transfers are performed using many buffers of the same size. Once the first transfer is started the HERON-API will continue to receive data in the background. As each buffer is filled and passed back to the user application in the second queue, a new transfer is automatically started in the background using a buffer from the first queue. For each time round the loop, one new buffer of data is passed to the user process, and the previous used buffer is returned to the SIO stream.

```
/* Declare a data pointer */
unsigned int *data;

/* Open FIFO 2 for reading using SIO */
handle = HeronSioOpenFifo(2, "r", &sio_obj, (Ptr *)&data);

/* Check the HeronSioOpenFifo call was successful */
if (handle == NULL) exit(0);

/* Start the transfer and processing of data */
for (i=0;i<N;i++) {
      status = HeronSioRead(handle,(Ptr *)&data);

      /* Do some processing here on the data */
      /* pointed to by the pointer 'data' */
}
```

The following code example shows how SIO can be used to write data to a FIFO. In this

example, repeated transfers are performed using many buffers of the same size. Once the first transfer is started the HERON-API will continue to output data in the background. As each buffer is filled and passed to the SIO stream in the first queue, an empty buffer is returned from the second queue.

For each time round the loop, one new buffer of data is passed to the SIO stream for output, and another empty buffer is returned.

```
/* Declare a data pointer */
unsigned int *data;

/* Open FIFO 2 for writing using SIO */
handle = HeronSioOpenFifo(2, "w", &sio_obj, (Ptr *)&data);

/* Check the HeronSioOpenFifo call was successful */
if (handle == NULL) exit(0);

/* Start the transfer and processing of data */
for (i=0;i<N;i++) {
      /* Do some processing here to */
      /* create an output buffer */

      /* Exchange buffer pointers */
      status = HeronSioWrite(handle,(Ptr *)&data);
}
```

## User DMA Management

If the user is going to use a DMA channel of the processor, a DMA channel *must* be claimed from the HERON-API before a DMA transfer is programmed. This will ensure that the user DMA does not conflict with the FIFO accessing functions of the HERON-API.

There are three DMA management functions. The first, `HeronDmaClaim` must be called to allocate a DMA channel to the user.

HERON-API uses only DMA reload register A, leaving reload register B free for the user to use when programming DMAs that have been claimed.

The second function `HeronDmaFree` should be called after the user has finished using a DMA channel. This function will return control of the DMA channel to the HERON-API.

The third function `HeronInstallIsr` is provided so that an interrupt service routine can be installed on a specific hardware event. This function should be used to add an interrupt function to a DMA complete interrupt that is associated with a DMA that was claimed with `HeronDmaClaim`. Alternatively, this could be used to add an interrupt service routine dynamically during program execution, rather than statically through the DSP/BIOS Configuration Database file (the `.cdb` file). In all other cases, an interrupt should be installed by setting up an appropriate ISR in the DSP/BIOS `.cdb` file.

## Hardware Access Functions

The hardware access functions do not really have a structure, but merely provide access to hardware features of the HERON modules.

These functions do not perform any device protection using an open/close sequence.

## HERON Serial Bus Functions

Each HERON module has a HERON Serial Bus (HSB) device that can be used to send and receive messages with other HSB devices located on other modules or on the carrier board.

As with the FIFO Access functions, the HSB device must first be opened before it is used. Once the HSB device has been successfully opened the user is given a HSB handle with which to operate on that device. Any subsequent attempts to open that device will fail, ensuring that a program cannot mistakenly use the same HSB device twice and cause program errors.

After successfully opening the HSB device, messages can then be sent and received to or from any other HSB device in the system.

The HERON-API includes two main functions for transmitting messages using the HERON serial bus (HSB). The first function `HeronHsbSendMessage` is used to send messages from the module to another hardware device and the second function `HeronHsbReceiveMessage` is used to receive messages.

The process of sending or receiving a message is further split into three parts, starting the message, sending or receiving multiple blocks of message data, and ending the message. The two main HSB functions are each implemented through calls to six lower level functions that handle the three parts of each message transmission.

Please note, although it is possible to directly access these lower level HSB functions using the HERON-API, it is recommended that this is only done when absolutely necessary. For the majority of users it is recommended that only the main HSB message functions are used.

When all message transfers have been completed, the HSB device should then be "closed". The operation of closing the HSB device returns it to a state where another process may re-open that device and perform a new task.

### HSB Messages

To use the HERON Serial Bus, you need to adhere to a protocol. That is, both the sending end and the receiving end must agree to the format of each message transmitted.

There is an implicit message format that is used by the HSB functions provided in the HERON-API. As such, any use of the HSB functions must build upon this basic format.

All HSB messages must start with an address byte. This is used to identify the intended destination for the message. This address is formed from information such as the board ID and the slot ID of the intended message recipient.

The HSB functions that are provided automatically create this address byte using the '`board`' and '`slot`' function arguments.

Following the address byte two data bytes are transmitted. The first of these bytes is used to define the '`message type`', and the second byte defines the '`address`' of the device initiating the message.

The message type byte can be used to define what to do with the optional data bytes that follow, and whether a reply is required. The second 'address' byte can be used to form the new address if message is to be sent in reply to the original message.

With these bytes sent, any number of optional data bytes may follow.

### The Message Type

For many applications the message type is an important value that defines the exact operation that is to take place when using HSB. A simple example is the message type of 1. This value is used by HUNT ENGINEERING to indicate a Module Type Query.

Messages transmitted with this message type will typically be used to find out information about the type of module that is located in a particular slot of a carrier board. The Module Type Query message is a simple message that has no optional data bytes, just the address, the message type (set to 1), and the address to reply to.

Other examples are the transmission of a serial bitstream to a FPGA located on a HERON-FPGA module. This message (message type 3) will always containing as many optional data bytes as there are bytes in the bitstream for the FPGA.

If the sender and receiver processes are both *your* code, no pre-defined message type is needed, essentially you may define your own. It is recommended however that you avoid the message types already used by HUNT ENGINEERING software to prevent confusion.

HUNT ENGINEERING defines message numbers starting from one upwards, so using message types greater than 128 will avoid confusion. The message protocol used by HUNT ENGINEERING is defined in a separate document.

### HSB and EM2 Inter-Board Connectors

Note that if you send a message across EM2 inter-board connectors to another board, it may happen that error `HERON_HSB_NO_RESPONSE` is returned. This is not necessarily a fault. If two HSB messages are sent in quick succession, the HSB on the other board may still be busy when you start the next message. The HSB that is busy will cause the function to return error message `HERON_HSB_NO_RESPONSE`.

In such situations, send the whole message again. If you use `HeronHsbSendMessage`, and it returns `HERON_HSB_NO_RESPONSE`, execute `HeronHsbSendMessage` again using the same parameters. With version 3.9 of the API, you can also use a function that automatically retries sending an HSB message upon error `HERON_HSB_NO_RESPONSE`. This is the `HeronHsbSendMessageEx` function. Parameter `retries` in this function allows you to specify how many times you want to retry sending the message.

If you use `HeronHsbStartSendMessage` and `HeronHsbSendMessageData` to send a message, the error `HERON_HSB_NO_RESPONSE` may be returned when `HeronHsbSend-MessageData` is called. To retry, you need to resend the whole message, that is, first again call `HeronHsbStartSendMessage`, then all calls to `HeronHsbSendMessageData` as you did in the message you tried to send the first time, using the same parameters.

In my experience, with two boards connected by two EM2 inter-board connectors, one retry is enough. With more than 2 boards connected up you may need perhaps 2 or 3 retries, as latencies are a bit longer.

## Selecting the Right Library for your Application

The HERON-API is provided in library form that must be linked with your application. There are four different libraries that are provided to support different combinations of data access and code access within the library.

When building your application you will only need to link to one of the four libraries depending on the requirements of your application. The library that you should link to is discussed in this section.

When developing your application there are two main points to consider that affect the memory model you compile with and the libraries that you link to.

The first is how data is placed and accessed, and the second is how code is placed and accessed.

### Data Access

The C compiler creates a default section in which it places all global and static data. This section is called `.bss`. With the creation of this section, the C compiler also generates a pointer to the beginning of this section. This pointer is called the Data Page Pointer (DP).

When you build your application for the default compiler memory model (the small memory model), the compiler will by default place all global and static data in the `.bss` section and will access that data using the `near` access method. The near access method will access this data relative to the Data Page Pointer, and each access will take one assembly instruction. This is the most efficient form of data access.

With the small (default) memory model, there is a requirement that the size of the `.bss` section is less than 32Kbytes. This is because for the small memory model, a near data access uses an offset from the Data Page Pointer, where this offset is limited to 32Kbytes in size.

An alternative method of data access is the far access method. The far access method can be used by compiling with one of the large memory model options or by using the `far` keyword when declaring data. By accessing data as far, that data is now no longer placed in the `.bss` section, and there is now no 32Kbyte limit on the size of all far data.

However, unlike near accesses, far accesses are less efficient as each access will take three assembly instructions to perform. The extra instructions will increase the time the access takes to complete, and will increase the size of the program code.

So there is a trade-off between the use of near data and far data. Near data offers the best performance but a program is limited to having 32Kbytes of near data at most.

Far data does not have this 32Kbyte limit, but offers slower performance than near data.

Typically, when developing your application you will want to access the most important variables in your program as near data, and access the rest as far data, while ensuring that the total amount of near data you use is less than 32Kbytes.

The simplest way to achieve this is to globally define data that you want to access as near, and dynamically allocate the remaining far data using functions such as the TI provided Run-Time-Support (RTS) function, `malloc`. Such a program would be built using the default, small memory model.

### Code Access

When a function is called, the program must perform a branch from the function that is currently executing to the new function that has been called. This branch can be performed in one of two ways.

The first method is generated by the compiler adding a relative offset to the current program counter in order to perform the branch. This method is the `near` access method, and takes one assembly instruction to perform. It is the most efficient form of function call, and hence offers the best performance.

The second method is generated by the compiler forming an absolute address to which the program counter will branch. This method is the `far` access method. It is not as efficient as a near function call as three assembly instructions are required. The extra instructions will increase the time the function call takes to complete, and will increase the size of the program code.

When using a near function call, there is a requirement that the relative offset for the branch is less than 1Mword from the current program counter position. If this requirement is not met, a far function call is required.

When building with the small (default) memory model, all function calls are treated as near. By compiling with one of the large memory model options or by using the `far` keyword when declaring functions, a far call will be used.

As a general rule, far function calls should be used where the total code size is greater than 1Mword (or 4Mbytes), or if code sections are being placed in separate memory segments. Where code is placed in more than one memory segment, this will usually result in an the address span between the code in one segment and the code in another exceeding 1Mword. In this case, far function calls must be used.

### Choosing a Library

There are four HERON-API libraries, where each uses a different combination of data access method and code access method. You must only link to one of these libraries according to the requirements of your application.

When choosing a library it is recommended that you start with the library **`herons.lib`**. This library is expected to satisfy the requirements of most applications, but where it does not you should choose one of the other libraries as discussed below.

The library `herons.lib` has been built to access all data as near. As such, all global data used by this version of the HERON-API library is placed in the default section `.bss`. The only exception to this is the global error variable `heronerr` which is always declared as far and is placed in the section '`heronapi_data`'. All functions calls are performed as near.

The `herons.lib` library should be used where the total amount of global and static data in your application plus the HERON-API global data is less than 32Kbytes, and where the total size of the program code is less than 1Mword (or 4Mbytes). Note, all code sections should be placed in the same memory segment. In this case, your application should be built using the default small memory model to achieve the best performance.

Where the total amount of application code exceeds 1Mword, or if code is being placed across several memory segments, then the application should be built to use far function calls. In this case you should build with the memory model option `-ml1`, and you should link to the library `heronl1.lib`. This version of the library will access data as near and

will perform far function calls.

When you need your application to use all of the `.bss` section you should link to the library `heronl0.lib`. This version of the library places all of the library global data in a far data section named '`heronapi_data`'. By using this library, your application is free to use all of the `.bss` section. In this case you can build with small memory model, but please ensure that you keep the size of the `.bss` section to less than 32Kbytes. This library will perform all function calls as near.

If you want to combine the features of both the `heronl0` and `heronl1` libraries, you can use the library `heronl3.lib`. This version of the library will place all library global data in a far data section named '`heronapi_data`'. The library will perform all function calls as far. In this case you should build your application with the memory model option `-ml1`, keeping the size of the `.bss` section to less than 32Kbytes.

The table below summarises when you would use each library.

| Is the size of the `.bss` section <32Kbytes inc. HERON-API data? | Are all code sections to be placed in the same memory segment? | Is the total code size <1Mword? | |
|---|---|---|---|
| Yes | Yes | Yes | Use herons.lib |
| Yes | No | Yes | Use heronl1.lib |
| Yes | Yes | No | Use heronl1.lib |
| No | Yes | Yes | Use heronl0.lib |
| No | No | Yes | Use heronl3.lib |
| No | Yes | No | Use heronl3.lib |

## Linking Issues

When linking your application the linker will generate error messages if the rules for near data accesses or near function calls are broken.

If you are using the small memory model and are accessing data in the `.bss` section, the following error message will be generated when the size of the `.bss` exceeds 32Kbytes.

```
>> relocation value truncated at 0x?? in section .text, file
example.obj
```

In this case, you have too much global and static data in your application. This problem can be solved by either dynamically allocating some of the data using functions such as `malloc`, or by declaring some of your data as far, or by using a different HERON-API library that doesn't use the `.bss` section.

If you try to call a function using a near function call, and if that function is too far to be reached with the normal program counter relative (PC-relative) branch instruction, you will see the following linker error message:

```
>> PC-relative displacement overflow. Located in file.obj,
section .text, SPC offset ??
```

In this case, you are building your application to use near function calls and either the amount of code in your application exceeds 1Mwords, or you have placed some code sections in one memory segment, and other code sections in a separate memory segment.

Other than reducing the size of your code, the simplest way to remove this problem is to build your application using the `-ml1` memory model in order to use far function calls.

When re-building your application with this memory model, you will also need to link to a different HERON-API library. If you were previously linking to the library `herons.lib` then you will need to use `heronl1.lib` instead. However if you were linking to the library `heronl0.lib` then you will need to use `heronl3.lib`.

## Building your Application

This section describes how you would build a typical application that uses the HERON-API by starting a new project in Code Composer Studio.

When you are building an application that uses the HERON-API your application will be using DSP/BIOS. DSP/BIOS is a multi-threading environment that is provided as part of the Code Composer Development Environment. It also provides services for configuring processor features such as hardware interrupts and timers.

DSP/BIOS is included in Code Composer Studio, along with the Compiler tools for the 'C6000, and all users of HERON hardware will be able to use it.

This section assumes that the user has installed Code Composer Studio and followed the confidence checks. The user should also be familiar with using Code Composer Studio.

An application that uses DSP/BIOS is based around a Configuration Database file, or `.cdb` file. This file is used to define and control the various elements that make up a DSP/BIOS application.

When Code Composer Studio has been set up on your machine, copies of the HERON `.cdb` files should exist in the '`c6000\bios\include`' directory. This directory will be under the directory in which Code Composer Studio was installed, and is typically '`c:\ti`'.

These files will be automatically copied to this directory by the Code Composer Studio installation. However, if this installation was not successful and these files do not exist in this directory, then you will need to copy the files by hand from the directory '`%HEAPI_DIR%\heron_api\cmd`'.

In Code Composer Studio, select '`Project→New`' and choose the path and name for your project. Remember this name as you must use the same name when you save the `.cdb` file.

Select '`File→New→DSP/BIOS Config`' and choose the correct `.cdb` file for your hardware. The `.cdb` file will have a name that uses your HERON module number and possibly an option that is available for that module.

In the DSP/BIOS Configuration Tool, right click on Global Settings properties, and check that the CLKOUT property is set to the frequency of your processor module. This is used by DSP/BIOS to calculate the correct settings for the timer period.

This `.cdb` file has some items set up which are for the HERON-API. DO NOT CHANGE THESE!

Use '`File→Save`' to save the `.cdb` file to your project directory. This file must have the same name as the project name you have used, with the extension `.cdb`.

Saving the `.cdb` file wll generate a `.cmd` file, but that file will not control the placement of the sections '`heronapi_code`' and '`heronapi_data`'. For this reason there is a linker command file (a `.cmd` file), in the directory '`%HEAPI_DIR%\heron_api\cmd`' that will be called by your HERON module number and have '`_bios.cmd`' at the end. i.e. '`heronx_bios.cmd`'. You need to copy this file to your project directory.

Now add the source files to the project and the `.cdb` file. Also add the '`heronx_bios.cmd`' linker command file. In this linker command file is a line that references the linker command file that is generated when the `.cdb` file is saved. This line will need to be edited to the replace the asterisks with the correct part of the DSP/BIOS linker command file-name.

The section in this user manual entitled 'Selecting the Right Library for your Application' discusses which HERON-API library should be used for your application.

When you have selected a HERON-API library to link to, this library must be added to the project. Add the appropriate library to the project. The libraries are located in the directory '`%HEAPI_DIR%\heron_api\lib`'.

Depending on the choice of library you may wish to build the application for a memory model other than the default small memory model. If so, edit the Project Options to select the appropriate memory model.

Go to Project Options and add '`%HEAPI_DIR%\heron_api\inc`' to the include path.

The `.cdb` file you have selected will actually place all code into external memory, and will switch on the program cache. This is a good general purpose setting, but might need to be changed for your application.

You are now ready to build your HERON-API application, assuming that where necessary, you have set any application specific settings in the Configuration database file.

## Building an Application for Use with the Server/Loader

If your application is going to use the HUNT ENGINEERING Server/Loader, you will need to follow the description given above, and then make the following changes.

In place of the linker command file '`heronx_bios.cmd`' you should use the file '`heronx_slbios.cmd`'. Again, edit this linker command file so that it includes the linker command file generated when you saved the `.cdb` file. Please check that the path is correct for the inclusion of the Server/Loader library. If a different Server/Loader library is required, edit the line to specify the alternate library.

Ensure that you do not add the Server/Loader library to the project. The inclusion of this library must only come from the entry in the linker command file.

## How it Works With DSP/BIOS

The HERON-API uses some of the features DSP/BIOS. As such, any application that uses the HERON-API will also use DSP/BIOS, and as a result, each project will include a DSP/BIOS configuration database file (a `.cdb` file).

There are four important DSP/BIOS objects that are required by the HERON-API. These objects are automatically included when you select one of the HERON DSP/BIOS Configuration template `.cdb` files.

The first required object is the `HRN_prd` object. The `HRN_prd` object is used to run a periodic function in the HERON-API library. This function is used to keep FIFO transfers running smoothly. The template `.cdb` files use a timer tick set up on Timer0, that runs every 1ms.

The `HRN_prd` object uses this timer tick to run a function called '`HeronxTick`' where `x` is the number of the HERON module you are using. The function is run every 2ms.

You are free to change the way the timers are used in the `.cdb` file, but when doing so, please ensure one of the timers is used to drive a periodic tick, and please select a timer value that allows the `HeronxTick` function to run approximately every 2ms.

The next three required objects are three User Defined Devices. Each User Defined Device defines a Streaming I/O driver. Two of the devices define FIFO read drivers and one device defines a FIFO write driver. When creating an SIO object to use with HeronSioRead or HeronSioWrite, one of these three devices must be selected.

Device '`Dx0`' is a driver for reading from a FIFO using a trash buffer, device '`Dx1`' is a driver for reading from a FIFO without using a trash buffer and device '`Dx2`' is a driver for writing to a FIFO. The `x` in the device name represents the HERON module type you are using. For example, when using a HERON1 you would use devices '`D10`', '`D11`' and '`D12`'.

In addition to the `HRN_prd` object and User Defined Devices, the HERON-API uses eight of the hardware interrupts listed in the `Hardware Interrupt Service Routine Manager`. These hardware interrupts are required by the HERON-API, and therefore must not be used by the user. As such, the HERON template `.cdb` files include control of these interrupts.

The only exception to this rule is that it is possible to dynamically install an ISR on one of the DMA complete (DMA_INTx) interrupt cases. This may be done by calling `HeronInstallIsr` after a DMA channel has been successfully claimed from the HERON-API through a call to the function `HeronDmaClaim`.

The HERON-API can also be used to post a software interrupt using a SWI object created in your `.cdb` file, or post a semaphore using a SEM object.

By using the function `HeronSwiOpenFifo` to open a FIFO handle, a SWI object can be attached to that FIFO handle. This allows the HERON-API to post a software interrupt (SWI) each time a transfer on that FIFO completes. The use of software interrupts is further discussed in the description of the function `HeronSwiOpenFifo`.

Similarly by using the function `HeronSemOpenFifo` to open a FIFO handle, a SEM object can be attached to that FIFO handle. This allows the HERON-API to post a semaphore on completion of a FIFO transfer. The use of semaphores is further discussed in the description of the function `HeronSemOpenFifo`.

## Using the HERON-API Inline Functions

The HERON-API provides several functions as inline functions contained in the header file for the HERON module type you are using. For example, if you are using a HERON1 module, you will include the header file `heron1.h`, which will contain the inline function definitions.

An inline function is advantageous for several reasons. By inlining a function into the code that calls it, the overhead of a function call is avoided and the optimizer is free to optimize the function in context with the surrounding code.

Inline function expansion is performed when the compiler is invoked with optimization (using the `-x` option) and the `inline` keyword is found in a function definition.

Therefore, by including the appropriate `heronx.h` header file and compiling with the optimizer, the functions contained in the HERON-API header file will be inlined into your application.

In general, when using the optimizer with the inline HERON-API functions, the performance of your application will be improved, but this will also increase code size, as each call to an inline function will result in that code being added at the point of the call.

If your application contains many calls to functions that are defined as inline, you may experience long compile times and large code size due to the many replications of the inline functions. If this is the case you may want to prevent inlining from taking place.

The automatic inlining of functions can be prevented in many ways. By not using the optimizer when compiling, inline functions are kept as function calls. However, this approach also results in a general degradation of performance as no optimization is performed.

Alternatively, by defining the text '`HERON_NO_INLINE_FUNCTIONS`' as shown below,

```
#define HERON_NO_INLINE_FUNCTIONS
```

the automatic inlining of HERON-API functions is disabled. Please note, the definition of this text **must** occur before the inclusion of the HERON-API header file `heronx.h`. By defining this text, you are free to continue using the optimizer when you compile your application.

## FIFO Access Functions

### HeronFIFO *HeronOpenFifo(int fifono, char *s)

For opening a FIFO to use with `HeronTestIo` or `HeronWaitIo`, use the `HeronOpenFifo` function, where `fifono` is a number from 0 through 5 of the HERON FIFO to open.

The argument `s` is a pointer to a string that can contain,

| | |
|---|---|
| `'r'` | - open for reading |
| `'w'` | - open for writing |
| `'u'` | - force the CPU to have priority over this DMA (if used) |
| `'m'` | - force this DMA to have priority over the CPU (default case) |
| `'d'` | - force the use of a dedicated DMA for this fifo |

A simple example is

```
HeronFIFO *fifo;
fifo = HeronOpenFifo(2, "r");
if (fifo == NULL)
{
   printf("Cannot open fifo 2. Error %d\n", heronerr);
   srv_exit(0); /* HUNT ENGINEERING Server/Loader exit */
}
```

This code snippet shows that FIFO number 2 is opened for reading. To open for writing, you would use a specifier character 'w' rather than 'r'. If the call to `HeronOpenFifo` is successful, a FIFO handle is returned. The FIFO handle should be used as a parameter for subsequent calls to HERON-API functions.

A `NULL` value will be returned if the call fails. In this case, the global error variable `heronerr` will specify in more detail what problem was encountered.

#### Dedicated DMA

You can also reserve a DMA channel for exclusive use by a FIFO by using the 'd' option of `HeronOpenFifo`. For example,

```
fifo = HeronOpenFifo(2, "rd");
```

will open FIFO number 2 and will dedicate a DMA engine to it. If there are no DMA engines available (because the user has claimed all of them, or too many other opens have been made with this option) the `HeronOpenFifo` function will return `NULL`. The dedicated option is provided for use in the case that you have a high bandwidth non-blocking transfer that may lose data if a transfer is not ready to start when a FIFO flag indicates data is available.

If the dedicated DMA is not used in this case and several transfers are using the DMAs the delay while waiting for a DMA engine to become free may result in the loss of data. This option does not necessarily provide more bandwidth but guarantees availability of a DMA

engine to perform the transfer.

You can open 12 FIFOs at most. (A HERON module has 6 FIFOs for input and 6 for output.)

### DMA/CPU Priority

Each of the DMA engines on the 'C6201 and 'C6701 can be programmed to have the CPU have priority over the DMA, or to have the DMA have priority over the CPU.

Using the specifier 'u' you can set the CPU to have priority.

Specifier 'm' allows you to specify that DMA has priority (this is the default).

For example:

```
fifo = HeronOpenFifo(2, "ru");
```

Assigns that CPU should have priority over the DMA for this FIFO.

### Error Conditions

The global error variable `heronerr` is used to indicate which error condition has occurred when a call to `HeronOpenFifo` is unsuccessful.

If the argument `fifono` is outside the expected range (0 to 5) the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_INVALID_FIFO_NO`. The values for `heronerr` are defined in the header file for the HERON module type you are using.

If a `NULL` pointer is provided for the string argument `s` the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_NULL_POINTER`.

If the function call fails due to a lack of resources, the global error variable `heronerr` will be set to the value defined by `HERON_NO_RESOURCE`. This can occur if a call to `HeronOpenFifo` is made when all DMA channels have either been claimed by a call to `HeronDmaClaim` or when all DMA channels have been dedicated by previous calls to `HeronOpenFifo`.

If an invalid character is contained in the string pointed to by the argument `s`, the global error variable `heronerr` will be set to `HERON_INVALID_CHARACTER`. The valid characters are 'r', 'w', 'm', 'u' and 'd'.

If both 'r' and 'w' are specified in the string argument the global error variable `heronerr` will be set to `HERON_RW_NOT_SUPPORTED`. It is incorrect to open a FIFO for both reading and writing. The HERON-API will use one FIFO handle for a FIFO opened for reading and another FIFO handle for a FIFO opened for writing. Therefore, for reading and writing the same FIFO number, two calls should be made to `HeronOpenFifo`. One call should be made with the 'r' option only and one call with the 'w' option only.

It is also an error to not specify either 'r' or 'w' as part of the string argument. Doing so will cause the function call to fail with the global error variable `heronerr` set to `HERON_NO_IO_SPECIFIED`.

If a call to `HeronOpenFifo` is made for a FIFO that has already being opened, then the function call will fail, and the global error variable `heronerr` will be set `HERON_FIFO_OPEN`.

### HeronFIFO *HeronSwiOpenFifo(int fifono, char *s, SWI_Obj *swi)

For opening a FIFO that is to use software interrupts to notify of completion, use the `HeronSwiOpenFifo` function.

Where `fifono` can be 0 through 5, and is the number of the HERON FIFO to open.

The argument `s` is a pointer to a string that can contain,

| | |
|---|---|
| `'r'` | - open for reading |
| `'w'` | - open for writing |
| `'u'` | - force the CPU to have priority over this DMA (if used) |
| `'m'` | - force this DMA to have priority over the CPU (default case) |
| `'d'` | - force the use of a dedicated DMA for this fifo |

The argument `swi` is a pointer to a DSP/BIOS SWI object, created in the `.cdb` file you are using. The SWI object pointed to by this argument must define the function that is to be called when a transfer completes on the opened FIFO.

A simple example is

```
HeronFIFO *fifo;
extern far SWI_Obj user_swi;
fifo = HeronSwiOpenFifo(2, "r", &user_swi);
if (fifo == NULL)
{
   printf("Cannot open fifo 2. Error %d\n", heronerr);
   srv_exit(0); /* HUNT ENGINEERING Server/Loader exit */
}
```

This code snippet shows that FIFO number 2 is opened for reading. To open for writing, you would use a specifier character 'w' rather than 'r'. If the call to `HeronSwiOpenFifo` is successful, a FIFO handle is returned. The FIFO handle should be used as a parameter for subsequent calls to HERON-API functions.

A `NULL` value will be returned if the call fails. In this case, the global error variable `heronerr` will specify in more detail what problem was encountered.

#### Using a Software Interrupt

When reading or writing from a FIFO that has been opened with the function `HeronOpenFifo`, the functions `HeronTestIo` and `HeronWaitIo` are used to test or wait for the completion of a FIFO transfer.

By using the function `HeronSwiOpenFifo`, the completion of a transfer can be set up to automatically run a function that you have supplied.

The argument `swi` is a pointer to a DSP/BIOS software interrupt (SWI) object that has been created in the `.cdb` file you are using. When a transfer completes on the FIFO opened by this function, a software interrupt will be posted using the SWI object supplied in the call to `HeronSwiOpenFifo`.

When you create the SWI object in your `.cdb` file, you will need to edit the SWI object properties to set the name of the function that you want to be run on the completion of the FIFO transfer. Also, you must include a reference to this object in your source code so that a pointer to that object can be passed to the function `HeronSwiOpenFifo`.

For example, if using a SWI object named `swi0`, that declares a function called `transfer_done`, the following code would be required to attach that SWI object to the FIFO handle:

```
HeronFIFO *fifo;
extern far SWI_Obj swi0;

main_task() {
      int status;
      fifo = HeronSwiOpenFifo(2, "r", &swi0);
      status = HeronRead(fifo, buf, 1024);
}

void transfer_done() {
      printf("Fifo 2: transfer complete\n");
}
```

**Dedicated DMA**

You can also reserve a DMA channel for exclusive use by a FIFO by using the 'd' option of `HeronSwiOpenFifo`. For example,

```
fifo = HeronSwiOpenFifo(2, "rd");
```

will open FIFO number 2 and will dedicate a DMA engine to it. If there are no DMA engines available (because the user has claimed all of them, or too many other opens have been made with this option) the `HeronSwiOpenFifo` function will return `NULL`. The dedicated option is provided for use in the case that you have a high bandwidth non-blocking transfer that may lose data if a transfer is not ready to start when a FIFO flag indicates data is available.

If the dedicated DMA is not used in this case and several transfers are using the DMAs the delay while waiting for a DMA engine to become free may result in the loss of data. This option does not necessarily provide more bandwidth but guarantees availability of a DMA engine to perform the transfer.

You can open 12 FIFOs at most. (A HERON module has 6 FIFOs for input and 6 for output.)

**DMA/CPU Priority**

Each of the DMA engines on the 'C6201 and 'C6701 can be programmed to have the CPU have priority over the DMA, or to have the DMA have priority over the CPU.

Using the specifier 'u' you can set the CPU to have priority.

Specifier 'm' allows you to specify that DMA has priority (this is the default).

For example:

```
fifo = HeronSwiOpenFifo(2, "ru");
```

Assigns that CPU should have priority over the DMA for this FIFO.

**Error Conditions**

The global error variable `heronerr` is used to indicate which error condition has occurred when a call to `HeronSwiOpenFifo` is unsuccessful.

If the argument `fifono` is outside the expected range (0 to 5) the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_INVALID_FIFO_NO`. The values for `heronerr` are defined in the header file for the HERON module type you are using.

If a `NULL` pointer is provided for the string argument `s` the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_NULL_POINTER`.

If a `NULL` pointer is provided for the SWI_Obj pointer argument `swi` the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_NULL_POINTER`.

If the function call fails due to a lack of resources, the global error variable `heronerr` will be set to the value defined by `HERON_NO_RESOURCE`. This can occur if a call to `HeronSwiOpenFifo` is made when all DMA channels have either been claimed by a call to `HeronDmaClaim` or when all DMA channels have been dedicated by previous calls to `HeronSwiOpenFifo`.

If an invalid character is contained in the string pointed to by the argument `s`, the global error variable `heronerr` will be set to `HERON_INVALID_CHARACTER`. The valid characters are '`r`', '`w`', '`m`', '`u`' and '`d`'.

If both '`r`' and '`w`' are specified in the string argument the global error variable `heronerr` will be set to `HERON_RW_NOT_SUPPORTED`. It is incorrect to open a FIFO for both reading and writing. The HERON-API will use one FIFO handle for a FIFO opened for reading and another FIFO handle for a FIFO opened for writing. Therefore, for reading and writing the same FIFO number, two calls should be made to `HeronSwiOpenFifo`. One call should be made with the '`r`' option only and one call with the '`w`' option only.

It is also an error to not specify either '`r`' or '`w`' as part of the string argument. Doing so will cause the function call to fail with the global error variable `heronerr` set to `HERON_NO_IO_SPECIFIED`.

If a call to `HeronSwiOpenFifo` is made for a FIFO that has already being opened, then the function call will fail, and the global error variable `heronerr` will be set `HERON_FIFO_OPEN`.

### HeronFIFO *HeronSemOpenFifo(int fifono, char *s, SEM_Obj *sem)

For opening a FIFO that is to use semaphores to notify of completion, use the `HeronSemOpenFifo` function.

Where `fifono` can be 0 through 5, and is the number of the HERON FIFO to open.

The argument `s` is a pointer to a string that can contain,

| | |
|---|---|
| `'r'` | - open for reading |
| `'w'` | - open for writing |
| `'u'` | - force the CPU to have priority over this DMA (if used) |
| `'m'` | - force this DMA to have priority over the CPU (default case) |
| `'d'` | - force the use of a dedicated DMA for this fifo |

The argument `sem` is a pointer to a DSP/BIOS SEM object, created in the `.cdb` file you are using. The SEM object pointed to by this argument must define the semaphore that is to be posted when a transfer completes on the opened FIFO.

A simple example is

```
HeronFIFO *fifo;
extern far SEM_Obj user_sem;
fifo = HeronSemOpenFifo(2, "r", &user_sem);
if (fifo == NULL)
{
   printf("Cannot open fifo 2. Error %d\n", heronerr);
   srv_exit(0); /* HUNT ENGINEERING Server/Loader exit */
}
```

This code snippet shows that FIFO number 2 is opened for reading. To open for writing, you would use a specifier character 'w' rather than 'r'. If the call to `HeronSemOpenFifo` is successful, a FIFO handle is returned. The FIFO handle should be used as a parameter for subsequent calls to HERON-API functions.

A `NULL` value will be returned if the call fails. In this case, the global error variable `heronerr` will specify in more detail what problem was encountered.

#### Using a Semaphore

When reading or writing from a FIFO that has been opened with the function `HeronOpenFifo`, the functions `HeronTestIo` and `HeronWaitIo` are used to test or wait for the completion of a FIFO transfer.

By using the function `HeronSemOpenFifo`, the completion of a transfer can be set up to automatically post a semaphore that you have supplied.

The argument `sem` is a pointer to a DSP/BIOS semaphore (SEM) object that has been created in the `.cdb` file you are using. When a transfer completes on the FIFO opened by this function, a semaphore will be posted using the SEM object supplied in the call to `HeronSemOpenFifo`.

You must include a reference in the source code to the SEM object that you have created in the `.cdb` file you are using. This object can then be passed to the function `HeronSemOpenFifo`.

For example, if using a SEM object named `sem0`, the following code would be required to

attach that SEM object to the FIFO handle. By using a call to the function SEM_pend, the completion of the transfer can be detected when the semaphore is posted:

```
HeronFIFO *fifo;
extern far SEM_Obj sem0;

main_task() {
      int status;
      fifo = HeronSemOpenFifo(2, "r", &sem0);
      status = HeronRead(fifo, buf, 1024);
      /* Wait for the transfer to complete */
      SEM_pend(&sem0);
      /* The transfer has completed */
}
```

**Dedicated DMA**

You can also reserve a DMA channel for exclusive use by a FIFO by using the 'd' option of `HeronSemOpenFifo`. For example,

```
fifo = HeronSemOpenFifo(2, "rd");
```

will open FIFO number 2 and will dedicate a DMA engine to it. If there are no DMA engines available (because the user has claimed all of them, or too many other opens have been made with this option) the `HeronSemOpenFifo` function will return `NULL`. The dedicated option is provided for use in the case that you have a high bandwidth non-blocking transfer that may lose data if a transfer is not ready to start when a FIFO flag indicates data is available.

If the dedicated DMA is not used in this case and several transfers are using the DMAs the delay while waiting for a DMA engine to become free may result in the loss of data. This option does not necessarily provide more bandwidth but guarantees availability of a DMA engine to perform the transfer.

You can open 12 FIFOs at most. (A HERON module has 6 FIFOs for input and 6 for output.)

**DMA/CPU Priority**

Each of the DMA engines on the 'C6201 and 'C6701 can be programmed to have the CPU have priority over the DMA, or to have the DMA have priority over the CPU.

Using the specifier 'u' you can set the CPU to have priority.

Specifier 'm' allows you to specify that DMA has priority (this is the default).

For example:

```
fifo = HeronSemOpenFifo(2, "ru");
```

Assigns that CPU should have priority over the DMA for this FIFO.

**Error Conditions**

The global error variable `heronerr` is used to indicate which error condition has occurred when a call to `HeronSemOpenFifo` is unsuccessful.

If the argument `fifono` is outside the expected range (0 to 5) the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_INVALID_FIFO_NO`. The values for `heronerr` are defined in the header file for the HERON module type you are using.

If a `NULL` pointer is provided for the string argument `s` the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_NULL_POINTER`.

If a `NULL` pointer is provided for the SEM_Obj pointer argument `sem` the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_NULL_POINTER`.

If the function call fails due to a lack of resources, the global error variable `heronerr` will be set to the value defined by `HERON_NO_RESOURCE`. This can occur if a call to `HeronSemOpenFifo` is made when all DMA channels have either been claimed by a call to `HeronDmaClaim` or when all DMA channels have been dedicated by previous calls to `HeronSemOpenFifo`.

If an invalid character is contained in the string pointed to by the argument `s`, the global error variable `heronerr` will be set to `HERON_INVALID_CHARACTER`. The valid characters are '`r`', '`w`', '`m`', '`u`' and '`d`'.

If both '`r`' and '`w`' are specified in the string argument the global error variable `heronerr` will be set to `HERON_RW_NOT_SUPPORTED`. It is incorrect to open a FIFO for both reading and writing. The HERON-API will use one FIFO handle for a FIFO opened for reading and another FIFO handle for a FIFO opened for writing. Therefore, for reading and writing the same FIFO number, two calls should be made to `HeronSemOpenFifo`. One call should be made with the '`r`' option only and one call with the '`w`' option only.

It is also an error to not specify either '`r`' or '`w`' as part of the string argument. Doing so will cause the function call to fail with the global error variable `heronerr` set to `HERON_NO_IO_SPECIFIED`.

If a call to `HeronSemOpenFifo` is made for a FIFO that has already being opened, then the function call will fail, and the global error variable `heronerr` will be set `HERON_FIFO_OPEN`.

### HeronFIFO *HeronSioOpenFifo(int fifono, char *s, SIO_Obj *sio, Ptr *pbuffer)

For opening a FIFO that is to use Streaming I/O (SIO), use the `HeronSioOpenFifo` function.

Where `fifono` can be 0 through 5, and is the number of the HERON FIFO to open.

The argument `s` is a pointer to a string that can contain,

`'r'` - open for reading
`'w'` - open for writing

The argument `sio` is a pointer to a DSP/BIOS SIO object, created in the `.cdb` file you are using. The SIO object pointed to by this argument must define the SIO stream that is to be used along subsequent calls to the function `HeronSioRead` or `HeronSioWrite`.

The pointer to pointer argument `pbuffer` is used to return a pointer to an SIO buffer. This SIO buffer that is returned should then be used by the first call to `HeronSioRead` or `HeronSioWrite`.

A simple example is

```
HeronFIFO *fifo;
extern far SIO_Obj user_sio;
unsigned int *data;
fifo = HeronSioOpenFifo(2, "r", &user_sio, (Ptr *)&data);
if (fifo == NULL)
{
   printf("Cannot open fifo 2. Error %d\n", heronerr);
   srv_exit(0); /* HUNT ENGINEERING Server/Loader exit */
}
```

This code snippet shows that FIFO number 2 is opened for reading with SIO. To open for writing, you would use a specifier character 'w' rather than 'r'. If the call to `HeronSioOpenFifo` is successful, a FIFO handle is returned. The FIFO handle should be used as a parameter for subsequent calls to the HERON-API functions `HeronSioRead` or `HeronSioWrite`.

A `NULL` value will be returned if the call fails. In this case, the global error variable `heronerr` will specify in more detail what problem was encountered.

#### Using SIO

When reading or writing from a FIFO that has been opened with the function `HeronOpenFifo`, the functions `HeronRead` and `HeronWrite` are used to start the transfer, and the functions `HeronTestIo` and `HeronWaitIo` are used to test or wait for the completion of the FIFO transfer.

When reading or writing from a FIFO using SIO, the functions `HeronSioRead` and `HeronSioWrite` are used. Unlike the HeronRead and HeronWrite functions, the SIO functions are not asynchronous. That is, each time the SIO functions are called, the function will not return until a buffer has been transferred.

However, rather than being used to directly transfer a buffer to or from a HERON FIFO, the SIO functions are used to pass data between two queues managed by the HERON-API. By using buffer queues, the transfer of data between the FIFOs and the user application is

further de-coupled in comparison to the asynchronous FIFO access functions.

The HERON-API is used to manage the transfer of data between the queues and the FIFOs. It works by repeatedly transferring many buffers of the same size. As soon as one buffer transfer is complete, the next is started.

This leaves the user application free to pass buffers to and from the other end of the two buffer queues at its own rate.

For each FIFO opened to use SIO, two queues are created. One for buffers to be passed to the HERON-API, and one for buffers passed from the HERON-API to the user application.

If the FIFO is opened for reading, the first queue is used to pass empty buffers to the HERON-API, and the second queue is used to receive full buffers from the HERON-API.

Conversely, if the FIFO is opened for writing, the first queue is used to pass full buffers to the HERON-API, and the second queue is used to receive empty buffers back from the HERON-API.

The argument `sio` is a pointer to a DSP/BIOS SIO object that has been created in the `.cdb` file you are using.

When you create the SIO object in your `.cdb` file, you will need to edit the SIO object properties to select the HERON-API SIO driver you wish to use. The template `.cdb` files that are provided with the HERON-API include three User Defined Devices for this purpose.

Device '`Dx0`' is a driver for reading from a FIFO using a trash buffer, device '`Dx1`' is a driver for reading from a FIFO without using a trash buffer and device '`Dx2`' is a driver for writing to a FIFO. The `x` in the device name represents the HERON module type you are using. For example, when using a HERON1 you would use devices '`D10`', '`D11`' and '`D12`'.

In addition to setting the device driver to be used by the SIO stream, you must also set the buffer size (entered in bytes), and the number of buffers to use between the HERON-API and the user application. You must also ensure that a tick is placed in the Allocate Static Buffers check box.

You must include a reference to the SIO object in your source code so that a pointer to that object can be passed to the function `HeronSioOpenFifo`.

The following code example shows how SIO can be used to read data from a FIFO. In this example, repeated transfers are performed using many buffers of the same size. Once the first transfer is started the HERON-API will continue to receive data in the background. As each buffer is filled and passed back to the user application in the second queue, a new transfer is automatically started in the background using a buffer from the first queue. For each time round the loop, one new buffer of data is passed to the user process, and the previous used buffer is returned to the SIO stream.

```
/* Declare a data pointer */
unsigned int *data;

/* Open FIFO 2 for reading using SIO */
handle = HeronSioOpenFifo(2, "r", &sio_obj, (Ptr *)&data);

/* Check the HeronSioOpenFifo call was successful */
if (handle == NULL) exit(0);
```

```
/* Start the transfer and processing of data */
for (i=0;i<N;i++) {
      status = HeronSioRead(handle,(Ptr *)&data);

      /* Do some processing here on the data */
      /* pointed to by the pointer 'data' */
}
```

**Error Conditions**

The global error variable `heronerr` is used to indicate which error condition has occurred when a call to `HeronSioOpenFifo` is unsuccessful.

If the argument `fifono` is outside the expected range (0 to 5) the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_INVALID_FIFO_NO`. The values for `heronerr` are defined in the header file for the HERON module type you are using.

If a `NULL` pointer is provided for the string argument `s` the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_NULL_POINTER`.

If a `NULL` pointer is provided for the SIO_Obj pointer argument `sio` the function call will return `NULL`, and the global error variable `heronerr` will be set to the value defined by `HERON_NULL_POINTER`.

If the function call fails due to a lack of resources, the global error variable `heronerr` will be set to the value defined by `HERON_NO_RESOURCE`. This can occur if a call to `HeronSioOpenFifo` is made when all DMA channels have either been claimed by a call to `HeronDmaClaim` or when all DMA channels have been dedicated by previous calls to `HeronSioOpenFifo`.

If an invalid character is contained in the string pointed to by the argument `s`, the global error variable `heronerr` will be set to `HERON_INVALID_CHARACTER`. The valid characters are '`r`' and '`w`'.

If both '`r`' and '`w`' are specified in the string argument the global error variable `heronerr` will be set to `HERON_RW_NOT_SUPPORTED`. It is incorrect to open a FIFO for both reading and writing. The HERON-API will use one FIFO handle for a FIFO opened for reading and another FIFO handle for a FIFO opened for writing. Therefore, for reading and writing the same FIFO number, two calls should be made to `HeronSioOpenFifo`. One call should be made with the '`r`' option only and one call with the '`w`' option only.

It is also an error to not specify either '`r`' or '`w`' as part of the string argument. Doing so will cause the function call to fail with the global error variable `heronerr` set to `HERON_NO_IO_SPECIFIED`.

If a call to `HeronSioOpenFifo` is made for a FIFO that has already being opened, then the function call will fail, and the global error variable `heronerr` will be set `HERON_FIFO_OPEN`.

### inline int HeronRead(HeronFIFO *handle, void *buffer, unsigned int size)

Each FIFO opened for reading can be read using the `HeronRead` function.

`handle` is the FIFO handle given by a call to the functions `HeronOpenFifo`, `HeronSwiOpenFifo` or `HeronSemOpenFifo`.

The buffer pointed to by `*buffer` is where the data will be written to.

The argument `size` is the number of 32 bit dwords to read.

Using `HeronRead` starts a process that reads data from a FIFO and writes to a buffer. For example:

```
status = HeronRead(fifo, buf, 1024);
```

The `HeronRead` function returns either `HERON_IO_IN_PROGRESS` or an error, but not `HERON_OK`. This is because `HeronRead` only initiates a read but does no reading of FIFOs itself. When a read is successfully submitted, it returns `HERON_IO_IN_PROGRESS`. Or otherwise an error is reported.

### int HeronRestartRead(HeronFIFO *handle, void *buffer)

The function `HeronRestartRead` is similar to the function `HeronRead` in that it can be used to read data from a FIFO opened for reading. The function will read the same number of elements as defined by the last call to `HeronRead` for the specified FIFO handle.

The function `HeronRestartRead` is intended for use in repeated loops where once an initial `HeronRead` has been performed, successive reads are required that will transfer data from a FIFO using the same transfer size.

The function `HeronRestartRead` must be used with care. The function will perform no error checking on the parameters supplied, or on the state of the FIFO handle. It will start a read using a transfer size given in the last call to `HeronRead` (for that handle), and will always return `HERON_IO_IN_PROGRESS` once the read has been started.

`handle` is the FIFO handle given by the `HeronOpenFifo`, `HeronSwiOpenFifo` or `HeronSemOpenFifo` functions. A call to `HeronRead` must have been made for the same FIFO handle, and that read must have completed before this function can be safely called.

The buffer pointed to by `*buffer` is where the data will be written to. This buffer may be a different buffer to the buffer specified in the previous call to `HeronRead`.

The following example illustrates how this function should be used in combination with the function `HeronRead`.

```
/* Start the first read */
status = HeronRead(handle,buffer,size);
/* Check that the transfer was started successfully */
if (status != HERON_IO_IN_PROGRESS) exit(0);
/* Wait for transfer to complete */
status = HeronWaitIo(handle);
/* Check there were no problems */
if (status != HERON_OK) exit(0);

while (1) {
      /* Restart a new read to the same buffer */
```

```
        status = HeronRestartRead(handle,buffer);
        /* Wait for transfer to complete */
        status = HeronWaitIo(handle);
        /* Check there were no problems */
        if (status != HERON_OK) exit(0);
}
```

### inline unsigned int HeronReadWord(HeronFIFO *handle, int *status)

The function `HeronReadWord` is provided as an optimised routine for reading a single word from a FIFO, but **must** be used with care. By using `HeronRead` and `HeronTestIo` or `HeronWaitIo` it is possible to safely read a single word at a time, but there is a large software overhead for the amount of data being transferred.

By using `HeronReadWord` the software overhead of reading a single word with `HeronRead` can be avoided.

The argument `handle` is the FIFO handle given by the `HeronOpenFifo`, `HeronSwiOpenFifo` or `HeronSemOpenFifo` functions.

The function returns a single dword, which is read from the input FIFO specified through the supplied handle. The function will not return until there is an available dword to be read from the FIFO.

The pointer argument `status` is used to record a value to indicate success or failure.

For example:

```
data = HeronReadWord(fifo, &stat);
```

This function must be used carefully. By default the function performs no checking of the validity of the arguments supplied in order to reduce software overheads. When using the function you must be very careful that the handle specified is a valid handle and that there are no outstanding I/Os on that handle. You must also ensure the pointer argument `status` is valid and not `NULL`. Failure to ensure this will result in program failure.

By defining the text '`HERON_INLINE_CHECKING`' as shown below,

```
#define HERON_INLINE_CHECKING
```

checking is enabled in the inline functions of the HERON-API. This define must appear before the inclusion of the HERON-API header file 'heron*x*.h'.

With this defined, the function `HeronReadWord` will check the arguments are valid, and that it is valid to read from the specified handle. Any error condition encountered will result in an error value being written to the integer variable pointed to by the argument `status`. If data is successfully read `status` will be set to `HERON_OK`.

### inline int HeronSioRead(HeronFIFO *handle, Ptr *pbuffer)

Each FIFO opened for reading with Streaming I/O (SIO) can be read using the `HeronSioRead` function.

`handle` is the FIFO handle given by a call to the function `HeronSioOpenFifo`.

For each FIFO opened to use SIO, two queues are created. One queue is used to manage empty buffers to be given to the SIO stream by the user process, and the other queue is used to return full buffers to the user process.

The first call to `HeronSioRead` will start an SIO process that will repeatedly read many buffers of the same size from a FIFO. This function should be used when reading from a high speed FIFO connection, where it is necessary to continuously read buffers of the same size.

The pointer-to-pointer argument `pbuffer` is used to exchange buffers between the user process and the SIO stream. For each call to `HeronSioRead`, a pointer to a pointer to an empty buffer must be supplied in the `pbuffer` argument. When the function returns, the `pbuffer` argument will point to a pointer to a new data buffer. This new data buffer will contain data that has been read from the FIFO and placed in the appropriate SIO queue.

An example of the use of `HeronSioRead` is given below:

```
/* Start the transfer and processing of data */
unsigned int *data;
for (i=0;i<N;i++) {
      status = HeronSioRead(handle,(Ptr *)&data);

      /* Do some processing here on the data */
      /* pointed to by the pointer 'data' */
}
```

If there is no error, the function `HeronSioRead` will block until there is a full buffer to return.

When the function returns, it will return either `HERON_OK` or an error.

**The SIO FIFO Read Trash Buffer**

The template `.cdb` files provided with the HERON-API include two SIO FIFO read drivers. One of these two drivers should be selected when configuring the SIO object used with the functions `HeronSioOpenFifo` and `HeronSioRead`.

The drivers are located in the User Defined Devices section of the `.cdb` file.

The first driver, '`Dx0`' uses a trash buffer, and the second driver '`Dx1`' does not, where the 'x' in the driver name represents the HERON module type you are using.

The HERON-API continuously reads buffers when using SIO. If the queue of empty buffers is empty when the next transfer must be started, the trash buffer driver will continue to read data and will discard the data into a 'trash' buffer. The size of the trash buffer is set to be the same as the size of the data buffers. Therefore, whenever data is written to the trash buffer, the SIO stream will lose an exact buffer of data.

Using the first driver may mean that data is lost, but it is always lost in known amounts.

The second driver (which does not use a trash buffer) will instead pause the reading of the

FIFO until another empty buffer is supplied to the SIO stream. With this driver, data is never discarded into the trash buffer, but the performance provided by this driver is less than that of the first FIFO read driver.

**The Trash Count**

When using the FIFO Read With Trash Buffer driver, the function SIO_ctrl can be used on the associated SIO stream to obtain the 'trash-count'.

The 'trash-count' is a count of how many times the trash buffer was used by the SIO driver. Each time the SIO_ctrl function is used to get the trash-count, the count is reset to 0.

The trash-count is a count of whole buffers, where the buffer length matches that set in the properties of the associated SIO object.

The following code shows how to use the SIO_ctrl function to obtain the trash-count.

```
Unsigned int *data;
Unsigned int trash_count;
Int status;

/* Process N buffers */
for (i=0;i<N;i++) {
     HeronSioRead(handle,(Ptr *)&data);

     /* Process the buffer here */
}

/* Use the SIO_ctrl function to find out how many */
/* times the trash buffer was used. The second */
/* argument must be set to 0 for the SIO_ctrl */
/* call to return the count of how many times */
/* the trash buffer was used. */
status = SIO_ctrl(&sio_obj,0,(Arg)&trash_count);
printf("The trash buffer was used %d times\n",trash_count);
```

## int HeronSioReadAlign(HeronFIFO *handle, int n, Ptr *pbuffer)

A FIFO opened for reading with Streaming I/O (SIO) can be aligned with the function `HeronSioReadAlign`, where `handle` is the FIFO handle given by a call to the function `HeronSioOpenFifo`.

When reading using SIO, the HERON-API will continuously transfer data using buffers of the same size. This buffer size is set when the SIO object is defined in the configuration database file (.cdb file).

When reading from a high speed I/O module such as an A/D module, the buffer size will typically be set to a multiple of the number of A/D channels in use. In this way, each buffer returned through HeronSioRead will have the first channel in the same place in the buffer.

Where the first channel is not located in the first element of each SIO buffer, HeronSioReadAlign can be used to adjust the position of the first channel in the data.

The SIO data is adjusted by reading a smaller amount of data into the next SIO buffer. The

number of words by which the next buffer size is reduced is specified through the integer argument n.

The pointer-to-pointer argument pbuffer is used to exchange buffers between the user process and the SIO stream in a similar way to that of HeronSioRead. For each call to HeronSioReadAlign, a pointer to a pointer to an empty buffer must be supplied in the pbuffer argument, and when the function returns, the pbuffer argument will point to a pointer to a new data buffer.

The new data buffer that is returned will contain data that has been aligned based on the word adjustment value, n.

When the function returns, it will return either HERON_OK or an error.

An example of the use of HeronSioReadAlign is given below. This example assumes that data is being read from an A/D module that has 4 channels. For each channel, one sample is output in a 32-bit word, with two bits of channel identification in bits 0 and 1. The example will use HeronSioReadAlign to align channel 0 into word 0 of each SIO buffer.

```
int n;
unsigned int *data;

HeronSioRead(handle,(Ptr *)&data);
/* Find channel 0 in the data to calculate the shift value */
if      ((data[0] & 0x03) == 0) n = 0;
else if ((data[1] & 0x03) == 0) n = 3;
else if ((data[2] & 0x03) == 0) n = 2;
else if ((data[3] & 0x03) == 0) n = 1;
/* Re-align the SIO stream */
HeronSioReadAlign(handle,n,(Ptr *)&data);
```

### inline int HeronWrite(HeronFIFO *handle, void *buffer, unsigned int size)

Each FIFO opened for write can be written using the `HeronWrite` function.

`handle` is the FIFO handle given by a call to the functions `HeronOpenFifo`, `HeronSwiOpenFifo` or `HeronSemOpenFifo`.

The buffer pointed to by `*buffer` is where the data will be read from.

The argument `size` is the number of 32 bit dwords to written.

Using `HeronWrite` starts a process that reads a buffer and writes the data to a FIFO. For example:

```
status = HeronWrite(fifo, buf, 1024);
```

The `HeronWrite` function returns either `HERON_IO_IN_PROGRESS` or an error, but not `HERON_OK`. This is because `HeronWrite` only initiates a write but does no writing of FIFOs itself. When a write is successfully submitted, it returns `HERON_IO_IN_PROGRESS`. Otherwise an error is reported.

### int HeronRestartWrite(HeronFIFO *handle, void *buffer)

The function `HeronRestartWrite` is similar to the function `HeronWrite` in that it can be used to write to a FIFO opened for writing. The function will write the same number of elements as defined by the last call to `HeronWrite` for the specified FIFO handle.

The function `HeronRestartWrite` is intended for use in repeated loops where once an initial `HeronWrite` has been performed, successive writes are required that will transfer data to a FIFO using the same transfer size.

The function `HeronRestartWrite` must be used with care. The function will perform no error checking on the parameters supplied, or on the state of the FIFO handle. It will start a write using a transfer size given in the last call to `HeronWrite` (for that handle), and will always return `HERON_IO_IN_PROGRESS` once the write has been started.

`handle` is the FIFO handle given by the `HeronOpenFifo`, `HeronSwiOpenFifo` or `HeronSemOpenFifo` functions. A call to `HeronWrite` must have been made for the same FIFO handle, and that write must have completed before this function can be safely called.

The buffer pointed to by `*buffer` is where the data will be read from. This buffer may be a different buffer to the buffer specified in the previous call to `HeronWrite`.

The following example illustrates how this function should be used in combination with the function `HeronWrite`.

```
/* Start the first write */
status = HeronWrite(handle,buffer,size);
/* Check that the transfer was started successfully */
if (status != HERON_IO_IN_PROGRESS) exit(0);
/* Wait for transfer to complete */
status = HeronWaitIo(handle);
/* Check there were no problems */
if (status != HERON_OK) exit(0);

while (1) {
```

```
        /* Restart a new write from the same buffer */
        status = HeronRestartWrite(handle,buffer);
        /* Wait for transfer to complete */
        status = HeronWaitIo(handle);
        /* Check there were no problems */
        if (status != HERON_OK) exit(0);
}
```

### inline int HeronWriteWord(HeronFIFO *handle, unsigned int data)

The function `HeronWriteWord` is provided as an optimised routine for writing a single word to a FIFO, but **must** be used with care. By using `HeronWrite` and `HeronTestIo` or `HeronWaitIo` it is possible to safely write a single word at a time, but there is a large software overhead for the amount of data being transferred.

By using `HeronWriteWord` the software overhead of writing a single word with `HeronWrite` can be avoided.

The argument `handle` is the FIFO handle given by the `HeronOpenFifo`, `HeronSwiOpenFifo` or `HeronSemOpenFifo` functions.

The function writes a single dword to the output FIFO specified through the supplied handle. The function will not return until the write has completed.

For example:

```
stat = HeronWriteWord(fifo, data);
```

This function must be used carefully. By default the function performs no checking of the validity of the arguments supplied in order to reduce software overheads. When using the function you must be very careful that the handle specified is a valid handle and that there are no outstanding I/Os on that handle. Failure to ensure this will result in program failure.

By defining the text '`HERON_INLINE_CHECKING`' as shown below,

```
#define HERON_INLINE_CHECKING
```

checking is enabled in the inline functions of the HERON-API. This define must appear before the inclusion of the HERON-API header file 'heron*x*.h'.

With this defined, the function `HeronWriteWord` will check the arguments are valid, and that it is valid to write with the specified handle. Any error condition encountered will result in an error value returned. If `data` is successfully written the function will return `HERON_OK`.

### inline int HeronSioWrite(HeronFIFO *handle, Ptr *pbuffer)

Each FIFO opened for writing with Streaming I/O (SIO) can be written using the `HeronSioWrite` function.

`handle` is the FIFO handle given by a call to the function `HeronSioOpenFifo`.

For each FIFO opened to use SIO, two queues are created. One queue is used to manage full buffers to be output to the SIO stream from the user process, and the other queue is used to return empty buffers to the user process.

The first call to `HeronSioWrite` will start an SIO process that will repeatedly write many buffers of the same size to a FIFO. This function should be used when writing to a FIFO at a fast rate, and where it is necessary to continuously output buffers of the same size.

The pointer-to-pointer argument `pbuffer` is used to exchange buffers between the user process and the SIO stream. For each call to `HeronSioWrite`, a pointer to a pointer to an full buffer must be supplied in the `pbuffer` argument. When the function returns, the `pbuffer` argument will point to a pointer to an empty data buffer. This empty data buffer can then be used to construct the next buffer to be output.

An example of the use of `HeronSioWrite` is given below:

```
/* Start the transfer and processing of data */
unsigned int *data;
for (i=0;i<N;i++) {
      /* Do some processing here to */
      /* create an output buffer */

      /* Exchange buffer pointers */
      status = HeronSioWrite(handle,(Ptr *)&data);
}
```

If there is no error, the function `HeronSioWrite` will block until there is an empty buffer to return.

When the function returns, it will return either `HERON_OK` or an error.

The template `.cdb` files provided with the HERON-API include one SIO FIFO write driver. This driver must be selected when configuring the SIO object used with the functions `HeronSioOpenFifo` and `HeronSioWrite`.

The driver is located in the User Defined Devices section of the `.cdb` file and is named, '`Dx2`' where the 'x' in the driver name represents the HERON module type you are using.

For example, when using a HERON1, the FIFO write driver would be named '`D12`'.

### int HeronTestIo(HeronFIFO *handle)

You can use the function `HeronTestIo` to check the status of an I/O.

The argument `handle` is the FIFO handle returned by the `HeronOpenFifo` function.

Typically we will want the DSP to do some work, rather than idly wait for the I/O to complete. In such cases, use the non-blocking `HeronTestIo`.

For example:

```
status = HeronRead(fifo, buf, 1024);
while (HeronTestIo(fifo) == HERON_IO_IN_PROGRESS)
{
   /* do useful work */
}
if (status != HERON_OK)
{
   printf("Cannot read fifo 2. Error %d\n", heronerr)
   srv_exit(0);/* HUNT ENGINEERING Server/Loader exit */
}
```

The function `HeronTestIo` only checks if the read (or write) has completed or not, then returns. The return value is either `HERON_OK` if the read (or write) has completed, or `HERON_IO_IN_PROGRESS` if the I/O hasn't completed yet. If an invalid FIFO handle is supplied through the argument `handle`, the function will return `HERON_INVALID_HANDLE`.

### int HeronWaitIo(HeronFIFO *handle)

You can use the function `HeronWaitIo` to block until an I/O is complete.

The argument `handle` is the FIFO handle given by the `HeronOpenFifo` function.

The function is a blocking wait, and is used in the following example:

```
status = HeronRead(fifo, buf, 1024);
if (status == HERON_IO_IN_PROGRESS)
{
   status = HeronWaitIo(fifo);
}
if (status != HERON_OK)
{
   printf("Cannot read fifo 2. Error %d\n", heronerr)
   srv_exit(0); /* HUNT ENGINEERING Server/Loader exit */
}
```

As you can see, we call `HeronWaitIo` only after successfully initiating a read (or write) request. The function will wait until the read (or write) has finished. The function `HeronWaitIo` will return `HERON_OK` or it will report an error, but will never return `HERON_IO_IN_PROGRESS`.

The function `HeronWaitIo` will return `HERON_INVALID_HANDLE` if an invalid FIFO handle is passed through the argument `handle`.

### int HeronClose(HeronFIFO *handle)

The function `HeronClose` will close a previously opened FIFO.

The argument `handle` is the FIFO handle returned by the `HeronOpenFifo`, `HeronSwiOpenFifo`, or `HeronSemOpenFifo` functions.

For a FIFO handle previously opened with `HeronSioOpenFifo`, use the function `HeronSioClose`.

This function releases resources such as DMA channels that can now be used by other parts of a program. For example:

```
HeronClose(fifo);
```

If there is an outstanding I/O on this FIFO, the close will fail and return an error. In this case see `HeronCancelIo`.

The function `HeronClose` will return `HERON_INVALID_HANDLE` if an invalid FIFO handle is passed through the argument `handle`. If an attempt is made to close a FIFO which still has an outstanding I/O, the function will return `HERON_HANDLE_ACTIVE`.

### int HeronSioClose(HeronFIFO *handle)

The function `HeronSioClose` will close a previously opened Streaming I/O FIFO.

The argument `handle` is the FIFO handle returned by `HeronSioOpenFifo` function.

This function releases resources such as DMA channels that can now be used by other parts of a program. For example:

```
HeronSioClose(fifo);
```

The function `HeronSioClose` will return `HERON_INVALID_HANDLE` if an invalid FIFO handle is passed through the argument `handle`.

### int HeronCancelIo(HeronFIFO *handle)

The function `HeronCancelIo` will cancel all I/Os outstanding on a FIFO.

The argument `handle` is the FIFO handle returned by the `HeronOpenFifo`, `HeronSwiOpenFifo`, `HeronSemOpenFifo` or `HeronSioOpenFifo` functions.

This function cancels all I/Os as required by an error handler closing FIFOs after a failure has occurred. For example:

```
HeronCancelIo(fifo);
```

The function `HeronCancelIo` will return `HERON_INVALID_HANDLE` if an invalid FIFO handle is passed through the argument `handle`. Otherwise, after the outstanding I/O has been cancelled the function will return `HERON_OK`.

### int HeronCancelWithFlush(HeronFIFO *handle, unsigned int *done)

The function `HeronCancelWithFlush` will cancel all I/Os outstanding on a FIFO. For a handle opened for reading, the function will also attempt to flush data that is still available for reading from the input FIFO.

In the case of a read, any data successfully flushed from the FIFO is written to the buffer specified by the previous call to `HeronRead`. If the function successfully flushes enough data to complete the previously started read, any remaining data flushed from the FIFO is discarded.

In the case of a write, this function has the same effect as `HeronCancelIo`.

The argument `handle` is the FIFO handle returned by the `HeronOpenFifo`, `HeronSwiOpenFifo`, `HeronSemOpenFifo` or `HeronSioOpenFifo` functions.

The argument `done` is a pointer to a variable of type `unsigned int` that will be used to store the number of dwords successfully read with respect to the previous call to `HeronRead`. This number equals the number of dwords read up until the point of the call to this function, plus the dwords flushed by this function into the specified buffer.

This function cancels all I/Os as required by an error handler closing FIFOs after a failure has occurred. For example:

```
HeronCancelWithFlush(fifo);
```

The function `HeronCancelWithFlush` will return `HERON_INVALID_HANDLE` if an invalid FIFO handle is passed through the argument `handle`. The function will return `HERON_NULL_POINTER` if the pointer argument `done` is invalid (`NULL`).

If the function is attempting to flush the FIFO and there is still data being written to the other side of the FIFO, the function will return `HERON_TOO_MUCH_TO_FLUSH`, indicating that the FIFO could not be successfully flushed. This is detected by the FIFO not becoming empty after reading 80 words, which should always be more data than could be stored in the FIFO.

Otherwise, after the outstanding I/O has been cancelled and the FIFO flushed the function will return `HERON_OK`.

## User DMA Management Functions

### int HeronDmaClaim()

The HERON-API FIFO Access functions use all available DMA channels as default. The DMA channels form resources to the functions `HeronRead` and `HeronWrite`. When the user needs to directly program a DMA using one of the DMA channels the function `HeronDmaClaim` **must** be called to allocate a DMA channel to the user and avoid any conflict with the HERON-API.

The function `HeronDmaClaim` will attempt to allocate the highest DMA channel that is not in use by the HERON_API at the time of the call.

Each time the function is called another DMA channel will be allocated. This function is able to allocate all DMA channels. If the first call to the function `HeronOpenFifo`, `HeronSwiOpenFifo`, `HeronSemOpenFifo` or `HeronSioOpenFifo` is after all DMA channels have been claimed, the call to either function will fail as there are no available resources left. Therefore when calling this function you must consider whether any DMA resources will be required for using the HERON-API FIFO Access functions and limit the calls to `HeronDmaClaim` accordingly.

If the call is unsuccessful the function will return –1. If the call is successful the function will return the number of the DMA channel that has been allocated.

### int HeronDmaFree(int dma)

The function `HeronDmaFree` returns control to the HERON-API, of a DMA channel that was allocated by `HeronDmaClaim`.

The argument `dma` must specify the number of the DMA channel that was allocated by a previous call to `HeronDmaClaim`.

If the DMA channel number passed to the function is invalid, the function will return `HERON_INVALID_DMA_NO`.

If the DMA channel number passed to the function is a valid DMA number, but does not match a previously claimed DMA channel, then the function will return `HERON_FREE_UNCLAIMED_DMA`.

If the call was successful the function returns `HERON_OK`.

### int HeronInstallIsr(unsigned int isr, int int_num)

The function `HeronInstallIsr` should be used to add an interrupt function to a DMA complete interrupt that is associated with a DMA that was claimed with `HeronDmaClaim`.

Alternatively, this function should be used when you must add an interrupt service routine dynamically during program execution, rather than statically through the DSP/BIOS Configuration Database file (the `.cdb` file).

In all other cases, an interrupt should be installed by setting up an appropriate ISR in the DSP/BIOS `.cdb` file.

The argument `isr` is the address of the interrupt service routine to be installed. The argument `int_num` is a number between 0 and 15 specifying the interrupt number of the ISR to be installed.

The function `HeronInstallIsr` will not allow an interrupt service routine to be installed on an invalid interrupt number or on any of the external interrupts (EXT_INT0..3), as these interrupts must only be used internally by the HERON-API. As such, the function will return the error `HERON_INVALID_INTERRUPT` if the argument `int_num` is less than 0, greater than 15, or set to a number between 4 and 7 inclusive.

The function will also not allow an interrupt service routine to be installed on the DMA_INTx interrupts, unless the interrupt is associated to a DMA channel that has already been claimed from the HERON-API by a call to `HeronDmaClaim`.

When installing an interrupt service routine, the function checks the address of the ISR to be installed. If the address is in internal program memory space, the function `HeronInstallIsr` must use a DMA channel to install the ISR. As such, the function automatically shares one of the DMA channels being used by the HERON-API.

The function returns `HERON_OK` if the interrupt function is successfully installed.

The following code illustrates how this function would be used to install an interrupt service routine for a DMA channel that has been claimed with `HeronDmaClaim`.

```
interrupt dma_isr()
{
/* place your ISR code here */
}


main()
{
int dma,int_num;

   if ((dma = HeronDmaClaim()) == -1) {
      printf("Error: call to HeronDmaClaim failed\n");
      exit(0);
   }
```

```
   switch (dma) {
      case 0: int_num = 8;
              break;
      case 1: int_num = 9;
              break;
      case 2: int_num = 11;
              break;
      case 3: int_num = 12;
   }
   HeronInstallIsr((unsigned int)dma_isr, int_num);
}
```

## HERON Serial Bus Functions

### HeronHSB *HeronHsbOpen()

The function `HeronHsbOpen` will open the HSB device for sending and receiving messages. If the call to `HeronHsbOpen` is successful a HSB handle is returned. This handle should be used as a parameter for subsequent calls to HSB message sending and receiving functions.

If an attempt is made to open an already open HSB device, the function will fail by returning a NULL pointer and setting the global error variable `heronerr` to `HERON_HSB_OPEN`.

### int HeronHsbClose(HeronHSB *handle)

The function `HeronHsbClose` closes a previously opened HSB device.

The argument `handle` is the HSB device handle returned by the function `HeronHsbOpen`.

If the function successfully closes the opened HSB device, the function will return `HERON_OK`. The function `HeronHsbClose` will return `HERON_HSB_NOT_OPEN` if an attempt is made to close a HSB device that has not been opened.

### int HeronHsbSendMessage(HeronHSB *handle, int msg_type, int board, int slot, unsigned char *buffer, unsigned int size)

The function `HeronHsbSendMessage` sends a HSB message to the HSB device identified by the `board` and `slot` function arguments.

The function operates on the HSB device specified through the argument `handle`. This device handle is the handle returned by the function `HeronHsbOpen`.

The argument `msg_type` identifies the message type to be used in the send message protocol. The message type is an integer number between 0 and 255.

The integer arguments `board` and `slot` identify the destination for the message. The `board` argument is an integer between 0 and 15 that specifies the board number on which the destination HSB device is located. The `slot` argument is an integer between 0 and 7 that specifies the slot in which the destination HSB device is located.

The `buffer` argument is a pointer to an array of unsigned chars. This array must contain the optional data bytes to be transmitted in the message. The argument `size` defines the number of data bytes to be transmitted. This number must not exceed the buffer size.

If no optional data bytes are to be sent in the message, the argument `buffer` must be set to NULL, and the argument `size` must be set to 0.

The function `HeronHsbSendMessage` automatically generates the first three transmission bytes. The first byte transmitted is the address byte. The address byte is generated from the `board` and `slot` function arguments. The second byte transmitted is the message type. The third byte transmitted is the address of this HSB device, that is, the address of the transmitting end of the message transfer.

If any optional data bytes have been specified these are transmitted next. When all bytes have been successfully transmitted the function will return `HERON_OK`.

If an invalid HSB device handle is passed into HeronHsbSendMessage, the function will return HERON_INVALID_HANDLE.

If a non-zero optional data byte size is specified and a NULL pointer is supplied for the argument buffer, the function will return HERON_NULL_POINTER.

For each message transmitted the receiving end must acknowledge the address byte if the address matches its own address setting. If there is no response to the address byte the function will end transmission and will return HERON_HSB_NO_RESPONSE. This condition may occur if the destination device was incorrectly specified through the arguments board and slot, or if the destination device was unable to receive a message.

If transmission of any one of the optional data bytes is not completed successfully, the function will end transmission and return HERON_HSB_DATA_NOT_SENT.

### int HeronHsbSendMessageEx(HeronHSB *handle, int msg_type, int board, int slot, unsigned char *buffer, unsigned int size, int retries)

The function HeronHsbSendMessageEx is identical to HeronHsbSendMessage, but upon error HERON_HSB_NO_RESPONSE, HeronHsbSendMessageEx will try to send the message again. In effect, what HeronHsbSendMessageEx does is:

```
try = 0; status = HERON_HSB_NO_RESPONSE;
while (try<retries) {
     if (status!=HERON_HSB_NO_RESPONSE) status=HeronHsbSendMessage(…);
}
```

Why would you want to resend a message? When you have a system with EM2 inter-board connectors, it can happen that on one board HSB is ready, but on another board HSB is still busy. If on the board an HSB message is started, targeting a device on the not-yet-ready other board, the HSB message may fail with HERON_HSB_NO_RESPONSE. The busy state usually lasts only shortly, and simply retrying to send the message will work. This function allows to automatically retrying sending the message.

Note that if retries is 0, the function will retry to send the message indefinitely. For function parameters other than retries, please refer to HeronHsbSendMessage.

### int HeronHsbReceiveMessage(HeronHSB *handle, int *msg_type, int board, int slot, unsigned char *buffer, unsigned int size, unsigned int *count)

The function HeronHsbReceiveMessage receives a HSB message from the HSB device identified by the board and slot function arguments.

The function operates on the HSB device specified through the argument handle. This device handle is the handle returned by the function HeronHsbOpen.

The pointer argument msg_type must point to an integer variable into which the message type should be written.

The integer arguments board and slot identify the expected source of the message. The board argument is an integer between 0 and 15 that specifies the board number on which the source HSB device is located. The slot argument is an integer between 0 and 7 that specifies the slot in which the source HSB device is located.

The buffer argument is a pointer to an array of unsigned chars. This array will be used to store any optional data bytes that are received in the message. The argument size must

define the size of the data buffer.

The pointer argument `count` must point to an integer variable that will be used to record the number of optional data bytes received. If no optional data bytes are received, the integer pointed to by `count` will be set to 0.

The function will never store more bytes than is possible given the `size` of the data `buffer`. If the number of bytes received exceeds the size of the data buffer, the remaining bytes are discarded and the `count` will be set to the value specified by the argument `size`.

When all bytes have been successfully received the function will return `HERON_OK`.

If an invalid HSB device handle is passed into `HeronHsbReceiveMessage`, the function will return `HERON_INVALID_HANDLE`.

If a non-zero buffer `size` is specified and a NULL pointer is supplied for the argument `buffer`, the function will return `HERON_NULL_POINTER`. Also, if a NULL pointer is supplied for the argument `count`, the function will return `HERON_NULL_POINTER`.

If a message is received, but the source address contained in the message does not match the expected source specified through the `board` and `slot` arguments, the function will return `HERON_HSB_WRONG_SOURCE`.

### int HeronHsbStartSendMessage(HeronHSB *handle, int board, int slot)

The process of sending a message is made up of three parts. The first part involves setting up the address byte and generating the start condition on the serial bus. The second part is the transmission of all the data bytes, and the third part is ending the message and returning the bus to a free state.

The function `HeronHsbStartSendMessage` will perform the first part of message transmission. As such, this function must be used in combination with the functions `HeronHsbSendMessageData` and `HeronHsbEndOfSendMessage`.

*Please note* where possible, the user should use the function `HeronHsbSendMessage` when sending a message over HSB. The function `HeronHsbStartSendMessage` should only be used where it is not possible to use the standard send message function.

This function operates on the HSB device specified through the argument `handle`. This device handle is the handle returned by the function `HeronHsbOpen`.

The integer arguments `board` and `slot` identify the destination for the message. The `board` argument is an integer between 0 and 15 that specifies the board number on which the destination HSB device is located. The `slot` argument is an integer between 0 and 7 that specifies the slot in which the destination HSB device is located.

The function will generate the address byte from the `board` and `slot` arguments. The address byte is the first byte transmitted in each message.

If the message is successfully started the function will return `HERON_OK`. The user must then make further calls to the functions `HeronHsbSendMessageData` and `HeronHsbEndOfSendMessage`.

If an invalid HSB device handle is passed into `HeronHsbStartSendMessage`, the function will return `HERON_INVALID_HANDLE`.

If no response is received when the address byte is transmitted the function will return `HERON_HSB_NO_RESPONSE`.

### int HeronHsbSendMessageData(HeronHSB *handle, unsigned char *buffer, unsigned int size)

The process of sending a message is made up of three parts. The first part involves setting up the address byte and generating the start condition on the serial bus. The second part is the transmission of all the data bytes, and the third part is ending the message and returning the bus to a free state.

The function `HeronHsbSendMessageData` will perform the second part of message transmission. As such, this function must be used in combination with the functions `HeronHsbStartSendMessage` and `HeronHsbEndOfSendMessage`.

*Please note* where possible, the user should use the function `HeronHsbSendMessage` when sending a message over HSB. The function `HeronHsbSendMessageData` should only be used where it is not possible to use the standard send message function.

This function operates on the HSB device specified through the argument `handle`. This device handle is the handle returned by the function `HeronHsbOpen`.

The `buffer` argument is a pointer to an array of unsigned chars. This array must contain the data bytes to be transmitted. The argument `size` defines the number of data bytes to be transmitted. This number must not exceed the buffer size.

When using this function to send a message, it must be called after successfully starting a message through the function `HeronHsbStartSendMessage`. Once the message has been started this function can be called repeatedly until all message data has been transferred.

For the first call to this function after starting the message, the first two data bytes supplied must be the message type byte and the reply (or source) address. The message type byte is an integer number between 0 and 255. For a further description of the message type please read the earlier section on 'The Message Type' in the chapter on 'HERON Serial Bus Functions'.

The reply address is generated by combining the board ID and slot number for the module on which the HSB send message is being run. The bottom three bits of the address byte must be set to the slot ID (between 0 and 7) and the next four bits of the address must be set to the board ID (between 0 and 15). The following code excerpt shows how the address would be generated for slot 2 of board 0:

```
board = 0; slot = 2;
address = slot;
address = address | (board << 3);
```

After sending the message type and source address, all remaining data bytes are optional data bytes.

For each call to `HeronHsbSendMessageData` the function will return `HERON_OK` when it has successfully transmitted the number of data bytes given by the argument `size`.

If an invalid HSB device handle is passed into `HeronHsbSendMessageData`, the function will return `HERON_INVALID_HANDLE`.

If a NULL pointer is supplied through the argument `buffer` the function will return `HERON_NULL_POINTER`, and if the argument `size` is set to zero, the function will return `HERON_HSB_ZERO_BYTE_COUNT`.

If the function is unable to send all of the specified data bytes, the function will return HERON_HSB_DATA_NOT_SENT.

Note that if you send HSB messages across EM2 inter-board connectors to another board, it may happen that error HERON_HSB_NO_RESPONSE is returned. This is not necessarily a fault. If two HSB messages are sent in quick succession, the HSB on the other board may still be busy when you start the next message. The HSB that is busy will cause the function to return error message HERON_HSB_NO_RESPONSE. In such situations, resend the whole message again (that is, start again with HeronHsbStartSendMessage and then execute again HeronHsbSendMessageData). You may have to retry a few times, but I find that with two boards (connected via EMs) one retry is usually enough.

### int HeronHsbEndOfSendMessage(HeronHSB *handle)

The process of sending a message is made up of three parts. The first part involves setting up the address byte and generating the start condition on the serial bus. The second part is the transmission of all the data bytes, and the third part is ending the message and returning the bus to a free state.

The function HeronHsbEndOfSendMessage will perform the third part of message transmission. As such, this function must be used in combination with the functions HeronHsbStartSendMessage and HeronHsbSendMessageData.

*Please note* where possible, the user should use the function HeronHsbSendMessage when sending a message over HSB. The function HeronHsbEndOfSendMessage should only be used where it is not possible to use the standard send message function.

This function operates on the HSB device specified through the argument handle. This device handle is the handle returned by the function HeronHsbOpen.

This function must be called to end the transmission of a HSB message after successfully starting the message through a call to HeronHsbStartSendMessage, and after one or more successful calls to the function HeronHsbSendMessageData.

If an invalid HSB device handle is passed into HeronHsbEndOfSendMessage, the function will return HERON_INVALID_HANDLE, else the function will end the message, return the serial bus to a free state and return HERON_OK.

### int HeronHsbStartReceiveMessage(HeronHSB *handle, unsigned char *id)

The process of receiving a message is made up of three parts. The first part involves waiting to be addressed by the transmitting end. The second part is the reception of the data bytes, and the third part is ending the message and waiting for the bus to return to a free state.

The function HeronHsbStartReceiveMessage will perform the first part of message reception. As such, this function must be used in combination with the functions HeronHsbReceiveMessageData and HeronHsbEndOfReceiveMessage.

*Please note* where possible, the user should use the function HeronHsbReceiveMessage when receiving a message over HSB. The function HeronHsbStartReceiveMessage should only be used where it is not possible to use the standard receive message function.

This function operates on the HSB device specified through the argument handle. This device handle is the handle returned by the function HeronHsbOpen.

The unsigned char variable pointed to by the argument id is used to store the address of

the message that is being received. For the message to be correct, this address ID must match the expected address of the receiving module. The address value can be generated by combining the board ID and slot number, for the module on which the HSB receive message is being run.

The bottom three bits of the address byte must equal the slot ID (between 0 and 7) and the next four bits of the address must equal the board ID (between 0 and 15). The following code excerpt shows how the address would be generated for slot 2 of board 0:

```
board = 0; slot = 2;
address = slot;
address = address | (board << 3);
```

If the start of the message is successfully detected the function will return `HERON_OK`. The user must then make further calls to the functions `HeronHsbReceiveMessageData` and `HeronHsbEndOfReceiveMessage`.

If an invalid HSB device handle is passed into `HeronHsbStartReceiveMessage`, the function will return `HERON_INVALID_HANDLE`.

If a NULL pointer is supplied through the pointer argument `id` the function will return `HERON_NULL_POINTER`.

### int HeronHsbReceiveMessageData(HeronHSB *handle, unsigned char *buffer, unsigned int size, unsigned int *count)

The process of receiving a message is made up of three parts. The first part involves waiting to be addressed by the transmitting end. The second part is the reception of the data bytes, and the third part is ending the message and waiting for the bus to return to a free state.

The function `HeronHsbReceiveMessageData` will perform the second part of message reception. As such, this function must be used in combination with the functions `HeronHsbStartReceiveMessage` and `HeronHsbEndOfReceiveMessage`.

*Please note* where possible, the user should use the function `HeronHsbReceiveMessage` when receiving a message over HSB. The function `HeronHsbReceiveMessageData` should only be used where it is not possible to use the standard receive message function.

This function operates on the HSB device specified through the argument `handle`. This device handle is the handle returned by the function `HeronHsbOpen`.

The `buffer` argument is a pointer to an array of unsigned chars. This array will be used to store data bytes that are received in the message. The argument `size` must define the size of the data buffer.

The pointer argument `count` must point to an integer variable that will be used to record the number of data bytes received. If no data bytes are received, the integer pointed to by `count` will be set to 0.

The function will never store more bytes than is possible given the `size` of the data `buffer`. If the number of bytes received equals the size of the data buffer, the function will return. If there are any more data bytes to be received, these will be received by calling the function again, or alternatively will be discarded if the next function that is called is `HeronEndOfReceiveMessage`.

When using this function to receive a message, it must be called after successfully detecting

an incoming message through the function HeronHsbStartReceiveMessage. Once the message has been detected this function can be called repeatedly until all message data has been transferred.

For the first call to this function after starting the message, the first two data bytes received will be the message type byte and the reply (or source) address. The message type byte is an integer number between 0 and 255. For a further description of the message type please read the earlier section on 'The Message Type' in the chapter on 'HERON Serial Bus Functions'.

The reply address can be used to form a new destination address if a message must be returned to the sender. After receiving the message type and source address, all remaining data bytes are optional data bytes.

For each call to HeronHsbReceiveMessageData the function will return HERON_OK when it has successfully received a number of data bytes.

The number of data bytes received will be between 0 and the number given by the argument size. When a call to this function returns less bytes than the size argument this indicates that there is no more data to be received for this message. In this case reception of the message must be terminated by a call to HeronHsbEndOfReceiveMessage.

If an invalid HSB device handle is passed into HeronHsbReceiveMessageData, the function will return HERON_INVALID_HANDLE.

If a NULL pointer is supplied through the argument buffer the function will return HERON_NULL_POINTER, and if the argument size is set to zero, the function will return HERON_HSB_ZERO_BYTE_COUNT.

If a NULL pointer is supplied through the pointer argument count the function will return HERON_NULL_POINTER.

### int HeronHsbEndOfReceiveMessage(HeronHSB *handle)

The process of receiving a message is made up of three parts. The first part involves waiting to be addressed by the transmitting end. The second part is the reception of the data bytes, and the third part is ending the message and waiting for the bus to return to a free state.

The function HeronHsbEndOfReceiveMessage will perform the third part of message reception. As such, this function must be used in combination with the functions HeronHsbStartReceiveMessage and HeronHsbReceiveMessageData.

*Please note* where possible, the user should use the function HeronHsbReceiveMessage when receiving a message over HSB. The function HeronHsbEndOfReceiveMessage should only be used where it is not possible to use the standard receive message function.

This function operates on the HSB device specified through the argument handle. This device handle is the handle returned by the function HeronHsbOpen.

This function must be called to end the reception of a HSB message after successfully detecting an incoming message through a call to HeronHsbStartReceiveMessage, and after one or more successful calls to the function HeronHsbReceiveMessageData.

If an invalid HSB device handle is passed into HeronHsbEndOfReceiveMessage, the function will return HERON_INVALID_HANDLE, else the function will end the message and return HERON_OK.

## General Hardware Access Functions

### void HeronConfigOff()

This function causes the HERON module to release its drive from the Config signal.

### void HeronConfigOn()

This function causes the HERON module to assert its drive of the Config signal.

### inline Int HeronDigioIn()

This function returns the value of the digital inputs.

### inline Void HeronDigioOut(int byte)

This function puts the value "byte" onto the digital output lines.

### int HeronModId()

This function returns the Module ID in the bottom four bits of the integer returned and the Carrier ID in the next four bits.

## Uncommitted Module Interconnect (UMI) Functions

### int HeronUmiIn(int line)

This function selects which of the Uncommitted Module Interconnect lines is used to drive the Timer In pin (TINP) of the CPUs' Timer 0. The argument `line` is used to specify the UMI line to connect.

This function is only supported by the HERON1. For all other module types you must use the correct `HeronUmix_In` function, where *x* is the UMI line that you wish to control.

### int HeronUmiOut(int line)

This function selects which of the Uncommitted Module Interconnect lines is driven by the Timer Out pin (TOUT) of the CPU's Timer 0. The argument `line` is used to specify the UMI line to connect.

This function is only supported by the HERON1. For all other module types you must use the correct `HeronUmix_Out` function, where *x* is the UMI line that you wish to control.

### int HeronUmi0_In(int select, unsigned int *umi_value)

This function selects which Timer In pin (TINP) is driven by UMI0. The function is not supported by the HERON1, therefore when using the HERON1 module you must use the function `HeronUmiIn`.

The argument `select` is used to specify whether UMI0 is connected to the Timer In pin of Timer 0 or Timer 1.

The function will return the state of UMI0 line into the integer pointed to by the argument `umi_value`. If this pointer argument is NULL, then the state of UMI0 is ignored.

When connecting UMI0 to the timer input of either Timer, any previous connection to that timer will be overwritten, while the other timer setting will remain unaffected.

The `heronx.h` header file includes three definitions that should be used for the `select` argument. The definitions are listed below.

| | |
|---|---|
| To connect UMI0 to Timer 0, use: | `HERON_UMI_TIMER0` |
| To connect UMI0 to Timer 1, use: | `HERON_UMI_TIMER1` |
| To read the state of UMI0 only: | `HERON_UMI_READ` |

Note, the last choice will only read the state of the UMI0 line using the integer pointed to by the argument `umi_value`. Using this choice will not alter the current connection made by the UMI0 line.

For example to connect to the TINP pin of Timer 1 and to read the state of the UMI0 line into the integer '`umi0_val`', the function call would be performed as follows:

```
status = HeronUmi0_In(HERON_UMI_TIMER1,&umi0_val);
```

If the connection specified by the argument `select` can not be performed the function will return `HERON_INVALID_SELECTION`. If the function is able to make the selected connection to the UMI line, the function returns `HERON_OK`.

### int HeronUmi1_In(int select, unsigned int *umi_value)

This function selects which Timer In pin (TINP) is driven by UMI1. The function is not supported by the HERON1, therefore when using the HERON1 module you must use the function `HeronUmiIn`.

The argument `select` is used to specify whether UMI1 is connected to the Timer In pin of Timer 0 or Timer 1.

The function will return the state of UMI1 line into the integer pointed to by the argument `umi_value`. If this pointer argument is NULL, then the state of UMI1 is ignored.

When connecting UMI1 to the timer input of either Timer, any previous connection to that timer will be overwritten, while the other timer setting will remain unaffected.

The `heronx.h` header file includes three definitions that should be used for the `select` argument. The definitions are listed below.

| | |
|---|---|
| To connect UMI1 to Timer 0, use: | `HERON_UMI_TIMER0` |
| To connect UMI1 to Timer 1, use: | `HERON_UMI_TIMER1` |
| To read the state of UMI1 only: | `HERON_UMI_READ` |

Note, the last choice will only read the state of the UMI1 line using the integer pointed to by the argument `umi_value`. Using this choice will not alter the current connection made by the UMI1 line.

For example to connect to the TINP pin of Timer 1 and to read the state of the UMI1 line into the integer '`umi1_val`', the function call would be performed as follows:

```
status = HeronUmi1_In(HERON_UMI_TIMER1,&umi1_val);
```

If the connection specified by the argument `select` can not be performed the function will return `HERON_INVALID_SELECTION`. If the function is able to make the selected connection to the UMI line, the function returns `HERON_OK`.

### int HeronUmi2_In(int select, unsigned int *umi_value)

This function selects which Timer In pin (TINP) is driven by UMI2. The function is not supported by the HERON1, therefore when using the HERON1 module you must use the function `HeronUmiIn`.

The argument `select` is used to specify whether UMI2 is connected to the Timer In pin of Timer 0 or Timer 1.

The function will return the state of UMI2 line into the integer pointed to by the argument `umi_value`. If this pointer argument is NULL, then the state of UMI2 is ignored.

When connecting UMI2 to the timer input of either Timer, any previous connection to that timer will be overwritten, while the other timer setting will remain unaffected.

The `heronx.h` header file includes three definitions that should be used for the `select` argument. The definitions are listed below.

| | |
|---|---|
| To connect UMI2 to Timer 0, use: | `HERON_UMI_TIMER0` |
| To connect UMI2 to Timer 1, use: | `HERON_UMI_TIMER1` |
| To read the state of UMI2 only: | `HERON_UMI_READ` |

Note, the last choice will only read the state of the UMI2 line using the integer pointed to by the argument `umi_value`. Using this choice will not alter the current connection made by the UMI2 line.

For example to connect to the TINP pin of Timer 1 and to read the state of the UMI2 line into the integer '`umi2_val`', the function call would be performed as follows:

```
status = HeronUmi2_In(HERON_UMI_TIMER1,&umi2_val);
```

If the connection specified by the argument `select` can not be performed the function will return `HERON_INVALID_SELECTION`. If the function is able to make the selected connection to the UMI line, the function returns `HERON_OK`.

### int HeronUmi3_In(int select, unsigned int *umi_value)

This function selects which Timer In pin (TINP) is driven by UMI3. The function is not supported by the HERON1, therefore when using the HERON1 module you must use the function `HeronUmiIn`.

The argument `select` is used to specify whether UMI3 is connected to the Timer In pin of Timer 0 or Timer 1.

The function will return the state of UMI3 line into the integer pointed to by the argument `umi_value`. If this pointer argument is NULL, then the state of UMI3 is ignored.

When connecting UMI3 to the timer input of either Timer, any previous connection to that timer will be overwritten, while the other timer setting will remain unaffected.

The `heronx.h` header file includes three definitions that should be used for the `select` argument. The definitions are listed below.

| | |
|---|---|
| To connect UMI3 to Timer 0, use: | `HERON_UMI_TIMER0` |
| To connect UMI3 to Timer 1, use: | `HERON_UMI_TIMER1` |
| To read the state of UMI3 only: | `HERON_UMI_READ` |

Note, the last choice will only read the state of the UMI3 line using the integer pointed to by the argument `umi_value`. Using this choice will not alter the current connection made by the UMI3 line.

For example to connect to the TINP pin of Timer 1 and to read the state of the UMI3 line into the integer '`umi3_val`', the function call would be performed as follows:

```
status = HeronUmi3_In(HERON_UMI_TIMER1,&umi3_val);
```

If the connection specified by the argument `select` can not be performed the function will return `HERON_INVALID_SELECTION`. If the function is able to make the selected connection to the UMI line, the function returns `HERON_OK`.

### int HeronUmi0_Out(int select)

This function selects how UMI0 is driven. The function is not supported by the HERON1, therefore when using the HERON1 module you must use the function `HeronUmiOut`.

There are five options for how each UMI line is driven. A UMI line may be driven by the Timer Out pin (TOUT) of Timer 0, by the Timer Out pin of Timer 1, it may be set low, or set high, or not driven. The argument `select` is used to specify which of these five options is required.

The `heronx.h` header file includes five definitions that should be used for the `select` argument. The definitions are listed below.

| | |
|---|---|
| To drive UMI0 by Timer 0, use: | `HERON_UMI_TIMER0` |
| To drive UMI0 by Timer 1, use: | `HERON_UMI_TIMER1` |
| To drive UMI0 low, use: | `HERON_UMI_LOGIC0` |
| To drive UMI0 high, use: | `HERON_UMI_LOGIC1` |
| To set UMI0 to be not driven, use: | `HERON_UMI_UNDRIVEN` |

For example to drive UMI0 with the TOUT pin of Timer 0, the function call would be performed as follows:

```
status = HeronUmi0_Out(HERON_UMI_TIMER0);
```

If the connection specified by the argument `select` can not be performed the function will return `HERON_INVALID_SELECTION`. If the function is able to make the selected connection to the UMI line, the function returns `HERON_OK`.

### int HeronUmi1_Out(int select)

This function selects how UMI1 is driven. The function is not supported by the HERON1, therefore when using the HERON1 module you must use the function `HeronUmiOut`.

There are five options for how each UMI line is driven. A UMI line may be driven by the Timer Out pin (TOUT) of Timer 0, by the Timer Out pin of Timer 1, it may be set low, or set high, or not driven. The argument `select` is used to specify which of these five options is required.

The `heronx.h` header file includes five definitions that should be used for the `select` argument. The definitions are listed below.

| | |
|---|---|
| To drive UMI1 by Timer 0, use: | `HERON_UMI_TIMER0` |
| To drive UMI1 by Timer 1, use: | `HERON_UMI_TIMER1` |
| To drive UMI1 low, use: | `HERON_UMI_LOGIC0` |
| To drive UMI1 high, use: | `HERON_UMI_LOGIC1` |
| To set UMI1 to be not driven, use: | `HERON_UMI_UNDRIVEN` |

For example to drive UMI1 with the TOUT pin of Timer 0, the function call would be performed as follows:

```
status = HeronUmi1_Out(HERON_UMI_TIMER0);
```

If the connection specified by the argument `select` can not be performed the function will return `HERON_INVALID_SELECTION`. If the function is able to make the selected

connection to the UMI line, the function returns HERON_OK.

### int HeronUmi2_Out(int select)

This function selects how UMI2 is driven. The function is not supported by the HERON1, therefore when using the HERON1 module you must use the function HeronUmiOut.

There are five options for how each UMI line is driven. A UMI line may be driven by the Timer Out pin (TOUT) of Timer 0, by the Timer Out pin of Timer 1, it may be set low, or set high, or not driven. The argument select is used to specify which of these five options is required.

The heronx.h header file includes five definitions that should be used for the select argument. The definitions are listed below.

| | |
|---|---|
| To drive UMI2 by Timer 0, use: | HERON_UMI_TIMER0 |
| To drive UMI2 by Timer 1, use: | HERON_UMI_TIMER1 |
| To drive UMI2 low, use: | HERON_UMI_LOGIC0 |
| To drive UMI2 high, use: | HERON_UMI_LOGIC1 |
| To set UMI2 to be not driven, use: | HERON_UMI_UNDRIVEN |

For example to drive UMI2 with the TOUT pin of Timer 0, the function call would be performed as follows:

```
status = HeronUmi2_Out(HERON_UMI_TIMER0);
```

If the connection specified by the argument select can not be performed the function will return HERON_INVALID_SELECTION. If the function is able to make the selected connection to the UMI line, the function returns HERON_OK.

### int HeronUmi3_Out(int select)

This function selects how UMI3 is driven. The function is not supported by the HERON1, therefore when using the HERON1 module you must use the function HeronUmiOut.

There are five options for how each UMI line is driven. A UMI line may be driven by the Timer Out pin (TOUT) of Timer 0, by the Timer Out pin of Timer 1, it may be set low, or set high, or not driven. The argument select is used to specify which of these five options is required.

The heronx.h header file includes five definitions that should be used for the select argument. The definitions are listed below.

| | |
|---|---|
| To drive UMI3 by Timer 0, use: | HERON_UMI_TIMER0 |
| To drive UMI3 by Timer 1, use: | HERON_UMI_TIMER1 |
| To drive UMI3 low, use: | HERON_UMI_LOGIC0 |
| To drive UMI3 high, use: | HERON_UMI_LOGIC1 |
| To set UMI3 to be not driven, use: | HERON_UMI_UNDRIVEN |

For example to drive UMI3 with the TOUT pin of Timer 0, the function call would be performed as follows:

```
status = HeronUmi3_Out(HERON_UMI_TIMER0);
```

If the connection specified by the argument `select` can not be performed the function will return `HERON_INVALID_SELECTION`. If the function is able to make the selected connection to the UMI line, the function returns `HERON_OK`.

### int HeronEnableUmiInt(int umi, int polarity)

This function is used to enable an interrupt on the selected UMI line. This function is not supported by the HERON1.

The argument `umi` selects the UMI line, (from UMI 0 to UMI 3), for which an interrupt will be enabled. The interrupt can be configured as a rising edge interrupt or a falling edge.

To set the interrupt to be a rising edge on the selected UMI line, set the argument `polarity` to `1`. To set the interrupt to be a falling edge on the selected UMI line, set the argument polarity to `0`.

When the interrupt occurs, the HERON-API posts either a DSP/BIOS semaphore, or DSP/BIOS software interrupt (SWI). In order to do this the function must be used in conjunction with either `HeronInstallUmiSwi` or `HeronInstallUmiSemaphore`.

If you require a DSP/BIOS software interrupt (SWI) to occur when an interrupt is received, you should install a SWI using the function `HeronInstallUmiSwi` for the same UMI line.

Alternatively, if you require a DSP/BIOS task to block until the interrupt is received, you should install a semaphore using the function `HeronInstallUmiSemaphore` for the same UMI line.

If the argument `umi` is not in the range `0` to `3`, or if the argument `polarity` is not set to `0` or `1`, the function will return `HERON_INVALID_SELECTION`. If the function call is successful, the function returns `HERON_OK`.

### int HeronDisableUmiInt(int umi)

This function is used to disable an interrupt that was previously enabled with a call to `HeronEnableUmiInt`. This function is not supported by the HERON1.

The argument `umi` selects the UMI line for which the interrupt is to be disabled.

If the argument `umi` is not in the range `0` to `3` the function will return `HERON_INVALID_SELECTION`, else the function will return `HERON_OK`.

### int HeronInstallUmiSwi(int umi, SWI_Obj *swi)

This function is used to install a DSP/BIOS software interrupt (SWI) on to a UMI interrupt. This function is not supported by the HERON1.

The argument `umi` selects the UMI line for which the SWI will be installed. When an interrupt occurs on the associated UMI line, the HERON-API will post a software interrupt using the SWI_Obj object provided by the argument `swi`.

Once a software interrupt has been installed with this function, the interrupt can be enabled with a call to the function `HeronEnableUmiInt`.

If the argument `umi` is not in the range `0` to `3` the function will return `HERON_INVALID_SELECTION`, else the function will return `HERON_OK`.

### int HeronInstallUmiSemaphore(int umi, SEM_Obj *sem)

This function is used to install a DSP/BIOS semaphore on to a UMI interrupt. This function is not supported by the HERON1.

The argument `umi` selects the UMI line for which the semaphore will be installed. When an interrupt occurs on the associated UMI line, the HERON-API will post a semaphore using the SEM_Obj object provided by the argument `swi`.

Once a semaphore has been installed with this function, the interrupt can be enabled with a call to the function `HeronEnableUmiInt`.

If the argument `umi` is not in the range `0` to `3` the function will return `HERON_INVALID_SELECTION`, else the function will return `HERON_OK`.

## Where is the Library?

The library is installed as part of the HUNT ENGINEERING API (host-side API) installation program. It will appear in the directory that you chose for the API installation, in the sub-directory `\heron_api\lib`. The default location is `\heapi\heron_api\lib`. The HUNT ENGINEERING API installation actually sets an environmental variable `HEAPI_DIR` to show which installation directory you chose, so the library can be addressed as `%HEAPI_DIR%\heron_api\lib`.

When selecting a library to use in your application, please refer to the section 'Selecting the Right Library for you Application'.

| | |
|---|---|
| Herons.lib | 'C6000 library. Data access is near, and code access is near. |
| Heronl0.lib | 'C6000 library. Data access is far, and code access is near. |
| Heronl1.lib | 'C6000 library. Data access is near, and code access is far. |
| Heronl3.lib | 'C6000 library. Data access is far, and code access is far. |

## Where is the Source Code?

The source code for all functions in the HERON-API library can be found in the directory that you chose for the API installation, in the sub-directory `\heron_api\src`.

You should not change the HERON-API source code yourself, as that will prevent you from easily using later versions provided by HUNT ENGINEERING. This source code is provided to you for reference only. If you do make changes to this source code please inform HUNT ENGINEERING what you have done so that it could be included in the released software.

## Can I Compile the Library?

For each module type there is a sub-directory below the directory `\heron_api\src`. For example, for the HERON1 module type the HERON-API source is located in the sub-directory `\heron_api\src\heron1`. In each of these sub-directories there is a batch file called `heron_api.bat`. This batch file actually calls another batch file, named `makelib6.bat`. This batch file does the actual work but it takes some parameters. The `heron_api.bat` merely sets these parameters and then invokes the `makelib6.bat` batch file.

Again, it is not recommended that you change the HERON-API sources yourself as this could prevent you from easily using later versions provided by HUNT ENGINEERING.

On the HUNT ENGINEERING CD there are some examples of using the HERON_API. Use the CD front end program to select "software examples". There are some pure HERON-API examples in the directory "heron_api_examples", and a general starting point example in the "starting_development" directory. The IO board examples in the directory "board_examples" also show the use of the HERON-API.

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/index.htm on the HUNT ENGINEERING web site. This includes FAQs, latest product, software and documentation updates etc. Or contact your local supplier - if you are unsure of details please refer to www.hunteng.co.uk for the list of current re-sellers.

HUNT ENGINEERING technical support can be contacted by emailing support@hunteng.demon.co.uk, calling the direct support telephone number +44 (0)1278 760775, or by calling the general number +44 (0)1278 760188 and choosing the technical support option.

## Processor Interrupts

When transferring data to or from FIFOs the HERON-API will use DMAs. For the HERON1, the HERON-API provides two different methods for using the DMA channels that are available. The first (default) method is referred to as floating DMA, and the second method is dedicated DMA.

The HERON-API implements a group of floating DMA channels. The floating DMA channels are used on a first come first serve basis to meet the needs of any FIFO transfer that is in progress. Each time a block needs to be transferred, the next free DMA resource is used, leaving one less resource for the next transfer that comes along. When there is a transfer to be performed and all DMA channels are in use, the transfer will wait until a DMA channel becomes free.

When transferring data using the floating DMA method, the HERON-API will divide the whole DMA transfer into blocks. For each of these blocks a processor interrupt is required by the HERON-API in order to update how much has been done and to restart a DMA for the next block.

There are several ways in which a user application can affect the processor interrupts required by the HERON-API when performing a DMA transfer. Interrupts can be stopped by switching of global interrupts in the CSR register of the CPU and by writing tight loops of code that result in a sequence of assembly instructions that is un-interruptable.

For a blocking data-transfer, such as the transfer from one processor to another, disabling interrupts in this way will mean that DMA transfers are halted by the HERON-API until an interrupt occurs and the DMA can continue and as a result the performance of the blocking transfer will drop.

However, for a non-blocking transfer, such as the transfer from a high speed A to D module to the processor, this will typically result in the loss of data.

It is therefore very important to consider how interrupts are affected by any program you write.

Firstly, it is vital that interrupts are globally enabled while using the HERON-API for the HERON1. Please ensure that if interrupts are disabled globally, that this is only done for a very short time (for no more than a few C instructions), before interrupts are re-enabled. To be complete safe, it is recommend that global interrupts are never disabled.

Secondly, when writing highly optimised loops that contain a small number of instructions that un-interruptable loops are avoided. If an un-interruptable loop is created you must ensure that the processor will not remain in that loop for long, in order that interrupts can continue once it has completed, and the HERON-API can continue DMA transfers.

### Dedicated DMA

The `HeronOpenFifo`, `HeronSwiOpenFifo` and `HeronSemOpenFifo` functions of the HERON-API provide a switch to open a FIFO with a dedicated DMA.

The dedicated switch in the call to `HeronOpenFifo`, `HeronSwiOpenFifo` or `HeronSemOpenFifo` enables a DMA channel to be allocated to the FIFO specified in the function call.

This means that for the specified FIFO, any transfer will continue straight away using the resource that it was allocated. This removes the possibility that a floating DMA channel was not available and therefore removes any potential delay to that transfer.

The use of dedicated DMA channels is therefore recommended for high performance transfers such as those from a high bandwidth A to D module.

For the Streaming I/O function `HeronSioOpenFifo`, the FIFO will always being opened to use dedicated DMA.

## Processor Interrupts

When transferring data to or from FIFOs the HERON-API will use DMAs. For the HERON4, the HERON-API provides two different methods for using the DMA channels that are available. The first (default) method is referred to as floating DMA, and the second method is dedicated DMA.

The HERON-API implements a group of floating DMA channels. The floating DMA channels are used on a first come first serve basis to meet the needs of any FIFO transfer that is in progress. Each time a block needs to be transferred, the next free DMA resource is used, leaving one less resource for the next transfer that comes along. When there is a transfer to be performed and all DMA channels are in use, the transfer will wait until a DMA channel becomes free.

When transferring data using the floating DMA method, the HERON-API will divide the whole DMA transfer into blocks. For each of these blocks a processor interrupt is required by the HERON-API in order to update how much has been done and to restart a DMA for the next block.

There are several ways in which a user application can affect the processor interrupts required by the HERON-API when performing a DMA transfer. Interrupts can be stopped by switching of global interrupts in the CSR register of the CPU and by writing tight loops of code that result in a sequence of assembly instructions that is un-interruptable.

For a blocking data-transfer, such as the transfer from one processor to another, disabling interrupts in this way will mean that DMA transfers are halted by the HERON-API until an interrupt occurs and the DMA can continue and as a result the performance of the blocking transfer will drop.

However, for a non-blocking transfer, such as the transfer from a high speed A to D module to the processor, this will typically result in the loss of data.

It is therefore very important to consider how interrupts are affected by any program you write.

Firstly, it is vital that interrupts are globally enabled while using the HERON-API for the HERON4. Please ensure that if interrupts are disabled globally, that this is only done for a very short time (for no more than a few C instructions), before interrupts are re-enabled. To be complete safe, it is recommend that global interrupts are never disabled.

Secondly, when writing highly optimised loops that contain a small number of instructions that un-interruptable loops are avoided. If an un-interruptable loop is created you must ensure that the processor will not remain in that loop for long, in order that interrupts can continue once it has completed, and the HERON-API can continue DMA transfers.

### Dedicated DMA

The `HeronOpenFifo`, `HeronSwiOpenFifo` and `HeronSemOpenFifo` functions of the HERON-API provide a switch to open a FIFO with a dedicated DMA.

The dedicated switch in the call to `HeronOpenFifo`, `HeronSwiOpenFifo` or `HeronSemOpenFifo` enables a DMA channel to be allocated to the FIFO specified in the function call.

This means that for the specified FIFO, any transfer will continue straight away using the resource that it was allocated. This removes the possibility that a floating DMA channel was not available and therefore removes any potential delay to that transfer.

The use of dedicated DMA channels is therefore recommended for high performance transfers such as those from a high bandwidth A to D module.

For the Streaming I/O function `HeronSioOpenFifo`, the FIFO will always being opened to use dedicated DMA.