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
URL: http://www.hunteng.co.uk
http://www.hunt-dsp.com

HUNT ENGINEERING

API

Reference Manual

Document Rev C
API software Rev 1.9.9
P.Warnes / J.Thie 15-07-05

COPYRIGHT

This documentation and the product it is supplied with are Copyright HUNT ENGINEERING 1999-2001. All rights reserved. HUNT ENGINEERING maintains a policy of continual product development and hence reserves the right to change product specification without prior warning.

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.

Revision History

API ver 1.0
Pre release version, internal release only

API ver 1.1
First release version.

API ver 1.2
First release to be put on HUNT ENGINEERING CD. Added VxDs for Win 95. Several Bug fixes, including asynchronous I/O fixes for Pll C Winserver. Installation completely changed, and troubleshooting options added.

API ver 1.3
VxWorks Support added.
HEPC6 support added.
Installation made to use uncompressed files.

API ver 1.4
HERON support added.
Board specific information moved to appendices

API ver 1.5
HERON VxWorks support added.
LINUX support added.

API Doc rev B
Complete re-organisation of manual for easier navigation.

API ver 1.6

Support for Serial Bus added.

API ver 1.7

Support for RTOS-32 added, manual re-structured.

API ver 1.8

Support for HEPC9 added.

API ver 1.9

Support for Windows 2000 and XP added (WDM drivers).

API ver 1.9.6

New functions documented (HSB Ex functions, HeGetDeviceInfo)

API ver 1.9.7

RTOS32 support updated to full 1.9.7
VxWorks support updated to full 1.9.7
Linux support updated to full 1.9.7

API Doc rev C
Rewritten HSB section
Rewritten Error Codes section

TABLE OF CONTENTS

REVISION HISTORY 3

WHY DO I NEED THE API? 6

DEVELOPMENT SYSTEMS AND TARGET SYSTEMS 6

WHAT IS THE API? 7

PLATFORM INDEPENDENCE 7

CONSISTENT INTERFACE 8

HOW IS IT DONE? 9

API LIBRARIES 9

DEVICE DRIVERS 10

WHAT IS SUPPORTED? 11

PLATFORMS 11

Development & Target systems 11

Target Only systems 11

HUNT ENGINEERING Host Interface Boards 11

INSTALLATION 13

API INTERFACE: CONCEPTS 14

WRITING PROGRAMS THAT USE THE API 14

DEVICES 14

OPEN AND CLOSE A DEVICE 14

ASYNCHRONOUS ACCESS 15

Writing 15

Reading 16

BUFFER ALLOCATION (AND THE "HUGE" MEMORY MODEL) 16

A SIMPLE PROGRAM THAT USES THE API INTERFACE 17

MAINTAINING PLATFORM INDEPENDENCE 19

LOCKING OF DEVICES FOR EXCLUSIVE ACCESS 20

Disabling Of Lock Files 21

API INTERFACE: DATA STRUCTURES 22

USING THE HANDLES 22

HE_HANDLE 22

HE_MEMHANDLE 23

HE_IOSTATUS 23

API INTERFACE: FUNCTIONS 24

HeOpen() 24

HeOpen1() 25

HeOpenS() 25

HeClose() 26

HeRead() 26

HeWrite() 26

HeDelay() 27

HeReset() 27

HeReset1() 27

HeInitIoStatus 28

HeWaitForIo() 28

HeTestIo() 28

HeErr2Text() 29

HeGetIoGranularity() 29

HeGetBoardInfo() (HERON carriers only) 29

HeAlloc() ... *30*
HeFree() ... *30*
HeLock() ... *31*
HeUnlock() ... *31*
HeConfig() ... *31*
HeJtagWrite() ... *32*
HeJtagRead() ... *32*
HeGetLastOsError() ... *32*
HeHSBSendMessageEx() ... *33*
HeHSBReceiveMessageEx() ... *33*
HeHSBStartSendMessageEx() ... *34*
HeHSBSendMessageDataEx() ... *35*
HeHSBEndOfSendMessageEx() ... *35*
HeHSBStartReceiveMessageEx() ... *36*
HeHSBReceiveMessageDataEx () ... *36*
HeHSBEndOfReceiveMessageEx() ... *37*
HeHSBInit() ... *37*
HeHSBMaster() ... *37*
HeHSBSlave() ... *38*
HeHSBListen() ... *38*
HeHSBFlush() ... *38*
HeGetDeviceInfo() ... *39*
STATUS CODES. ... 39
Notes ... *47*

HERON SERIAL BUS (HSB) ... **49**
INTRODUCTION ... 49
HSB IDs or identifiers ... *49*
HSB speed ... *49*
HUNT HSB Protocol ... *49*
Non HSB Protocol Messages ... *50*
Accessing the Heron Serial Bus ... *50*
Level 3 Serial Bus functions ... *50*
Level 2 Serial Bus functions ... *52*
Level 1 Serial Bus Functions (lowest level) ... *54*
HERON Serial Bus Message Types ... *56*

JTAG ... **57**

CODE COMPOSER STUDIO PLUGINS ... **59**
WHAT IS A PLUGIN? ... 59
THE ‘RESET SYSTEM’ PLUGIN ... 59
What does it do? ... *59*
How do I start it? ... *59*
How do I use it? ... *60*
Options ... *60*
THE ‘CREATE NEW HERON-API PROJECT’ PLUGIN ... 61
What does it do? ... *61*
How do I start it? ... *62*
How do I use it? ... *62*
Options ... *64*
What actions does the plugin perform? ... *65*

EXAMPLE PROGRAMS ... **67**

TECHNICAL SUPPORT ... **68**

Why do I need the API?

The objective of supplying the HUNT ENGINEERING API is to provide a common interface between software on the host machine and all HUNT ENGINEERING host boards. This interface is also common for all of the supported host machine operating systems, which are listed in the tables below.

HUNT ENGINEERING support a set of development tools for their products, and users often need to gain efficient access to HUNT ENGINEERING module carriers. The supported way of doing this is the API interface described in this document.

This document discusses only the API itself. The platform and board independent interface is discussed, and we look in detail at the functions supported. In this document we look only at things in the API that are the same across the supported operating systems and boards.

Installation of the API is discussed in a per-operating-system document (in pdf format). At the time of writing this document there are 4 such documents:

- 'api – windows (installation & user manual)' for Windows 95/98/ME/NT/W2K,
- 'api – linux (installation & user manual) for LINUX,
- 'api – vxworks (installation & user manual) for VxWorks,
- 'api – rtos-32 (installation & user manual) for RTOS-32.

Additional supported operating systems will have a similarly named installation and user manual dedicated to them.

Development Systems and Target Systems

There are two types of API support: support for Development Systems, and support for Target Systems.

An Operating System is a Target System if a system can be interfaced with it via the API standard interface. An Operating System is a Development System if the system can be interfaced with it via the API standard interface, and if there are development tools available for that operating system. The development tools implied are the TI compiler/linker/assembler, 3L Parallel C, Server/Loader, and Code Composer.

The API uses a simple asynchronous communications model, which we briefly discuss here – for full details refer to later sections of this user manual.

First the device must be claimed by performing an `HeOpen()` on the device. This function takes a board identifier, (to specify which type of HUNT ENGINEERING module carrier is to be opened), a board number (to specify which of the boards in the system) and a device number (to specify which resource on that board). The function returns a file descriptor if the call is successful, or else an API error code.

If the system is to be booted, a reset must be performed using `HeReset()` on the file descriptor given by the open call.

A write to the FIFO can be started using the `HeWrite()` function on the file descriptor.

A read from the FIFOs can be started using the `HeRead()` function on the file descriptor.

Both the read and write functions will return immediately, with either a successful status, an in-progress status or an error. The in-progress status allows the host side application to continue processing of previous data while the hardware access is taking place.

The status of an I/O can be tested at any time using the `HeTestforIO()` function, or the host program can be blocked until it is complete by using the `HeWaitforIO()` function.

Platform Independence

The API is available on a number of platforms, it is possible that these use different definitions of certain data types, for example the size of an integer, how many bits are in a pointer, how large an area of memory a pointer can handle, etc. To provide Application level consistency the API introduces a set of Variable Types that allow a programmer to avoid platform specific data types, examples are:

eight bit unsigned	`HE_BYTE`
32 bit unsigned	`HE_DWORD`
32-bit signed	`HE_INT32`
Pointer to eight bit oriented data	`HE_PBYTEBUFFER`
Boolean	`HE_BOOLEAN`

This API is provided for all platforms and saves the developer from having to modify their source code to handle such platform differences.

An example of the power of this is on DOS where in order for a pointer to transparently address more than 64Kbytes it must be declared "huge". The API does this automatically. It does affect performance but this is more than outweighed by the portability benefits. It is also the case that some devices require 32 bit aligned data for DMA (e.g. PCI Master Mode), using the defined pointers and API memory allocation routines this is handled automatically.

Consistent interface

The API adopts a simple I/O model:

An individual device on a Host Interface can be described as:

- A board type - e.g. "hep3b" for the HEPC3 rev B interface. Board types are represented as strings to allow the introduction of new board types to be transparent to application programs
- A Board Number, where a system may have more than one Host Interface on the system the board number identifies which one is required. This is a 16-bit (`HE_WORD`) unsigned integer starting at 0.
- A Device Name identifies the separate devices on the Module carrier. This is represented as a 16 bit unsigned integer (`HE_WORD`), however a number of constants are pre-defined for all the current possibilities, e.g. ComportA, ComportB, JTAG, …

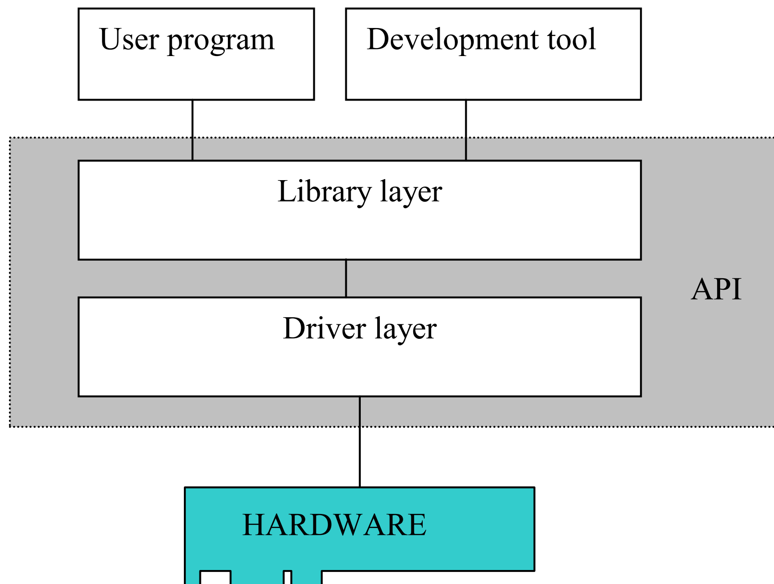
An individual device (e.g. ComportA) can only be accessed once it has been "opened". It is available to other applications (or parts of the same application) after it has been "closed". This is enforced through the use of "lock files" which are used by all variants of the API running on a system (e.g. DOS, Win 3.1 and Win32 applications running on Windows 95). In those Operating Systems which implement this automatically, this "lock file" is not used. An example of such an Operating System is VxWorks. It ensures that it is not possible for one application to interfere with another. The API provides two routines for this: `HeOpen()` and `HeClose()`. An Open device has a `HE_HANDLE` associated with it for use by all the other routines.

Generally an application needs to "Read from", "Write to" and "Reset" a Device, in the API this is represented by `HeRead()`, `HeWrite()` and `HeReset()`.

As mentioned earlier there are one or two other useful things that may be required, for example: memory allocation (`HeAlloc()`, `HeLock()`, `HeUnlock()`, `HeFree()`).

For performance reasons the API supports the concept of "Overlapped" or "Asynchronous" read/write. To do this it uses a structure called a `HE_IOSTATUS` and some associated API functions to track what is happening.

The implementation details of the API are specific to each Operating system, but the concept is always the same:-



The Library layer provides an interface to the development tools and user "host side application" programs, which is the same simple interface for all HUNT ENGINEERING hardware.

The driver layer provides an interface that is optimised for each operating system, and host board.

Thus the API provides an interface between the hardware and "host side" software that remains the same regardless of hardware type or revision, and also regardless of operating system.

This brings you a well-supported interface coupled to maximum performance for those operating systems that the HUNT ENGINEERING API supports.

API libraries

The API tries to provide as much transparency as possible by providing libraries that form the point of interface to the API.

On DOS the API routines are provided in a "static library". A new release of the API will require applications to be re-linked against the new "static library".

On Windows platforms the API routine entries are provided in a "stub library". A new release of the API will not require applications to be re-linked. The API routines themselves are provided in a "DLL". A DLL is a Dynamic Link Library, which is installed in a common directory and loaded "on demand". This means that new DLLs can be provided and applications will continue to work without requiring re-linking.

On VxWorks the API routines are supplied in the standard compiled code (“.o”) file format that is used by VxWorks.

As an example, suppose HUNT ENGINEERING have just developed a new host board, we will ship an updated DLL that supports this board. If an application identifies the host board it uses via a parameter of some sort (on the Command Line, in a “.ini” file, from the Registry) then the application can start using the new device straight away. See the discussions in the Section on tools such as Parallel C, Code Composer, Server Loader for tools that support this idea.

Device Drivers

There is much confusion about Device Drivers so we will try to explain a few terms as used in the API.

Some platforms support Direct I/O, this is where an application or Library can issue I/O or memory access instructions to Devices without support from the Operating System. MSDOS is a good example of such an Environment. Other platforms actually prevent such “direct” access - Windows NT is a good example.

A Device Driver is a piece of code that has a close link to the Operating System; examples are Kernel Mode Drivers on Windows NT and VxDs on Windows 3.1 and Windows 95.

The API libraries use a mixture of techniques:

On MSDOS they only use Direct I/O

On Windows 95 they can use a VxD for normal I/O and use direct I/O for JTAG (for performance reasons). If a VxD is properly installed for the board being used, then this will be used by the DLL, if there is no VxD then the DLL will use direct I/O to the board.

On Windows NT / 2000 / XP they only use Kernel Mode Drivers.

On VxWorks they use standard compiled code (“.o”) files that are downloaded onto the target system. The “*.o” files implement a POSIX standard driver (hevxdrv.o) as the API library interface, other “*.o” are implementations for the different boards).

On LINUX, modular device drivers are used, which take care of all I/O.

Detecting and using the Device Driver is the responsibility of the appropriate library. Providing the Driver for Windows 95 and Windows NT has been installed then the Library is able to use it - it will identify it based on the Host Interface Name in the `HeOpen()` call. The Win32 library (`hendrv.dll` for Microsoft Languages) detects whether it is on Windows NT or Windows 95. If it detects Windows NT, it will use a Kernel Mode Driver. If it detects Windows 95 it will use a VxD if one is present and Direct I/O if a VxD is not available.

This flexibility is automatically provided to applications which “should not” be concerned with “How” the API is provided - that is an Installation and HUNT ENGINEERING provided facility. It may be that the first release of a library for a new board is only provided by Direct I/O, moves onto a Device Driver without performance enhancements and then provides a highly optimised device driver. All of this is transparent to the application program which benefits from the improvements without needing to re-code or (in many cases) re-link.

Platforms

This release of the API (Ver 1.9.7) supports Intel based PCs (386, 486, Pentiums) that are running one of the Microsoft operating systems, Windows 95/98, Windows NT, 2000 and XP. In addition to those mentioned the API also supports VxWorks for the Intel x86 platform with the PC board support package (BSP). For VxWorks there is target support only. LINUX support exists for Intel based PCs, with a kernel version of 2.4.*. The API was developed and tested on RedHat 6.1, 7.1 and 9.

Current support is summarised below:-

Development & Target systems

O/S	API	Compiler	Server/Loader	(C4x) 3L Parallel C	Code Composer
Win95/98	yes	yes	Yes	Yes	Yes
WinNT	yes	yes	Yes	Yes (with the 32-bit 3L Winserver)	Yes
W2000/XP	yes	yes	Yes	no	Yes
Solaris (Sun)	yes	yes	Yes	Yes	No

Target Only systems

O/S	API	Compiler	Server/Loader	(C4x) 3L Parallel C	Code Composer
Linux (Intel)	yes	no	Yes	No	No
RTOS-32	yes	no	Yes	No	No
DOS	yes	yes	Yes	Yes	No
VxWorks	yes	no	Yes	No	No

Note that VxWorks is not running on the DSP, it is actually running on the host system.

Future releases may add support for other PC based operating systems, but that support may be limited by which of the DSP development tools are available for that platform.

HUNT ENGINEERING Host Interface Boards

	HEPC2E	HEPC3	HEV40-4	HERON-BASE2	HEPC8	HEPC9
DOS	Yes	Yes	Yes	No	No	No
Win95/98	Yes	Yes	Yes	No	Yes	Yes
WinNT	Yes	Yes	No	No	Yes	Yes
W2000/XP	No	Yes	No	Yes	Yes	Yes
Linux	Yes	Yes	No	No	Yes	Yes
RTOS-32	No	No	No	No	Yes	Yes
VxWorks	Yes	Yes	No	No	Yes	Yes

Note that for DOS, Windows and RTOS-32 the Microsoft Visual C++, and Borland C++ compilers are supported. For VxWorks and LINUX the GNU C/C++ compiler is

supported.

Installation

Installation of the API is discussed in a different document. There is an installation guide per supported operating system. Please refer to:

- ‘api – windows (installation & user manual)’ for Windows 95/98/ME/NT/W2K,
- ‘api – linux (installation & user manual) for LINUX,
- ‘api – vxworks (installation & user manual) for VxWorks,
- ‘api – rtos-32 (installation & user manual) for RTOS-32.

Writing Programs that use the API

The HUNT ENGINEERING API allows you to write a host computer application that can communicate and transfer data between the DSP system and the host machine.

Please note that the HUNT ENGINEERING Server/Loader can boot a network of DSP processors. The software comes in both executable and library format. If what you want is just to simply and quickly boot a network of DSP processors from within your own application, then using the Server/Loader library is much easier and quicker than developing your own network boot application using the API.

The Server/Loader uses the API when booting (and serving) a network of processors. You could see it as a layer above the HUNT ENGINEERING API. You can also use the Server /Loader and API together. For example, you can use the Server/Loader to boot a network of DSP processors, then use the API to communicate between your DSP programs and your PC host application (running on Windows, LINUX, VxWorks, RTOS-32 or another supported operating system).

Devices

The HUNT ENGINEERING API works with a concept called 'devices'. A carrier board has 1 or more devices. For example, the HEPC8 has a FIFO connecting the PCI interface to the first module on the board. This FIFO is one device ('FifoA'). The HEPC8 also has a serial bus interface. This is another device ('HSB'). Finally, there is a JTAG interface, used (for example) by Code Composer Studio, called 'Jtag'.

Different carrier boards may have different devices. For example, some boards may have more than 1 FIFO, and may support a device 'FifoB'. As another example, some carrier boards may have no serial bus interface. Typically there is always at least a 'FifoA' device and a 'Jtag' device, but this is not a rule and you must not assume that a certain device exists on all carrier boards.

Currently defined are devices:

```
FifoA
FifoB
FifoC
FifoD
FifoE
FifoF
Jtag
HSB
```

Open and Close a Device

The basic principle of the interface to the API is that you gain access to a device that is correctly identified by its 'board type', 'board number' and 'device type' using the `HeOpen()`, `HeOpen1()` or `HeOpenS()` function. This function is the ONLY function

that needs to know which device you are accessing, so you should provide a simple way to alter the details it uses so that a different device can easily be selected if the system requires it. Examples of how to do this are: command line parameters or storing the details in a text based "ini" type file.

This function will not allow you access to a device that is already in use, or does not exist. If it succeeds then it provides you with a unique "handle" which you use to identify the device in all future uses.

Once you have ownership of an open device you can perform reset, write and read type accesses to that device.

The read and write functions use "buffers" of memory that the data is passed through, and there are some memory management issues (Allocation and locking) that are aided by some API functions.

Asynchronous access

The read and write operations are asynchronous; that is, you start them with `HeRead()` or `HeWrite()`, but they will complete even if the transfer has not completed. The API then provides you with status functions (`HeTestIo()` and `HeWaitForIo()`) to allow you to track the progress of the transfer. This allows, but does not force, your host-based application to overlap the transfer of data with the processing or storage of data.

Conceptually, you can think of a read or write transfer to proceed 'in parallel' with your application. The `HeRead()` or `HeWrite()` function only requests the parallel thread to perform a read or write transfer. The status functions (`HeTestIo()` and `HeWaitForIo()`) verify if the parallel thread has already completed the transfer or not.

The `HeTestIo()` function just asks the parallel thread whether is has completed. It then immediately returns. The other status function, `HeWaitForIo()`, works differently. It will explicitly wait for the transfer to complete.

We used the term 'parallel thread' in a conceptual sense. The actual implementation can be anything: perhaps a device driver, an interrupt routine, win32 threads, and so on.

Writing

To write data to a carrier board, use the 'HeWrite' function. If 'HeWrite' completes the transfer, its return value will be 'HE_OK'. If the transfer is still ongoing, the return value is 'HE_IoInProgress'. Or it returns an error value.

Two functions exist that you can use to test whether the write transfer has completed, 'HeTestIo' and 'HeWaitForIo'. The first function, 'HeTestIo', quickly tests whether the transfer has completed, and then returns (it has the same return values as 'HeWrite'). The second function, 'HeWaitForIo' explicitly waits for the transfer to complete. It returns 'HE_OK' or an error value, but not 'HE_IoInProgress'.

Using 'HeWaitForIo', write-to-carrier-board code would typically look like:

```
Status = HeWrite(hDevice, WriteBuffer, size, WriteIoStatus);
// do some of your other work here
if (Status == HE_IoInProgress)
       Status = HeWaitForIo(hDevice, WriteIoStatus);
if (Status != HE_OK) // report an error and return
```

Using 'HeTestIo', write-to-carrier-board code would typically look like:

```
Status = HeWrite(hDevice, WriteBuffer, size, WriteIoStatus);
while (Status == HE_IoInProgress)
{
        // do some of your other processing here
        Status = HeTestIo(hDevice, WriteIoStatus);
}
if (Status != HE_OK) // report an error and return
```

Reading

To read data from a carrier board, use the 'HeRead' function. If 'HeRead' completes the transfer, its return value will be 'HE_OK'. If the transfer is still ongoing, the return value is 'HE_IoInProgress'.

Two functions exist that you can use to test whether the read transfer has completed, 'HeTestIo' and 'HeWaitForIo'. The first function, 'HeTestIo', quickly tests whether the transfer has completed, and then returns (it has the same return values as 'HeRead'). The second function, 'HeWaitForIo' explicitly waits for the transfer to complete. It returns HE_OK or an error value, but not 'HE_IoInProgress'.

Using 'HeWaitForIo', read-from-carrier-board code would typically look like:

```
Status = HeRead(hDevice, ReadBuffer, size, ReadIoStatus);
// do some of your other work here
if (Status == HE_IoInProgress)
        Status = HeWaitForIo(hDevice, ReadIoStatus);
if (Status != HE_OK) // report an error and return
```

Using 'HeTestIo', read-from-carrier-board code would typically look like:

```
Status = HeRead(hDevice, ReadBuffer, size, ReadIoStatus);
while (Status == HE_IoInProgress)
{
        // do some of your other processing here
        Status = HeTestIo(hDevice, ReadIoStatus);
}
if (Status != HE_OK) // report an error and return
```

Buffer Allocation (and the "huge" memory model)

The API is designed to be compatible across a range of Host System platforms. One of the major areas of difference between MSDOS and Win32 is the amount of memory that can be addressed by a pointer and how that memory can be allocated.

In order to achieve this platform independence, the API uses API specific pointer types and memory allocation routines, it also places some restrictions on the memory model a 16-bit application can be built with:

1) On MSDOS applications have to be built using the "large" memory model.

2) All buffers must be allocated using the four memory management API functions - `HeAlloc()`, `HeLock()`, `HeUnlock()` and `HeFree()`. These provide a "lowest common denominator" scheme that will work on all platforms. On some platforms, some of the routines perform no useful action but ensure that applications using them still function correctly.

   The memory allocation routines are fully described in another section, but in summary:

HeAlloc() Allocates a buffer, it does this using a huge alloc (halloc) where appropriate.

HeLock() Returns a (void *) pointer to the buffer allocated by HeAlloc() - this should be cast to an appropriate HE_PxxxxBUFFER to ensure the appropriate "huge" address calculation by the Compiler.

Although there is a performance impact of using "huge addressing" it eliminates any barriers such as a maximum buffer size of 64KBytes and retains source code compatibility across the range of supported platforms.

A simple program that uses the API interface

The following illustrates how to write a simple application. The main body of this manual should be consulted for full details of the routines used.

```
#include <stdio.h>                   // Always useful to have
#include <stdlib.h>                  // ditto
#include <string.h>
#include "heapi.h"                   // defines ALL the API things we need

HE_HANDLE hDevice = NULL;            // Handle for the device we will be
                                        using
HE_IOSTATUS ReadIoStatus = NULL;     // IoStatus for Read transfers
HE_IOSTATUS WriteIoStatus = NULL;    // IoStatus for Write transfer
HE_MEMHANDLE memHandle = NULL;       // Handle for allocated memory


HE_DWORD Status;                     // General Status variable
HE_PDWORDBUFFER test;                // pointer for a data buffer (to be allocated
                                        later)


void error(HE_DWORD errorcode)       // utility routine to provide useful error
                                        messages
{
char text[120];


HeErr2Text(Status, text);            // API routine that converts API error codes
                                       into Text
printf("Failed - %s, OsError = %d\n", text, errorcode);


    Status = HeClose(&hDevice);      // Make sure the Device is closed after an
                                       error
exit(1);                             // just terminate - simple failure handling!
}




void main(int argc, char *argv[])
```

```
{
HE_WORD  Board;                        // The board number we will be using
char *devname;                         // The Host Interface Type we will be using
HE_WORD Device;                        // The device we want to use
int i;

strcpy(devname, "hep3b");              // Use a HEPC3 rev B motherboard
Board = 0;                             // Board 0
 Device= ComportA;                     // Comport A on the HEPC3 (uses predefined
                                         constant)

Status = HeAlloc(&memHandle, 1000);    // Allocate a buffer
if(Status != HE_OK) error(HeGetLastOsError(memHandle));  // report an error on
                                                                 failure

Status = HeLock(memHandle, &test);     // finish memory allocation
if(Status != HE_OK) error(HeGetLastOsError(memHandle));

Status = HeOpen(devname, Board, ComportA, &hDevice);     // Open the Device
if(Status != HE_OK) error(HeGetLastOsError(hDevice));

printf("Resetting...\n");
Status = HeReset(hDevice);             // reset the device
if(Status != HE_OK) error(HeGetLastOsError(hDevice));

Status = HeInitIoStatus(hDevice, &ReadIoStatus);   // Initialise read IoStatus
if(Status != HE_OK) error(HeGetLastOsError(hDevice));

Status = HeInitIoStatus(hDevice, &WriteIoStatus);  // Initialise write IoStatus
if(Status != HE_OK) error(HeGetLastOsError(hDevice));

/* at this point we assume that the host board has some data to send, its unlikely,
but this                               is only an example program */


Status = HeRead(hDevice, test, 1000, ReadIoStatus);    // Initiate a read
if(Status == HE_IoInProgress) {                        // If read hasn't finished
Status = HeWaitForIo(hDevice, ReadIoStatus);           // wait for it to finish
                           }
if(Status != HE_OK) error(HeGetLastOsError(hDevice));  // Check for errors

Status = HeUnlock(memHandle);                          // unlock the memory
```

```
if(Status != HE_OK) error(HeGetLastOsError(memHandle));  // check for errors
Status = HeFree(&memHandle);                             // free the buffer
if(Status != HE_OK) error(HeGetLastOsError(memHandle));  // check for errors

Status = HeClose(&hDevice);                  // close the device - we're done
if(Status != HE_OK) error(HeGetLastOsError(hDevice));    // check for errors

exit(0);                                // Finished
}
```

There are a few things to note here:

- ALWAYS check for errors after calling an API routine!
- It's much better to print a useful text message than an error value so the API provides a suitable routine to help in doing this
- Because I/O is asynchronous you always need to check whether it is finished - this does not have to be straight away if you can do other useful work first. HOWEVER, the API does not support MULTIPLE outstanding I/O, so you must wait for an I/O on a particular device to finish BEFORE initiating another. It is, however, quite reasonable to have outstanding read and write on the same device but not more than 1 read or more than 1 write.
- It should not be necessary to ever use a Data type other than an API data type for the standard sort of activities. Care should ALWAYS be taken when using platform specific types (the most common problem is to use an `int` rather than an `HE_WORD`). If the compiler complains then you should find out why rather than just "cast" away the problem. If an API requires a `HE_WORD` don't cast an "`int`" variable when required, declare the variable as a `HE_WORD` if that is what it is.
- The program above won't achieve much in practice as it resets the 'C4x connected to ComportA and doesn't load an application on it to read from. Look at the example "reads" to see a more practical use of the API.

Maintaining Platform Independence

The sample programs shipped with the library are all platform independent and will compile, link and run on all the supported platforms without modification. As this platform independence is one of the objectives of the API it is **strongly** recommended that your application uses the same variable and pointer types as described below:

The following types will be defined as appropriate for the compiler and platform:

HE_DWORD	32 bit unsigned data item
HE_WORD	16 bit unsigned data item
HE_BYTE	8 bit unsigned data item
HE_INT32	32 bit signed data item
HE_INT16	16 bit signed data item

HE_INT8	8 bit signed data item
HE_PBYTEBUFFER	pointer to a buffer of 8-bit unsigned data items (this will be declared __huge on a 16 bit platform allowing use of buffers > 64Kbyte)
HE_PWORDBUFFER	pointer to a buffer of 16-bit unsigned data items (this will be declared __huge on a 16 bit platform allowing use of buffers > 64Kbyte)
HE_PDWORDBUFFER	pointer to a buffer of unsigned 32-bit data items (this will be declared __huge on a 16 bit platform allowing use of buffers > 64Kbyte)
HE_PINT8BUFFER	pointer to a buffer of 8-bit signed data items (this will be declared __huge on a 16 bit platform allowing use of buffers > 64Kbyte)
HE_PINT16BUFFER	pointer to a buffer of 16-bit signed data items (this will be declared __huge on a 16 bit platform allowing use of buffers > 64Kbyte)
HE_PINT32BUFFER	pointer to a buffer of 32-bit signed data items (this will be declared __huge on a 16 bit platform allowing use of buffers > 64Kbyte)
HE_BOOLEAN	platform independent Boolean type.
HE_HANDLE	Handle to Underlying Device Driver.
HE_IOSTATUS	Pointer to data structure used to contain the current status of an I/O operation, it is typically used for the Status of an asynchronous read/write operation
HE_MEMHANDLE	memory handle used when allocating "locked" buffers.

Locking of Devices for Exclusive access

All the API libraries support an "exclusive access" model except for the VxWorks API:

Whenever a device (e.g. Comport, FIFO or Jtag) is opened (via `HeOpen()`) a lock file will be created in the lock directory. This file is opened with exclusive write access and so forces any other attempt to open the file to fail.

This mechanism provides a platform wide common protection mechanism, in particular it ensures that MSDOS programs and Win32 programs can coexist on the same platform (e.g. Windows 95).

By default the lock directory will be created as:

```
c:\HE_LOCK
```

To change the default, create an environmental variable giving the directory to create `HE_LOCK` under - e.g.

```
set HE_LOCK=d:\fred
```

will cause the lock directory to be

```
d:\fred\HE_LOCK
```

Typically this should be done in `autoexec.bat` for MSDOS and Windows 95 and in the System Environmental Variables section of the System Option in the Control panel in Windows NT.

A program failing to get exclusive access to the relevant lock file will report a `HE_LockFailed` error along with whatever Operating System specific error code corresponds.

The lock file name is of the form xxxxyyzz.lck where:

`xxxxyyzz`	is a HEX Number where
`      ||`	is the Board Id (0, 1, ...)
`    ||`	is the Device Id (0=ComportA, ...)
`||||`	is a unique board type identifier,

- hep2d=0x02
- hep2e=0x03
- hep3b=0x04
- hep8a=0x08

Disabling Of Lock Files

PLEASE NOTE that lock files are not used by VxWorks.

There are a small number of cases where the use of a lock file causes problems to a system. An example of this could be a system where a ROM-disk file system is used to run the host code. In this case the file system is read only and the generation of a lock file will fail. To allow this to be supported the `HeOpenS()` function is provided which allows some of the more unusual options to be passed to it in a structure. There is an `HE_Switch_NoLockFile` flag that can be passed indicating that no locking should take place. WARNING this function should only be used in cases where it is absolutely necessary as it is disabling one of the major features of the API.

Using the Handles

There are 3 data structures used by the API:

- `HE_HANDLE`
- `HE_IOSTATUS`
- `HE_MEMHANDLE`

They are allocated and de-allocated by the API libraries. From the application point of view these are managed as pointers, they are declared as

```
HE_HANDLE hDevice = NULL;              // Initialise device handle
HE_IOSTATUS pIoStatus = NULL;          // Initialise IoStatus handle
HE_MEMHANDLE memHandle = NULL;         // Initial memory handle
```

The requirement to initialise them to NULL is deliberate in order to catch programming errors, the behaviour of the API routines when the handles have not been initialised is unpredictable and unsupported. The routines will check for NULL handle and report an error when it is not correct.

If you look at the `heapi.h` file you will see that the three data types are in fact "pointers to structures". All that is visible to the application is a simple structure. However, the pointers do actually point to a much larger data structure inside the libraries. The only field visible to an application is the first - `HandleType`. This field is used to validate that a handle is pointing to a valid structure and of the correct type for the operation.

In the current release of the API:

```
hDevice->HandleType =       0xdeafeed0
pIoStatus->HandleType =     0xdeafeed2
memHandle->HandleType =     0xdeafeed1
```

The data structures corresponding to these handles are allocated, used and destroyed as follows:

HE_HANDLE

This must be initialised to NULL before calling the `HeOpen()` function.

It will be allocated by `HeOpen()` and released by `HeClose()`. Both these routines require a pointer to the `HE_HANDLE` to be passed to them.

IMPORTANT - whether the `HeOpen()` succeeds or fails it should ALWAYS be matched by a `HeClose()` - this ensures that the data structure is correctly cleaned up. If no data structure had been allocated (i.e. the Handle is still NULL) the close routine will report this via its status return.

Currently there are only 2 status returns from `HeOpen` when the data structure will not have been allocated:

`HE_HandlePointerNotNull`	No data allocated as handle must be NULL when calling `HeOpen()`. `HeClose()` will probably report an error of `HE_InvalidHandlePointer` if the pointer is to random data.
`HE_FailedToAllocMemoryForHandle`	No Data allocated as there was an error calling `malloc` (typically `_fmalloc`). `HeClose()` will report an error of `HE_NullHandlePointer` as it must have been NULL for `HeOpen` to get as far as reporting this error.
Anything else	data allocated - to release call `HeClose()`

This means that there are a number of error status returns that still require a `HeClose()`. Normally one would not expect to Close a device that failed to Open but the requirements of the implementation of the API on some platforms requires some compromise of the generic model.

Always match EVERY `HeOpen()` with a `HeClose()` whether it succeeds or not.

HE_MEMHANDLE

This should be initialised to NULL before calling `HeAlloc()`.

It is allocated by the `HeAlloc()` routine and released by the `HeFree()` routine

All `HeAlloc()`s should be matched by a `HeFree()`.

HE_IOSTATUS

This should be initialised to NULL before calling `HeInitIoStatus()`.

It is allocated by `HeInitIoStatus()` and released when the associated `hDevice` is `HeClosed()`'d.

The following routines are always present:

HeOpen()

```
HE_DWORD HeOpen(char *BoardType, HE_WORD BoardId, HE_WORD DeviceId,
HE_HANDLE *hDevice);
```

This function provides exclusive access to the requested device. Where appropriate it will access a suitable device driver for the platform and device. Wherever possible it will "Open" the device to support asynchronous I/O - i.e. the `HeRead()` and `HeWrite()` routines may return before the I/O is complete and it is necessary to call either `HeTestIo()` or `HeWaitForIo()` routines. Even if asynchronous I/O is not possible the `HeTestIo()` and `HeWaitForIo()` routines will behave correctly and so it should **ALWAYS** be assumed that I/O is asynchronous even though at any particular time a particular implementation may not be.

BoardType Establish access to a board identified by an ASCII character string. Currently supported HUNT ENGINEERING boards are:

"hep4a"	HEPC4 rev A/B
"hep3b"	HEPC3 rev B/C
"hep2e"	HEPC2E
"hep2d"	HEPC2-M rev D
"hep6a"	HEPC6 rev A
"hep8a"	HEPC8 rev A
"hep9a"	HEPC9 rev A

For MSDOS and Windows 95 these boards are also supported:

`xahev` HEV40-4, for use with a XYCOM card with 486 processor
`xbhev` HEV40-4, for use with a XYCOM card with Pentium processor

These 2 boards are always accessed directly by the drivers.

BoardId is used to identify the required board of this type where more than 1 is supported. First board is 0, second 1, etc.

DeviceId selects the *BoardType* specific device where a board supports more than 1 device. There are a number of predefined constants for typical situations. The following definitions are currently made:

- FifoA
- ComportA
- ComportB
- Jtag N.B. Not available under VxWorks and LINUX.
- ComportC
- ComportD
- HSB N.B. Not available under VxWorks and LINUX.

If you want to use `HeOpen()` with one of these constants (as a string), rather than the corresponding value then use the `HeOpen1()` routine.

`*hDevice`	pointer to the handle for the device - used in all subsequent operations
`return`	`HE_OK` is returned when the call was successful. When the call was not successful an error code is returned. See the "Status Codes" section elsewhere in this chapter for details. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

IMPORTANT: All calls of `HeOpen()` **MUST** be matched by a call to `HeClose()`, even if the `HeOpen()` returned an error code.

HeOpen1()

```
HE_DWORD HeOpen(char *BoardType, HE_WORD BoardId, char *DeviceC, HE_HANDLE
*hDevice);
```

This routine is exactly the same as the `HeOpen()` routine except that it accepts a `DeviceId` as a string instead of as number. This is of benefit to programs that retrieve the `DeviceId` from some form of user input and don't want to have to "hard code" the string to constant mapping of the `DeviceId`.

IMPORTANT: All calls of `HeOpen1()` **MUST** be matched by a call to `HeClose()`, even if the `HeOpen1()` returned an error code

HeOpenS()

```
HE_DWORD HeOpenS(char *BoardType, HE_WORD BoardId, char *DeviceC, HE_HANDLE
*hDevice, HE_DWORD *switches);
```

This routine is exactly the same as the HeOpen1() routine except that it accepts a switch array (an array of HE_DWORDS, the last entry must have the value HE_Switch_Last) to allow for some of the rarer options on opening. The currently supported options are:

- `HE_Switch_Last` — Last option in the switch array - indicates end of list
- `HE_Switch_NoLockFile` — Don't use a lock file **Use this option with caution** as it is now **YOUR** responsibility to ensure exclusive device access.
- `HE_Switch_ByteSwap` — Where available turn on the module carrier's Byteswap logic.
- `HE_Switch_NoPCIMasterMode` — Do not use master mode for PCI boards.
- `HE_Switch_NoInterrupts` — Do not use interrupts when reading or writing.
- `HE_Switch_TestInterrupt` — Perform an interrupt test for this device.
- `HE_Switch_IRQ10` — For VxWorks used with the HEPC2: use interrupts with IRQ 10
- `HE_Switch_IRQ11` — For VxWorks used with the HEPC2: use interrupts with IRQ 11
- `HE_Switch_IRQ12` — For VxWorks used with the HEPC2: use interrupts with IRQ 12

- `HE_Switch_IRQ15` — For VxWorks used with the HEPC2: use interrupts with IRQ 15

IMPORTANT: All calls of `HeOpenS()` **MUST** be matched by a call to `HeClose()`, even if the `HeOpenS()` returned an error code.

HeClose()

```
HE_DWORD HeClose(HE_HANDLE *hDevice);
```

Release resources allocated when the device was opened, and

hDevice	is the handle returned from HeOpen.
return	`HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeRead()

```
HE_DWORD HeRead(HE_HANDLE hDevice, void *data, HE_DWORD Count, HE_IOSTATUS
IoStatus);
```

Initiate a read operation on a previously opened device. Under MSDOS the function will return after all bytes have been read. Under Windows, the function returns immediately, and progress of the read operation can be checked by examining `IoStatus` (using `HeTestIo()`). `IoStatus` must have been initialised using the `HeInitIoStatus` function.

hDevice	is the handle returned from `HeOpen`.
data	is a pointer to the start of the buffer to read into
Count	is the number of **BYTES** to read
IoStatus	is the `IoStatus` handle. It is used to obtain the status of the outstanding I/O or to wait for its completion.
return	`HE_OK` is returned when the call was successful, and I/O was completed. `HE_IoInProgress` is returned when the call was successful, but when I/O was not yet completed. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`. In particular if the I/O is complete it will return HE_OK and if it is still in progress it returns `HE_IoInProgress`.

HeWrite()

```
HE_DWORD HeWrite(HE_HANDLE hDevice, void *data, HE_DWORD Count, HE_IOSTATUS
IoStatus);
```

Initiate a write operation on a previously open device. Under MSDOS the function will return after all bytes have been written. Under Windows, the function returns immediately, and progress of the write operation can be checked by examining `IoStatus` (using `HeTestIo()`). `IoStatus` must have been initialised using the `HeInitIoStatus`

function.

hDevice is the handle returned from HeOpen.

***data** is a pointer to the start of the buffer to write from

Count is the number of **BYTES** to write

IoStatus is the `IoStatus` handle. It is used to obtain the status of the outstanding I/O or to wait for its completion.

return `HE_OK` is returned when the call was successful, and I/O was completed. `HE_IoInProgress` is returned when the call was successful, but when I/O was not yet completed. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`. In particular if the I/O is complete it will return `HE_OK` and if it is still in progress it returns `HE_IoInProgress`.

HeDelay()

```
HE_DWORD HeDelay(HE_DWORD wait);
```

Platform independent delay routine

wait time in milliseconds to wait for

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeReset()

```
HE_DWORD HeReset(HE_HANDLE hDevice);
```

Reset the device opened on `hDevice`. In 'C4x terms this would reset a Comport on a Board NOT the board itself

hDevice is the handle returned from HeOpen.

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeReset1()

```
HE_DWORD HeReset1(HE_HANDLE hDevice, HE_DWORD ResetHold);
```

Reset the device opened on `hDevice`. In 'C4x terms this would reset a Comport on a Board NOT the board itself - the reset will be help asserted for `ResetHold` ms.

This routine supersedes `HeReset()` which uses an arbitrarily short reset hold time

hDevice is the handle returned from `HeOpen`.

ResetHold time in ms to hold reset asserted

return HE_OK is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling GetLastOsError(hDevice).

HeInitIoStatus

HE_DWORD HeInitIoStatus(HE_HANDLE hDevice, HE_IOSTATUS *pIoStatus)

Initialise the HEIOSTATUS object - this is **REQUIRED** before use in HeRead() or HeWrite(). If it has not been called then the behaviour of HeRead(), HeWrite, HeWaitForIo() and HeTestIo() is unpredictable and insupportable.

hDevice is the handle returned from HeOpen.

***pIoStatus** is a pointer to the IoStatus Handle. It is used to obtain the status of the outstanding I/O or to wait for its completion.

return HE_OK is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling GetLastOsError(hDevice).

HeWaitForIo()

HE_DWORD HeWaitForIo(HE_HANDLE hDevice HE_IOSTATUS IoStatus,);

Wait until a Read/Write operation is complete

hDevice is the handle returned from HeOpen.

IoStatus is the IoStatus handle. It is used to obtain the status of the outstanding I/O or to wait for its completion.

return HE_OK is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling GetLastOsError(hDevice).

HeTestIo()

HE_DWORD HeTestIo(HE_HANDLE hDevice, HE_IOSTATUS IoStatus);

Check whether the I/O has finished

hDevice is the handle returned from HeOpen.

IoStatus is the IoStatus handle. It is used to obtain the status of the outstanding I/O or to wait for its completion.

return HE_OK is returned when the call was successful, and I/O was completed. HE_IoInProgress is returned when the call was successful, but when I/O was not yet completed. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling GetLastOsError(hDevice). In particular if the I/O is complete it will return HE_OK and if it is still in progress it returns HE_IoInProgress.

HeErr2Text()

```
void HeErr2Text(HE_DWORD errcode, char *errtxt);
```

Converts an error code defined in the API into an ASCII text string suitable for output to a terminal or log file.

`errcode` is an error code returned by one of the routines.

`*errtxt` is a pointer to the text buffer - this should be at least 80 BYTES long

HeGetIoGranularity()

```
HE_DWORD HeGetIoGranularity(HE_HANDLE hDevice, HE_DWORD *granularity);
```

Will return the granularity required for I/O buffers. For the 'C4x this is 4 bytes,

`hDevice` is the handle returned from `HeOpen`.

`granularity` is the number of bytes required for I/O on this device - use this in conjunction with the error message `HE_IllegalCount`

`return` `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

The following functions support a General Buffer allocation scheme which will ensure that the Device Driver and User Program have a consistent view of the buffers location in memory.

HeGetBoardInfo() *(HERON carriers only)*

```
HE_DWORD HeGetBoardInfo(char *BoardType, HE_WORD BoardId, HE_DWORD
*information);
```

This function will return information for the specified board.

BoardType Legacy boards, such as the HEPC2E and HEPC3B, don't support this function, but C6x carrier boards (such as the HEPC8 and HEPC9) do.

BoardId is used to identify the required board of this type where more than 1 is supported.

`information` is a pointer to an array whose length is defined according to the board information function required. The currently supported board information function types are:

`HE_Get_ModuleTypes` Return module types

When calling this function, the first element of the array pointed to by **`information`** must contain the function type listed above.

You **must** ensure the array is at least 21 HE_DWORDS in size.

For the function type HE_Get_ModuleTypes, the returned information is as follows:

information[1] = A module is fitted to slot 1 of the carrier

information[2] = The module in slot 1 has a processor

information[3] = The module in slot 1 has a serial bus connection

information[4] = The module in slot 1 supports JTAG

information[5] = The data-path width for the module in slot 1

information[6] = A module is fitted to slot 2 of the carrier

information[7] = The module in slot 2 has a processor

information[8] = The module in slot 2 has a serial bus connection

information[9] = The module in slot 2 supports JTAG

information[10] = The data-path width for the module in slot 2

information[11] = A module is fitted to slot 3 of the carrier

information[12] = The module in slot 3 has a processor

information[13] = The module in slot 3 has a serial bus connection

information[14] = The module in slot 3 supports JTAG

information[15] = The data-path width for the module in slot 3

information[16] = A module is fitted to slot 4 of the carrier

information[17] = The module in slot 4 has a processor

information[18] = The module in slot 4 has a serial bus connection

information[19] = The module in slot 4 supports JTAG

information[20] = The data-path width for the module in slot 4

return `HE_OK` is returned when the call was successful. If this function is called for a **`BoardType`** that is not supported (i.e any board that is not a HERON carrier), the function will return **`HE_Unsupported`**. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter.

HeAlloc()

`HE_DWORD HeAlloc(HE_MEMHANDLE *memHandle, HE_DWORD MessageSize);`

Allocates a buffer for use by the `HeLock()` routine, note this does not provide an address for the buffer, that will be generated by `HeLock()`.

`MessageSize` Number of BYTES to allocate

`*memHandle` pointer to the Memory Handle

`return` `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeFree()

`HE_DWORD HeFree(HE_MEMHANDLE *memHandle);`

Release the buffer allocated by `HeAlloc()`.

memHandle pointer to the Memory Handle returned by HeAlloc()

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeLock()

```
HE_DWORD HeLock(HE_MEMHANDLE memHandle, void *tmpptr);
```

Generates an address for the buffer allocated by `HeAlloc()` and ensure that the buffers address will not be changed

***tmpptr** address of the buffer allocated by `HeAlloc()`

memHandle Memory Handle returned by `HeAlloc()`

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeUnlock()

```
HE_DWORD HeUnlock(HE_MEMHANDLE memHandle);
```

Releases the address generated for the buffer allocated by `HeLock()`

memHandle Memory Handle returned by `HeAlloc()`

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeConfig()

```
HE_DWORD HeConfig(HE_HANDLE hDevice, HE_DWORD *Config);
```

Read the state of the Config Signal from the module carrier

hDevice is the handle returned from `HeOpen()`.

Config pointer to a variable to receive the status of Config -

1 ⇒ Config is asserted by at least one TIM on the module carrier

0 ⇒ Config has been released by all of the TIMs

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`. Reading of the Config bit is not supported on all module carriers and so the return code should be checked for errors before using attempting to use the Config bit any further.

HeJtagWrite()

```
HE_DWORD HeJtagWrite(HE_HANDLE hDevice, HE_DWORD PortData);
```

hDevice is the handle returned from HeOpen.

PortData is a simple encoding of the offset from the Jtag base address to write to and the data to write to that offset - the Port offset is in the high 16 bits and the data in the low 16 bits

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

PLEASE NOTE that JTAG is not supported under VxWorks.

HeJtagRead()

```
HE_DWORD HeJtagRead(HE_HANDLE hDevice, HE_DWORD *PortData);
```

hDevice is the handle returned from `HeOpen`.

***PortData** is a pointer to the encoded Jtag data. On calling the routine the Jtag offset should be in the high 16 bits (as it is for `HeJtagWrite`), the low 16 bits is undefined. On return the data read will be in the low 16 bits and the high 16 bits is undefined

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

PLEASE NOTE That JTAG is not supported under VxWorks.

HeGetLastOsError()

```
HE_DWORD HeGetLastOsError(void *hand);
```

***hand** is a pointer to a HUNT ENGINEERING API handle: a device (HE_HANDLE) or a memory handle (HE_MEMHANDLE). The function will return the operating system error code of the last HUNT ENGINEERING API call done before a call to this function.

return The return value is an O/S specific error code.

The HUNT ENGINEERING API may run into broadly two types of errors: operating system errors (for example, a file cannot be opened or a device cannot be found), or API specific errors (for example, using a wrong handle or calling HeWrite with a Count that is not a multiple of 4). In both cases, a HUNT ENGINEERING API function will return an error (return value unequal to HE_OK). If the error resulted from an unsuccessfull operating system call, then this function will specify exactly what error occurred. For example, if you try to open a board that isn't installed, HeOpen will return error HE_OpenFailed, and HeGetLastOsError() will then return 2 ("No such file or directory"), under Windows.

HeHSBSendMessageEx()

```
HE_DWORD HeHSBSendMessageEx(HE_HANDLE hDevice, HE_BYTE msg_type,
                            HE_WORD tgbd, HE_BYTE slot,
                            void *data, HE_DWORD size,
                            int msec);
```

This function sends a message over the serial bus. The message is addressed to serial bus device `slot`. The message itself is an array, pointed to by `data` and is `size` bytes long. If the message cannot be sent to `slot` within `msec` milli-seconds, the function will time out. The parameter `msg_type` is the message type. For example, to ask a HERON processor module for module information, `msg_type` is 1, `data` is NULL and `size` is 0.

There is also a similar function called HeHSBSendMessage. This function doesn't have the `tgbd` parameter. HeHSBSendMessage assumes that `slot` is on the same board as `hDevice`. The HeHSBSendMessageEx function allows access to any slot, even on other boards. (Assuming that an HSB connection exists between the boards.)

HeHSBSendMessageEx, and its predecessor HeHSBSendMessage, are used to send HUNT HSB protocol messages. A full list of HUNT HSB protocol message types can be found in the HSB specification document. (The file is named 'hsbspec.pdf' and usually located on the HUNT CD in the \web\pdfs\tech directory. You can also use the HUNT CD front-end, "User Manuals" → "Technology Documents" → "HERON Serial Bus Specification". Review Appendix 1.)

`hDevice` is the handle returned from `HeOpen`. (using device name 'hsb' or 3).

`msg_type` is the message type. Some of the message types can be found defined in the serial bus section in this manual. Message types are special to HUNT ENGINEERING serial bus implementations. If you create your own serial bus functions (eg using the HERON-API serial bus functions) you may want to create your own message types.

`tgbd` is the board switch of the board on which the target slot resides. The tgbd parameter together with the slot parameter is used to create an hsb id. (Bits 0..2 are slot id, bits 3..6 are target board switch.)

`slot` is the serial bus device the message is aimed at. Use the slot number where the HERON module is plugged in as the id. For example, to send a serial bus message to a HERON module in slot 1 of a HEPC8 board, used `slot=1`.

`data` is the pointer to the serial bus message. Not all serial bus messages use a data array. Some messages consist only of a message type. An example is the processor module query (`msg_type=1`).

`size` is the the size of the serial bus message, in bytes.

`msec` is the number of milli-seconds after which the function times out.

HeHSBReceiveMessageEx()

```
HE_DWORD HeHSBReceiveMessageEx(HE_HANDLE hDevice, HE_BYTE_*msg_
                                type, HE_WORD tgbd, HE_BYTE slot,
                               void *data, HE_DWORD size,
                               HE_DWORD *read, int msec);
```

This function receives a message over the serial bus. The message is expected to be received

from serial bus device *tgbd<<3+slot*. The message itself is to be stored in an array pointed to by *data* and is expected to be *size* bytes long. If you don't know the exact size, take a reasonable number larger than expected. If the message cannot be received from *tgbd<<3+slot* within *msec* milli-seconds, the function will time out. The parameter *msg_type* is the message type of the message received. For example, to receive HERON processor module information after a query, *msg_type* will be 2.

There is also a similar function called HeHSBReceiveMessage. This function doesn't have the tgbd parameter. HeHSBReceiveMessage assumes that slot is on the same board as hDevice. The HeHSBReceiveMessageEx function allows access to any slot, even on other boards. (Assuming that an HSB connection exists between the boards.)

HeHSBReceiveMessageEx, and its predecessor HeHSBReceiveMessage, are used to receive HUNT HSB protocol messages. A full list of HUNT HSB protocol message types can be found in the HSB specification document. (The file is named 'hsbspec.pdf' and usually located on the HUNT CD in the \web\pdfs\tech directory. You can also use the HUNT CD front-end, "User Manuals" → "Technology Documents" → "HERON Serial Bus Specification". Review Appendix 1.)

hDevice is the handle returned from HeOpen. (using device name 'hsb' or 3).

msg_type is the message type received. Some of the message types can be found defined in the serial bus section. Message types are special to HUNT ENGINEERING serial bus implementations. If you create your own serial bus functions (eg using the HERON-API serial bus functions) you may want to create your own message types.

tgbd is the board switch of the board on which the target slot resides. The tgbd parameter together with the slot parameter is used to create a heronid. (Bits 0..2 are slot id, bits 3..6 are target board switch.)

slot is the serial bus device the message is to be received from. Use the slot number where the HERON module is plugged in as the id. For example, to receive a serial bus message from a HERON module in slot 1 of a HEPC8 board, used *slot=1*.

data is the pointer to a buffer to receive the serial bus message in.

size is the the size of the expected size of the message, in bytes. Make sure that *data* points to an array that is at least this size (in bytes).

read is the the size of the actual message received.

msec is the number of milli-seconds after which the function times out.

HeHSBStartSendMessageEx()

```
HE_DWORD HeHSBStartSendMessageEx(HE_HANDLE hDevice,
                                 HE_WORD tgbd, HE_BYTE slot,
                                 int msec);
```

This function prepares to send a message over the serial bus. After calling this function successfully, use HeHSBSendMessageDataEx to send the actual message. The message is addressed to serial bus device *tgbd<<3+slot*.

There is also a similar function called HeHSBStartSendMessage. This function doesn't have the tgbd parameter. HeHSBStartSendMessage assumes that slot is on the same board as

hDevice. The HeHSBStartSendMessageEx function allows access to any slot, even on other boards. (Assuming that an HSB connection exists between the boards.)

hDevice	is the handle returned from HeOpen. (using device name 'hsb' or 3).
tgbd	is the board switch of the board on which the target slot resides. The tgbd parameter together with the slot parameter is used to create a heronid. (Bits 0..2 are slot id, bits 3..6 are target board switch.)
slot	is the serial bus device the message is aimed at. Use the slot number where the HERON module is plugged in as the id. For example, to send a serial bus message to a HERON module in slot 1 of a HEPC8 board, used *slot=1*.
msec	is the number of milli-seconds after which the function times out.

HeHSBSendMessageDataEx()

```
HE_DWORD HeHSBSendMessageDataEx(HE_HANDLE hDevice,
                                void *data, HE_DWORD size,
                                int msec);
```

This function sends a message over the serial bus. The message is an array, pointer to by *data* and is *size* bytes long. If the message cannot be sent to *tgbd<<3+slot* (as specified by HeHSBStartSendMessageEx) within *msec* milli-seconds, the function will time out. The first two bytes of the message must be the message type and the serial bus return address. For example, to ask a HERON processor module for module information, the message type is 1. The serial bus return address is always (board number<<3)+5. For example, for HEPC8 number 0, the return address is 5. For an HEPC8 number 1, the return address is 13 (0xd).

There is also a similar function called HeHSBSendMessageData. This function assumes that the hsb device (identified by slot in HeHSBStartSendMessage) is on the same board as hDevice. The HeHSBStartSendMessageEx function allows access to any slot, even on other boards. (Assuming that an HSB connection exists between the boards.)

hDevice	is the handle returned from HeOpen. (using device name 'hsb' or 3).
data	is the pointer to the serial bus message. The first byte in the message must be the message type. The second byte in the array must be the return address (5+(board number<<3). Not all serial bus messages use a data array larger than size 2. Some messages consist only of a message type and a return address. An example is the processor module query (*message type is 1*).
size	is the the size of the serial bus message, in bytes.
msec	is the number of milli-seconds after which the function times out.

HeHSBEndOfSendMessageEx()

```
HE_DWORD HeHSBEndOfSendMessageEx(HE_HANDLE hDevice, int msec);
```

This function signals the end of a serial bus send message transfer. It must be used after 1 or more calls to HeHSBSendMessageDataEx. Calling this function will free the serial bus so that other serial bus devices can access it.

There is also a similar function called HeHSBEndOfSendMessage. This function assumes

that the hsb device (identified by `slot` in HeHSBStartSendMessage) is on the same board as `hDevice`. The HeHSBStartSendMessageEx function allows access to any slot, even on other boards. (Assuming that an HSB connection exists between the boards.)

`hDevice`	is the handle returned from `HeOpen`. (using device name 'hsb' or 3).
`msec`	is the number of milli-seconds after which the function times out.

HeHSBStartReceiveMessageEx()

```
HE_DWORD HeHSBStartReceiveMessageEx(HE_HANDLE hDevice,
                                    HE_DWORD *id, int msec);
```

This function prepares to receive a message over the serial bus. Parameter *`id`* is the serial bus device the incoming message is aimed at. If no message is received within *`msec`* milli-seconds, the function will time out.

There is also a similar function called HeHSBStartReceiveMessage. This function is actually the same as HeHSBStartReceiveMessageEx. The HSB Ex functions allow access to any slot, even on other boards. (Assuming that an HSB connection exists between the boards.) However, the HSB receive functions don't know where a message comes from, so in effect HeHSBStartReceiveMessage is identical to HeHSBStartReceiveMessageEx.

`hDevice`	is the handle returned from `HeOpen`. (using device name 'hsb' or 3).
`id`	is the serial bus device the incoming message is addressed to .
`msec`	is the number of milli-seconds after which the function times out.

HeHSBReceiveMessageDataEx ()

```
HE_DWORD HeHSBReceiveMessageDataEx(HE_HANDLE hDevice,
                                   void *data, HE_DWORD size,
                                   HE_DWORD *read, int msec);
```

This function receives a message over the serial bus. The message is to be stored in the array pointed to by *`data`* and is expected to be *`size`* bytes long. If you don't know the exact size, take a reasonable number larger than expected. If a message cannot be received within *`msec`* milli-seconds, the function will time out. The first byte in the message received will be the message type. For example, to receive HERON processor module information after a query, the message type will be 2.

There is also a similar function called HeHSBReceiveMessageData. This function is actually the same as HeHSBReceiveMessageDataEx. The HSB Ex functions allow access to any slot, even on other boards. (Assuming that an HSB connection exists between the boards.) However, the HSB receive functions don't know where a message comes from, so in effect HeHSBReceiveMessageData is identical to HeHSBReceiveMessageDataEx.

`hDevice`	is the handle returned from `HeOpen`. (using device name 'hsb' or 3).
`data`	is the pointer to a buffer to receive the serial bus message in.
`size`	is the the size of the expected size of the message, in bytes. Make sure that *`data`* points to an array that is at least this size (in bytes).
`read`	is the the size of the actual message received.
`msec`	is the number of milli-seconds after which the function times out.

HeHSBEndOfReceiveMessageEx()

```
HE_DWORD HeHSBEndOfReceiveMessageEx(HE_HANDLE hDevice,int msec);
```

This function signals the end of receiving one full message over the serial bus. To receive more messages, you would need to call HeHSBStartReceiveMessage again.

There is also a similar function called HeHSBEndOfReceiveMessage. This function is actually the same as HeHSBEndOfReceiveMessageEx. The HSB Ex functions allow access to any slot, even on other boards. (Assuming that an HSB connection exists between the boards.) However, the HSB receive functions don't know where a message comes from, so in effect HeHSBEndOfReceiveMessage is identical to HeHSBEndOfReceiveMessageEx.

hDevice is the handle returned from `HeOpen`. (using device name 'hsb' or 3).

msec is the number of milli-seconds after which the function times out.

HeHSBInit()

```
HE_DWORD HeHSBInit(HE_HANDLE hDevice, HE_BYTE hsb_id);
```

This function initialises the serial bus. It has to be called after a system reset (HeReset()), or after an call to an HeOpen function but without resettting the board.

hDevice is the handle returned from `HeOpen`.

hsb_id is the serial bus device ID of the motherboard. On an HEPC8, for example, this is always 5. But when the HEPC8 board switch is not 0, but for example 2, then the device ID is 5+(2<<3) = 21 (hex 0x15).

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned. For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeHSBMaster()

```
HE_DWORD HeHSBMaster(HE_HANDLE hDevice,
                     HE_BYTE hsb_id, int msec);
```

This function tries to make the user application master of the serial bus. You must first become a master before you can write to the serial bus. Typically you call HeHSBMaster() immediately after a call to HeHSBinit(). The function waits at most `msec` milliseconds to become a master on the serial bus.

hDevice is the handle returned from `HeOpen`.

hsb_id is the serial bus device ID that you want to write to. On an HEPC8, for example, slot 1 has id 1, slot 2 has id2, slot 3 has id 3 and slot 4 has id 4. When the HEPC8 board switch is not 0, but for example 5, then slot 1 has id 1+(5<<3) = 41, slot 2 has id 2+(5<<3) = 42 and so on.

msec time in milliseconds that the function waits at most to become master.

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned (HE_TimeOut if after `msec` it cannot become master of the serial bus, or HE_HSBMasterFailed in the event of a driver problem). For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be

found by calling `GetLastOsError(hDevice)`.

HeHSBSlave()

```
HE_DWORD HeHSBSlave(HE_HANDLE hDevice, int msec);
```

This function tries to release mastership of the serial bus. You must first become a master before you can write to the serial bus. Typically you call HeHSBSlave() after a call to HeHSBMaster followed by one or more HeWrite calls. The function serves to release "mastership" of the serial bus. You wouldn't need to call this function if you're not the master of the serial bus, e.g. immediately after a call to HeReset(). The function waits at most `msec` milliseconds to release "mastership" of the serial bus.

hDevice is the handle returned from `HeOpen`.

msec time in milliseconds that the function waits at most to release "mastership".

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned (HE_TimeOut if after `msec` it cannot become master of the serial bus, or HE_HSBSlaveFailed in the event of a driver problem). For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeHSBListen()

```
HE_DWORD HeHSBListen(HE_HANDLE hDevice, HE_DWORD *id, int msec);
```

This function waits for the start of a message to arrive. Typically you call HeHSBListen() immediately after a call to HeHSBSlave(). The function waits at most `msec` milliseconds to wait for a message. Upon finding the start of a message, the `id` parameter will hold the serial bus id from to which the message is directed.

hDevice is the handle returned from `HeOpen`.

id is the serial bus device ID that a message is destined for. If it's for you, then `id` is identical to the `hsb_id` value you used in HeHSBinit().

msec time in milliseconds that the function waits at most.

return `HE_OK` is returned when the call was successful. When the call was not successful, an error code is returned (HE_TimeOut if after `msec` it cannot become master of the serial bus, or HE_HSBListenFailed in the event of a driver problem). For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling `GetLastOsError(hDevice)`.

HeHSBFlush()

```
HE_DWORD HeHSBFlush (HE_HANDLE hDevice, HE_DWORD *read);
```

This function flushes the HSB read state. Before you close the HSB, first call this function. It will ensure that any remaining bytes will be read. If this is not done, the HSB may be left in a state that prohibits any other device using the HSB. Level 2 and 3 HSB functions call this function automatically (in HeHSBEndOfReceiveMessage and HeHSBReceiveMessage). You only need to use HeHSBFlush if you use level 1 HSB functions to access the HSB.

hDevice	is the handle returned from HeOpen.
read	the number of bytes flushed.
return	HE_OK is returned when the call was successful. When the call was not successful, an error code is returned (HE_HSBFlushFailed in the event of a driver problem). For details, see the "Status Codes" section elsewhere in this chapter. An O/S specific error code can be found by calling GetLastOsError(hDevice).

HeGetDeviceInfo()

```
HE_DWORD HeGetDeviceInfo(HE_HANDLE hDevice, char *BoardDescr,
                         HE_WORD *BoardId, HE_WORD *DeviceId);
```

This function returns the board-type, board number and device with which the handle was opened. Parameters BoardDescr, BoardId and DeviceId must not be NULL.

hDevice	is the handle returned from HeOpen.
BoardDescr	The board description with which the handle was opened. For example, for an HEPC9, "hep9a" will be written into BoardDescr.
BoardId	The board number with which the handle was opened.
DeviceId	The device id with which the handle was opened. Possible values are listed in 'heapi.h': FifoA (0), FifoB (1), etc.

Status Codes.

As the HUNT ENGINEERING API continues to develop, more status codes may be added. The include file "heapi.h" will always have a full, complete, list of status codes. In this include file, every status (or error) code will be #defined as a "mnemonic". Hopefully the "mnemonic" will give a sufficiently good description to indicate the nature of the error.

Mnemonic	Value	Description
HE_OK	0x0000	No Error.
HE_SIG_FAIL	0x0001	Not used.
HE_HEPC3_DETECT_FAIL	0x0002	Failed to detect HEPC3 device on the PCI bus. (2)
HE_CONFIG_READ_FAIL	0x0003	Failed to read a configuration space register. (2)
HE_COMMAND_SET_FAIL	0x0004	Failed to set required values in PCI Command Register. (2)
HE_STATUS_CHECK_FAIL	0x0005	Unexpected value in the PCI Status Register. (2)
HE_Base1_NOT_IO	0x0006	Expected HEPC3 PCI config register base 1 to be in I/O space, but it wasn't. (2)
HE_Base2_NOT_IO	0x0007	Expected HEPC3 PCI config register base 2 to be in I/O space, but it wasn't. (2)
HE_Base3_NOT_IO	0x0008	Expected HEPC3 PCI config register base 3 to be in I/O space, but it wasn't. (2)

HE_Base4_NOT_IO	0x0009	Expected HEPC3 PCI config register base 4 to be in I/O space, but it wasn't. (2)
HE_Base5_NOT_IO	0x000a	Expected HEPC3 PCI config register base 5 to be in I/O space, but it wasn't. (2)
HE_Base6_NOT_IO	0x000b	Expected HEPC3 PCI config register base 6 to be in I/O space, but it wasn't. (2)
HE_NOTRAP_OPREG	0x000c	Failed to trap access to HEPC3 Operations Register. (1)
HE_NOTRAP_BDATA	0x000d	Failed to trap access to HEPC3 Comport B Data register. (1)
HE_NOTRAP_ACTRL	0x000e	Failed to trap access to HEPC3 Comport A Control Register. (1)
HE_NOTRAP_BCTRL	0x000f	Failed to trap access to HEPC3 Comport B Control Register. (1)
HE_No_String_IO	0x0010	No support to trap use of String I/O instructions (e.g. outsb). (1)
HE_No_Rep_IO	0x0011	No Support to trap use of Rep I/O instructions (e.g. rep outsb). (1)
HE_No_Addr_32_IO	0x0012	No Support to trap 32 bit I/O instructions. (1)
HE_No_Reverse_IO	0x0013	No support for Reversed string I/O operations. (1)
HE_CloseNotOpen	0x0014	Attempt to Close device that is already closed.
HE_AlreadyOpenOther	0x0015	Attempt to Open a device someone else has open.
HE_CloseNotOurs	0x0016	Attempt to close a device that someone else opened.
HE_IllegalBoard	0x0017	Board Id is invalid (board is not in the system).
HE_IllegalDevice	0x0018	The board does not support the device.
HE_NOTRAP_JTAG	0x0019	Failed to trap access to the Jtag register. (1)
HE_UnexpectedIoTrap	0x001a	Received a trap that we don't recognise. (1)
HE_Unsupported	0x001b	Unsupported Operation.
HE_ReadTimeout	0x001c	Where I/O is not supported by interrupts the I/O terminated because the routine exceeds a max. number of retries.
HE_WriteTimeout	0x001d	Where I/O is not supported by interrupts the I/O terminated because the routine exceeds a max. number of retries.
HE_MMReadInProgress	0x001e	A Master Mode Read was initiated when one was already in progress - this should never happen. (3)
HE_MMWriteInProgress	0x001f	A Master Mode Write was initiated when one was already in progress - this should never happen. (3)

HE_FailedAllocatePerVM	0x0020	Not used.
HE_HEPC8_DETECT_FAIL	0x0021	RTOS32 API: cannot detect HEPC8.
HE_MMAP_FAIL	0x0022	RTOS32 API: unable to map board into virtual address space (GetDevicePtr failed).
HE_IoInProgress	0x0023	A Read or Write is still in progress - returned by `HeRead()`, `HeWrite()` or `HeTestIo()`.
HE_UnknownCommand	0x0026	Not used.
HE_NOTRAP_Interrupt	0x0028	Not used.
HE_AlreadyOpenUs	0x0029	Attempt to open a device we already opened.
HE_OnlyBNoInterrupts	0x002b	Not used.
HE_UnlockNotLocked	0x002f	Not used.
HE_PbUnlockNotLocked	0x0031	Not used.
HE_NULLCallbackAddress	0x0033	Not used.
HE_ReadContinuousOnlyOnMM	0x0035	Not used.
HE_FailedAllocateReadPhysical	0x0037	Not used.
HE_FailedAllocateIRQ9AtCriticalInit	0x0039	Not used.
HE_Fatal	0x003a	Unknown but fatal error.
HE_IOLIB_INIT_FAIL	0x003f	VxWorks API: unable to initialise configuration access-method and addresses for PCI bus.
HE_HEPC9_DETECT_FAIL	0x0040	RTOS32 API: cannot detect HEPC9.
HE_Hep3aVxdMissing	0x1002	Not used.
HE_AllocFailed	0x1003	Failed to allocate memory, or size is 0.
HE_LockFailed	0x1025	Attempt to lock memory handle that was unsuccessfully created by HeAlloc.
HE_OpenFailed	0x1027	Open Failed. Unable to open device driver, typically because board isn't there, incorrect board number, or driver is not installed.
HE_CloseFailed	0x1028	Close Failed. Device driver says it can't close the handle. Use HeGetLastOsError to get an OS specific error code.
HE_NoMMBuffersSpecified	0x1100	RTOS32 API: you try to open a fifo with MasterMode, but you have not specified a MasterMode buffer size in the cfg file.
HE_MMBufferSizeTooLarge	0x1101	Not used.
HE_MMBufAddrTooHigh	0x1102	Not used.

HE_MMBufferSizeTooSmall	0x1103	RTOS32 API: MasterMode buffersize defined in your cfg file must be 4096 bytes or higher.
HE_IllegalCount	0x2001	The I/O operation specified an illegal number of bytes - e.g. size must be multiples of 4 bytes for a Fifo - see `HeGetIoGranularity()`.
HE_IllegalBoardType	0x2002	The function you used does not support the board you specified.
HE_UnknownError	0x2003	Unknown error. Try checking the error returned by the Operating System by using `HeGetLastOsError()`. Might also indicate a problem with the device driver.
HE_FreeFailed	0x2004	Not used.
HE_UnlockFailed	0x2005	Attempt to unlock memory handle that was unsuccessfully created by HeAlloc.
HE_InvalidBuffer	0x2006	Indicates that the API library has given the device driver too small a buffer to execute an i/o request. May be caused by corrupted driver or library, or a software bug.
HE_UnsupportedOperatingSystem	0x2007	The API library does not support the current Operating system.
HE_ResetFailed	0x2008	Attempt to reset the board failed: unable to communicate with the device driver.
HE_JtagReadFailed	0x2009	Attempt to read from JTAG failed: unable to communicate with the device driver.
HE_JtagWriteFailed	0x200a	Attempt to write to JTAG failed: unable to communicate with the device driver.
HE_InitIoStatusFailed	0x200b	Attempt to Initialise an `IoStatus` block failed. Typically because an event or semaphore could not be created or initialised.
HE_WaitForIoFailed	0x200c	The semaphore used to wait for completion of the read or write transfer has returned an error. Use `HeGetLastOsError()` to find out which.
HE_TestIoFailed	0x200d	The semaphore used to wait for completion of the read or write transfer has returned an error. Use `HeGetLastOsError()` to find out which.
HE_GetIoGranularityFailed	0x200e	Attempt to get granularity failed: cannot communicate with the device driver.
HE_GetDriverStatusFailed	0x200f	Attempt to get driver status failed: cannot communicate with the device driver, or the function is not supported on this OS.

HE_ReadFailed	0x2010	HeRead failed. Communication with the device driver failed, or an OS specific operation failed, such as semaphore or event creation/ initialisation. HeGetLastOsError() may tell more.
HE_WriteFailed	0x2011	HeWrite failed. Communication with the device driver failed, or an OS specific operation failed, such as semaphore or event creation/initialisation. HeGetLastOsError() may tell more.
HE_ConfigFailed	0x2012	HeConfig failed. Communication with the device driver failed, or config is not supported for that board or operating system. Use HeGetLastOsError() to find out more.
HE_SignalFailure	0x2013	A general error from routines that are using Semaphores or Events. Use HeGetLast-OsError() to find out more.
HE_WaitEventFailed	0x2014	A general error from routines and threads that are waiting for Semaphores or Events. Use HeGetLastOsError() to find out more.
HE_CreateEventFailed	0x2015	A general error from routines and threads that failed to create a Semaphore or Event. Use HeGetLastOsError() to find out more.
HE_ClearEventFailed	0x2016	A general error from routines and threads that failed to reset a Semaphore or Event. Use HeGetLastOsError() to find out more.
HE_IllegalHandle	0x2017	The function you used does not support the device you specified.
HE_UnableToCreateOpenLock	0x2018	When using the underlying file system to enforce exclusive access to a device, either an error occurred in creating the file or the device is actually already open!
HE_ThreadTerminationError	0x2019	Where underlying threads are used an error occurred terminating a thread when closing a device - see O/S specific error
HE_ThreadTerminated	0x201a	If a thread is successfully terminated on a close this is the status code it will be given - you should never see this as you should never close with outstanding I/Os.
HE_UnableToCloseLock	0x201b	When using the underlying file system to enforce exclusive access to a device, an error occurred in closing the file.

HE_VddAttachFailed	0x201c	When a DOS program is trying to attach to a Virtual Dos Driver (e.g. HENVDD) as a result of a call to HeOpen() in HEXDRV.LIB it can fail - this call will be returned from HeOpen() and hDevice->LastOsError will be set to the failure code - These are probably: - 1. DLL not found - i.e. HENVDD is probably not installed! 2. Dispatch routine not found 3. Init Routine Not Found 4. Insufficient Memory
HE_VddNotAttached	0x201d	An attempt by a DOS program on NT to use an API function without a successful attach to the Vdd by HeOpen()
HE_ExceededMaxDevices	0x201e	Where the API has to restrict the maximum number of Open Devices and this is exceeded then the error will be returned from HeOpen(). In additional hDevice->LastOsError is set to the max that was exceeded.
HE_NoInputAvailable	0x201f	Used by HeTestInputAvailable() to indicate that the incoming fifo is empty.
HE_NullHandlePointer	0x2020	A function is passed a pointer that is NULL.
HE_InvalidHandlePointer	0x2021	A function is passed a pointer to an object that is invalid. The object is not an HE_HANDLE or an HE_IOSTATUS.
HE_HandlePointerNotNull	0x2022	For functions such as HeOpen, the device pointer must be initialised to NULL. This error indicates that a pointer passed to a function is not NULL. The idea is to avoid memory leaks by checking if a pointer, that is to be initialised, is not pointing to an existing object. It does this be checking if it's NULL.
HE_FailedToAllocMemoryForHandle	0x2023	In several different functions, the API tries to allocate memory for a handle. It the memory allocation fails, this error is returned.
HE_HandleInUse	0x2024	Returned when a handle is to be created, but the current handle is already in use.
HE_IoStatusChainNotEmpty	0x2025	Not used.
HE_HandleNotInUse	0x2026	A handle (HE_HANDLE) is used that was not successfully opened, or has been closed.
HE_ReadAlreadyInProgress	0x2027	Used in single-threaded OS to indicate that a previous read transfer has not completed yet.
HE_WriteAlreadyInProgress	0x2028	Used in single-threaded OS to indicate that a previous write transfer has not completed yet.

HE_FailedDeleteRing0EventHandle	0x2029	Not used.
HE_VxDOpenFailed	0x202a	Unable to communicate with the device driver Use `HeGetLastOsError()` to get an error code specific to the OS used.
HE_VxDCloseFailed	0x202b	Unable to communicate with the device driver Use `HeGetLastOsError()` to get an error code specific to the OS used.
HE_VxDCreateFailed	0x202c	Unable to connect with the device driver. The device driver is missing; or board-number or board-type doesn't identify an exisiting board.
HE_UnknownOpenSwitch	0x202d	Used by HeOpenS to indicate that a switch is not recognised or is not supported on the OS that you use.
HE_ByteSwapNotSupported	0x202e	Returned when you specified to swap bytes on devices or boards that don't support that.
HE_ByteSwapFailed	0x202f	Unable to communicate with the device driver Use `HeGetLastOsError()` to get an error code specific to the OS used.
HE_IoCancelled	0x2030	If you close a device while a read or a write transfer has not completed yet, HeWaitForIo and HeTestIo may return this error code.
HE_JtagDisabled	0x2031	With boards such as the HEPC2E you can configure the device driver to not use JTAG addresses. This error indicates that you try to open JTAG while you configured the device driver not to access any JTAG registers.
HE_VTDVxdMissing	0x2032	Not used.
HE_GetIntsStatusFailed	0x2035	Unable to communicate with the device driver or a parameter passed to the function is NULL or the API is configured not to use the device driver.
HE_ModTypeFailed	0x2036	Not used.
HE_IntsStatusFailed	0x2037	Unable to communicate with the device driver Use `HeGetLastOsError()` to get an error code specific to the OS used.
HE_PingIntFailed	0x2038	Interrupt test performed on Open Failed.
HE_GetBoardInfoFailed	0x2039	Unable to open device driver, typically because board isn't there, incorrect board number, or driver is not installed, or unable to communicate with the device driver Use `HeGetLastOsError()` to get an error code specific to the OS used.
HE_InterruptsDisabled	0x2040	Interrupt test performed on Open Failed, because the device driver is configured not to use interrupts.

HE_TimeOut	0x2041	Functions that feature a timeout return this value if it cannot within the specified time.
HE_VxDIsDisabled	0x2042	Not used.
HE_HSBInitFailed	0x2043	Unable to communicate with the device driver Use `HeGetLastOsError()` to get an error code specific to the OS used.
HE_HSBMasterFailed	0x2044	Unable to communicate with the device driver `HeGetLastOsError()` may tell more.
HE_HSBSlaveFailed	0x2045	Unable to communicate with the device driver Use `HeGetLastOsError()` to get an error code specific to the OS used.
HE_HSBListenFailed	0x2046	Unable to communicate with the device driver or a pointer is NULL or the HSB bus reports an error. `HeGetLastOsError()` may give an error code specific to the OS used.
HE_HSBMsgNotForUs	0x2047	Used by `HeHSBReceiveMessageEx()` to indicate that a message was detected, but it was not addresses to us (ie the host device).
HE_HSBReceiveMessage Failed	0x2048	Null pointer in a parameter passed to the `HeHSBReceiveMessageEx()` function.
HE_HSBMsgMissingByte s	0x2049	Expected to receive 2 protocol defined bytes, but when the message completed no or only 1 byte were received. May occur when you send a message to the 'wrong' module and you expect a protocol reply from it.
HE_HSBMsgWrongSlot	0x2050	Expected to receive a message from device identified by 'slot' and 'tgbd', but sender was a different device.
HE_HSBReadFailed	0x2051	Unable to communicate with the device driver or a NULL pointer is passed to the function. `HeGetLastOsError()` may give an error code specific to the OS used.
HE_HSBWriteFailed	0x2052	Unable to communicate with the device driver or a NULL pointer is passed to the function or the HSB bus reports an error. `HeGetLastOsError()` may give an error code specific to the OS used.
HE_InvalidRemoteHand le	0x2053	Not used.
HE_RemoteMallocFaile d	0x2054	Not used.
HE_HSBStartSendMessa ge Failed	0x2055	Not used.
HE_HSBSendMessageDat a Failed	0x2056	Not used.

HE_HSBEndOfSendMessage Failed	0x2057	Not used.
HE_HSBStartReceiveMessage Failed	0x2058	Not used.
HE_HSBReceiveMessageData Failed	0x2059	Not used.
HE_HSBEndOfReceiveMessage Failed	0x205a	Not used.
HE_DetectOperatingSystem RemoteFailed	0x205b	Not used.
HE_HSBNotExistID	0x205c	Returned by HeHSBMaster() on HEPC8 boards if the target device doesn't exist.
HE_GetVersionFailed	0x206c	Not used.
HE_HSBFlushFailed	0x206d	Unable to communicate with the device driver or a NULL pointer is passed to the function. HeGetLastOsError() may give an error code specific to the OS used.
HE_HSBSurplusBytes	0x206e	Returned when HSB is flushed and there were actually bytes read and discarded. The flush operation itself was successful, and this return value isn't really an error code, but more like a remark or warning.
HE_IoctlFailed	0x206f	Unable to communicate with the device driver HeGetLastOsError() may give an error code specific to the OS used.
HE_ReadBusy	0x2070	VxWorks API: used to indicate that a previous read transfer has not completed yet.
HE_WriteBusy	0x2071	VxWorks API: used to indicate that a previous write transfer has not completed yet.
HE_SemCreateFailed	0x2072	VxWorks API: used by HeOpen functions to indicate that the creation of an internal semaphore failed.
HE_DeleteEventFailed	0x2073	Used by HeClose() if it was not able to close the handle to the device driver.
HE_ParameterNullPointer	0x2074	Returned by HeGetBoardInfoEx() if a NULL pointer was passed to the function.
HE_JTAGEnvVarParseFailed	0x2080	An error occurred while parsing JTAG Master and/or Slave environment variables.
HE_JtagConfigureFailed	0x2100	Unable to communicate with the device driver HeGetLastOsError() may give an error code specific to the OS used.

Notes

(1) Direct I/O Errors. On some platforms there is additional support from the drivers to

detect direct I/O operations to the device - i.e. not using the device driver. This is only supported on some 'legacy' C4x drivers.

(2) PCI Specific. These errors are typically signalled where the driver has had to probe the PCI device directly rather than this being done by the Operating System. However some of the errors can occur even when the O/S has already obtained configuration information from the PCI device.

(3) HEPC3 Specific. These error codes are specific to errors from the AMCC chip used on the HEPC3

Introduction

HSB stands for Heron Serial Bus. HSB is implemented on the HEPC8 and HEART based boards such as the HEPC9 and HECPCI9, but not on 'legacy' carrier boards such as the HEPC3 or the HEPC2E. Future HEART carrier boards will most certainly also implement HSB. Software support for HSB was introduced with API version 1.6.

The main purpose for HSB is as a 'control' bus. For example, it can be used to interrogate HERON processor modules for their type (C6701, C6201, C6203, other). HSB is also used to configure HEART, and to upload FPGA bitstreams.

But HSB can also be used by yourself, for general-purpose data exchange, or for your own control purposes. It must be remembered though that the HSB is not only slow, it is also arbitrated, so it cannot be relied upon for real time operation unless your system is carefully defined so that arbitration failures will never occur.

HSB IDs or identifiers

The Heron Serial Bus is a two-wire bus to which several 'devices' are attached. A 'device' may be a slot, a host interface, HEART, or something else. Each 'device' on the Heron Serial Bus has a unique ID that identifies it on the bus.

The HSB ID, or identifier, is a 7-bit number. The highest 4 bits is the board number. The lowest 3 bits is the device specifier: 1 for slot 1, 2 for slot 2, 3 for slot 3, 4 for slot 4, 5 for the host interface, 6 for the inter-board connector and 7 for the HEART device on carriers such as the HEPC9. For example, identifier 0x4 would denote slot 4 on a board with board number 0, identifier 0x13 would denote slot 3 on a board with board number 2, and identifier 0x1F would denote the HEART device on a board with board number 3.

HSB speed

HSB is relatively slow (compared to FIFO speeds). On the HEPC8 it ticks at 100 kHz, and on the HEPC9 at 1 Mhz. Actual achievable bandwidths will be lower than the 100 Kb/sec and the 1 Mb/sec that you perhaps would expect, due to arbitration on the bus.

HUNT HSB Protocol

There are pre-defined messages that can be sent to 'devices' for certain purposes. These messages follow a protocol, ie the bytes in the message have some meaning and must follow certain rules. For example, to ask a processor module what type it is, 2 bytes must be sent to that module over HSB. The first byte must be '1', the second byte the 'return address'. The 'return address' is the HSB ID of the sender, in this case, as we are sending the HSB message from the host PC, that would be '5' (assuming a board number of 0).

In the HUNT HSB protocol, the first byte in a message is called the 'message type'. A full list of HERON serial bus message types can be found in the HSB specification document. (The file is named 'hsbspec.pdf' and usually located on the HUNT CD in the \web\pdfs\tech directory. You can also use the HUNT CD front-end, "User Manuals" → "Technology Documents" → "HERON Serial Bus Specification". Review Appendix 1.)

Non HSB Protocol Messages

It is not the case that all HSB messages must follow the HUNT HSB protocol. Between two processor modules that run your own program you are completely free to exchange any HSB message you like. Similarly, between the host and a processor module that runs your own program you can exchange any HSB message you like.

On processor modules, after a system reset, a tiny program is loaded from flash memory. One purpose of this program is to load a bootstream over a fifo. But it also contains code that answers HSB messages: type '1' queries. Thus, as long as a processor module is not booted, you have to follow the HUNT HSB protocol and you can only send HSB messages that follow the HUNT HSB protocol to that module. But as soon as a processor module is booted with your own program, it can send or receive any HSB message – it's completely up to you to program whatever HSB exchange you like.

With devices such as a HEART device, or an inter-board connector, the HSB protocol response is hard-coded into the device. So you can only send HSB messages to a HEART device or an inter-board connector that follow the HUNT HSB protocol.

FPGA modules fall somewhat in between the above to cases. FPGA's are programmable, so there's a degree of freedom, but also they must always respond in the same way to an update bitstream request, so that part is hard-coded.

Accessing the Heron Serial Bus

To write or read the serial bus, first open it as usual with an HeOpen call, use device name "hsb" or a digit 3. The API offers 3 types of access methods, called 'level3', 'level 2' and 'level 1'. The highest level is easiest to use but can only be used with HUNT HSB protocol messages. It consists only of two functions: send a message and receive a message. If the message is non HUNT HSB protocol or if the message should ideally be sent or received in parts, use 'level 2' functions. Each 'level 3' function can be functionally split up in 3 'level 2' functions. The 'level 3' send message function splits into a 'start send message', a 'send message data', and an 'end of send message' function in 'level 2'. There are (thus) 6 'level 2' functions.

The 'level 1' functions are low-level functions and you are advised not to use them.

Level 3 Serial Bus functions

These functions are "high level" serial bus functions that are easy to use, and you don't need to know the particulars of how the serial bus works. But they can only be used for messages that follow the HUNT HSB protocol.

As usual with the HUNT ENGINEERING API, to access the serial bus you need to open it. You can use a HeOpenS with device name "hsb" or a different open with device 3. The HUNT HSB protocol operates by first sending a message to a slot (a "request for some action"), where the message specifies what action is required of the module in that slot. E.g. for a processor module, a first byte of 1 specifies an information request. The processor module then responds by sending a message with its heron module type, processor type, etc. To send a message, the following function is provided:

```
HeHSBSendMessageEx(HE_HANDLE hDevice, HE_BYTE msg_type, HE_WORD tgbd
                   HE_BYTE slot, void *data, HE_DWORD size, int msec);
```

hDevice is the handle returned from the open call. The msg_type is the message type, eg 1

when you ask a processor to return information. The tgbd parameter is the board switch of the board the target slot is on. Together with the slot parameter tgbd will be used to create an hsb id (tgbd<<3+slot). The slot parameter is the slot number where the module is fitted: 1, 2, 3 or 4. The host device is seen as 'slot' 5, an inter-board connector as 'slot' 6, and the HEART device as 'slot' 7. The data pointer would point to any data you want to send. For processor queries this isn't used, so fill the pointer as being NULL or set the size of the data array to 0 (size = 0). Finally, a message might timeout (eg the serial bus is busy for an extended period, perhaps it's used by other modules in between them). You can specify how long at most you're prepared to wait.

To receive messages coming in, use the following function:

```
HeHSBReceiveMessageEx(HE_HANDLE hDevice, HE_BYTE *msg_type,
                      HE_WORD tgbd, HE_BYTE slot, void *data,
                      HE_DWORD size, HE_DWORD *read, int msec);
```

In this case, msg_type is the message type of the message received. The tgbd and slot parameters form the hsb-id of the device that you sent the original message to. The data buffer receives the incoming message. The size parameter tells the function how many bytes you expect. The read parameter tells you how many have actually been read. Finally, msec allows you to specify a timeout on how long at most you're prepared to wait for a serial bus message.

Example. Query a processor module using level '3' functions

```
void TestHSB(char *devname, int Board, int heronid)
{
HE_BYTE           msg_type, reply[8];
int               count, slot, tgbd;
HE_DWORD          switches[1];

switches[0]=HE_Switch_Last;
hDevice = NULL;
Status = HeOpenS(devname, (HE_WORD)Board, "hsb", &hDevice, switches);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));
printf("HSB opened %lx\n", Status);

slot   = heronid&0xf;
tgbd   = (heronid>>4)&0xf;
Status = HeHSBSendMessageEx(hDevice, 1, tgbd, slot, NULL, 0, 1000);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));
printf("Write protocol, status was %lx\n", Status);

count  = 0;
Status = HeHSBReceiveMessageEx(hDevice, &msg_type, tgbd, slot, reply,
                              5, &count, 1000);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));
printf("Count           : %d.\n"      , count);
printf("Message type    : %d.\n"      , msg_type);
printf("Module type     : 0x%x\n"     , reply[0]);
printf("Heron module    : HERON%d\n"  , reply[1]);
printf("Processor type  : 0x%x\n"     , reply[2]);
printf("Boot ROM version: 0x%x\n"     , reply[3]);
printf("Option          : 0x%x\n"     , reply[4]);

Status = HeClose(&hDevice);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));
}
```

Level 2 Serial Bus functions

This level of serial bus functions allows you more flexibility. With level 3 functions you must send a serial bus message as a whole, and they can be used only when following the HUNT HSB protocol. With level 2 functions, you can split up a message and send it in parts. And level 2 functions can be used for non HUNT HSB protocol messages. The HeHSBSendMessageEx function can be seen of consisting of 3 'level 2' functions:

```
HeHSBStartSendMessageEx(HE_HANDLE h, HE_WORD tgbd,HE_BYTE slot,int ms);
HeHSBSendMessageDataEx (HE_HANDLE h, void *data, HE_DWORD Cnt, int ms);
HeHSBEndOfSendMessageEx(HE_HANDLE h, int ms);
```

HeHSBStartSendMessageEx prepares the serial bus to send a message to the hsb device identified by parameters 'tgbd' and 'slot'. Parameter 'slot' is typically 1, 2, 3 or 4. If the serial bus is busy for longer than 'ms' milliseconds, the function may timeout. Use a value of 0 for 'ms' if you don't want a timeout.

HeHSBSendMessageDataEx sends the actual message. Be aware, though, that this message is not equal to the message as used in HeHSBSendMessageEx. The HeHSBSendMessageEx function sends two extra bytes: the 'message type' and the 'return address', before it sends the actual message. With the HeHSBSendMessageDataEx function, when following the HUNT HSB protocol, you are expected to supply a data array where the first two bytes are the 'message type' and the 'return address'. Bytes 3 and higher are the message you would have used in HeHSBSendMessageEx. The return address is always 5 plus the board number left shifted 3 ((board number<<3)+5), because the API is run on the host PC (and the host PC HSB device identifier is 5).

The HeHSBSendMessageDataEx may also timeout, and in parameter 'ms' you can specify the number of milliseconds. In the previous case, HeHSBStartSendMessageEx, a timeout would typically be due to an arbitration timeout, ie the serial bus was busy. When sending a data array with HeHSBSendMessageDataEx, a timeout is usually due to the target module not being in the targeted slot. A timeout can also occur because the targeted module hasn't been coded to accept HSB messages, or in the case of the HUNT HSB protocol, if the target isn't programmed to accept or understand the message type you specified (for example, if you try to send a 'program bitstream' HSB message to a processor module).

Finally, you end a message with HeHSBEndOfSendMessageEx. The 'ms' parameter specifies a timeout value in milliseconds, and relates to how long you are prepared to wait for the arbitration (to 'release' the serial bus) to complete.

Similarly, HeHSBReceiveMessageEx can be seen of consisting of 3 'level 2' functions:

```
HeHSBStartReceiveMessageEx(HE_HANDLE h, HE_DWORD *id, int ms);
HeHSBReceiveMessageDataEx (HE_HANDLE h, void *data, HE_DWORD size,
                                                 HE_DWORD *read, int ms);
HeHSBEndOfReceiveMessageEx(HE_HANDLE h, int msec);
```

HeHSBStartReceiveMessageEx prepares the serial bus to receive a message from the HSB. The sending device will be returned in parameter 'id'. A timeout value can be specified in milliseconds in the 'ms' parameter. If a timeout occurs, it is because no message has been detected to arrive for 'ms' milliseconds.

HeHSBReceiveMessageDataEx receives the actual message. Parameter 'size' denotes how many bytes you expect to receive; parameter 'read' will return the actual number of bytes that were received. Parameter 'ms' allows you to specify a timeout in milliseconds.

With the HUNT HSB protocol, you would only try to receive a message after you have sent a protocol message to a target device. Depending on the message type, you would try to receive a message. A timeout could be due to the target module not being in the targeted slot. It may also occur if the target isn't programmed to accept or understand the message type you specified in the sent message.

With non HUNT HSB protocol messages, a timeout may also occur because the targeted module hasn't been coded to send or reply to HSB messages.

HeHSBEndOfReceiveMessageEx will complete the transfer. If there are any bytes left to read, they are read and then discarded.

Let's see how the earlier example would look like in 'level 2' functions.

Example. Query a processor module using level '2' functions

```
void TestHSB(char *devname, int Board, int heronid)
{
HE_BYTE           reply[8];
int               slot, tgbd, toi2c, bdi2c;
HE_DWORD          switches[1], count, id;
HE_BYTE           prot[2];

switches[0] = HE_Switch_Last;
hDevice = NULL;
Status = HeOpenS(devname, (HE_WORD)Board, "hsb", &hDevice, switches);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));
printf("HSB opened %lx\n", Status);

slot   = heronid&0xf;
tgbd   = (heronid>>4)&0xf;
Status = HeHSBStartSendMessageEx(hDevice, tgbd, slot, 1000);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));
prot[0] = 1;                  // Message Type (see heapi.h)
prot[1] = (0x5|(Board<<3));   // Board's HSB id: 0x5 + board switch<<3
Status = HeHSBSendMessageDataEx (hDevice, prot, 2, 1000);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));
Status = HeHSBEndOfSendMessageEx(hDevice, 1000);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));

count = 0;
toi2c = (slot|(tgbd<<3));     // HSB id that we expect message from
bdi2c = ( 0x5|(Board<<3))<<1; // Board's HSB id: 0x5 + board switch<<3
Status = HeHSBStartReceiveMessageEx(hDevice, &id, 1000);
if (Status  != HE_OK) error(HeGetLastOsError(hDevice));
if (id      != bdi2c) printf("This message is not for us.\n");
Status = HeHSBReceiveMessageDataEx (hDevice, reply, 7, &count, 1000);
if(Status   != HE_OK) error(HeGetLastOsError(hDevice));
if (count   != 7    ) printf("Expected at least seven bytes.\n");
if (reply[1]!= toi2c) printf("Sender %x, not %x??\n", reply[1], toi2c);
Status = HeHSBEndOfReceiveMessageEx(hDevice, 1000);
if(Status  != HE_OK) error(HeGetLastOsError(hDevice));
printf("Count            : %d.\n"      , count);
printf("Message type     : %d.\n"      , reply[0]);
printf("Module type      : 0x%x\n"     , reply[2]);
printf("Heron module     : HERON%d\n"  , reply[3]);
printf("Processor type   : 0x%x\n"     , reply[4]);
printf("Boot ROM version: 0x%x\n"      , reply[5]);
printf("Option           : 0x%x\n"     , reply[6]);

Status = HeClose(&hDevice);
if(Status != HE_OK) error(HeGetLastOsError(hDevice));
}
```

For non-protocol messages, there's a worked out example on the HUNT CD. It uses one HERON processor module, where the HERON API is used to receive and then return HSB messages in exchange with a host program. The example is located on the CD in cd:\software\examples\host_api_examples\c6x\hsb. The host code is in this directory; the HERON processor code is in the 'dsp' sub-directory. Via the CD front-end you can find the example via: Getting Started → Getting Started with C6000 modules and tools → Host API examples. This will open Windows Explorer and put you in the host_api_examples directory.

Level 1 Serial Bus Functions (lowest level)

Level 2 HSB functions are expressed in level 1 HSB functions. As it will be harder to use level 1 functions we would therefore recommend that you use level 2 functions. Level 1 functions are explained here just for completeness.

In all cases, you would first open the HSB device, and then initialise the serial bus, using "HeHSBInit()". However, in some cases you may want to do a system reset first (using a different device, "fifoa"). For example, on processor modules, answering serial bus queries is done by a small boot program that gets downloaded from flash memory after a reset. If the processor has already been booted with a user program, and that user program doesn't implement serial bus queries, then you won't get any answer to your queries. Therefore, it's sometimes best to first reset ("fifoa") device, then to open the serial bus, and only then do the "HeHSBInit()".

Following the "Query a processor module" example in the previous sections, to obtain information from processor modules, you would want to write a few bytes to it tell the target processor to return some information. This is a protocollized process and you cannot just write any value you like – unless you implemented your own serial bus protocol. The standard HUNT HSB protocol is 2 bytes: a digit 1 followed by the serial bus id of the motherboard you're using: 5. If the board switch were set at 1, the serial bus id would be 13. If the board switch were set at 2, the id would be 21, etc. (Id=(board switch<<3)+5).

But before writing the protocol value, first you need to become master of the serial bus. This is done with the "HeHSBMaster()" function. This function has a timeout; if after some time the function still was unable to become serial bus master, it times out. The user can set the timeout value.

Once you're master, you can write the protocol to the serial bus, using "HeWrite()". If you write to e.g. a FPGA device, simply write all the bytes to want to upload to the device.

After writing to the serial bus, it's time to release it and free the bus. This is done with the "HeHSBSlave()" function. As with "HeHSBMaster()", there's a timeout associated with it.

In the case of processor modules, we would now want to read the processor module's response. First, we need to wait for the arrival of the "start" of the message. This is done with the "HeHSBListen()" function. Basically, the processor module tries to become serial bus master and then writes to it. The "HeHSBListen()" waits until the message arrives, and gives you the serial bus ID the message is aimed at.

Finally, after a successful call to "HeHSBListen()", you can use "HeRead()" to read the response. "HeRead()" assumes that you know how long the message is. If this is inconvenient, you can read byte-by-byte, and use HeTestIo() to time out if there are no more message bytes arriving.

An example of usage now follows.

```
int serial_bus(char *devname, int Board, int slot)
{
      HE_IOSTATUS ReadIoStatus  = NULL;
      HE_IOSTATUS WriteIoStatus = NULL;
      char        *Comport = "hsb";
      HE_DWORD    switches[2];
      int         i;
      HE_BYTE     protocol[4];
      HE_BYTE     reply[8];

      hsbDevice       = NULL;
      ReadIoStatus   = NULL;
      WriteIoStatus  = NULL;

      switches[0]=HE_Switch_Last;

      /* Open the serial bus device */
      Status=HeOpenS(devname,Board,Comport,&hsbDevice, switches);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      /* Create an object for reading the serial bus */
      Status=HeInitIoStatus(hsbDevice, &ReadIoStatus);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      /* Create an object for writing the serial bus */
      Status=HeInitIoStatus(hsbDevice, &WriteIoStatus);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      Sleep(250);

      /* Initialise the device */
      Status=HeHSBInit(hsbDevice, 0x5);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      /* Try to become serial bus master */
      Status=HeHSBMaster(hsbDevice, slot, 1000);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      /* Initialise the protocol buffer for querying a HERON        */
      /* processor module                                            */
      protocol[0] = 1;        // protocol type
      protocol[1] = 0x5;      // return HSB address (HEPC8 0)

      /* Now write the protocol buffer onto the serial bus */
      Status=HeWrite(hsbDevice, protocol, 2, WriteIoStatus);
      if(Status==HE_IoInProgress)
            Status = HeWaitForIo(hsbDevice, WriteIoStatus);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      /* Done writing. Release the serial bus */
      Status=HeHSBSlave(hsbDevice, 1000);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      /* Listen for an answer… */
      Status=HeHSBListen(hsbDevice, &i, 1000);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      /* There was an answer. Now read it. */
      Status = HeRead(hsbDevice, reply, 7, ReadIoStatus);
      if(Status==HE_IoInProgress)
            Status = HeWaitForIo(hsbDevice, ReadIoStatus);
      if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

      /* A HERON processor module returns 7 bytes as follows:
```

```
        byte 0: response type (2)
        byte 1: serial bus ID from sender of message
        byte 2: module type (1=processor)
        byte 3: heron module type (1=HERON1, 4=HERON4)
        byte 4: processor type (1=C6201, 2=C6701)
        byte 5: boot ROM version (3 at the time of writing)
        byte 6: option (can be any, but usually 0)
    */
    printf("Serial bus:slot %d: HERON%d-%s, rom version %d.\n",
            reply[1],reply[3],GetProcessor(reply[4]),reply[5]);

    /* Done. Close the device. */
    Status = HeClose(&hsbDevice);
    if(Status!=HE_OK)return error(HeGetLastOsError(hsbDevice));

    return reply[3];
}
```

HERON Serial Bus Message Types

A full list of HERON serial bus message types can be found in the HSB specification document. (The file is named 'hsbspec.pdf' and usually located on the HUNT CD in the \web\pdfs\tech directory. You can also use the HUNT CD front-end, "User Manuals" → "Technology Documents" → "HERON Serial Bus Specification". Review Appendix 1.)

PLEASE NOTE: The JTAG is not supported under VxWorks and RTOS-32.

The API interface to a JTAG device on one of the HUNT ENGINEERING host boards is performed via two functions `HeJtagRead()` and `HeJtagWrite()`. These functions perform an address mapping relevant to that board type, and pass these accesses through to the target board.

Where it can (i.e. anywhere except Windows NT) the DLL will directly pass reads and writes to the board's JTAG interface having first mapped the address properly for that board type.

The interface to the `HeJtag` functions uses the following mapping for the JTAG interface registers:

TI Debugger Symbolic Name	Address with which to call the HeJtagRead/HeJtagWrite fnc
Group1	
SC_REG_CNTL0	0x0000
SC_REG_CNTL1	0x0002
SC_REG_CNTL2	0x0004
SC_REG_CNTL3	0x0006
SC_REG_CNTL4	0x0008
SC_REG_CNTL5	0x000A
SC_REG_CNTL6	0x000C
SC_REG_CNTL7	0x000E
SC_REG_CNTL8	0x0010
SC_REG_CNTL9	0x0012
SC_REG_MINOR_CMD	0x0014
SC_REG_MAJOR_CMD	0x0016
SC_REG_CNT1_LOW	0x0018
SC_REG_CNT1_HIGH	0x001A
SC_REG_CNT2_LOW	0x001C
SC_REG_CNT2_HIGH	0x001E
Group2	
SC_REG_STATUS0	0x0020
SC_REG_STATUS1	0x0022
SC_REG_STATUS2	0x0024
SC_REG_STATUS3	0x0026
SC_REG_CAPT_LOW	0x0028
SC_REG_CAPT_HIGH	0x002A
SC_REG_SERIAL_RD	0x002C
SC_REG_SERIAL_WR	0x002E
SC_REG_XL0	0x0030
SC_REG_XL1	0x0032
SC_REG_XL2	0x0034
SC_REG_XL3	0x0036
SC_REG_XL4	0x0038
SC_REG_XL5	0x003A
SC_REG_XL6	0x003C
SC_REG_XL7	0x003E

Group3

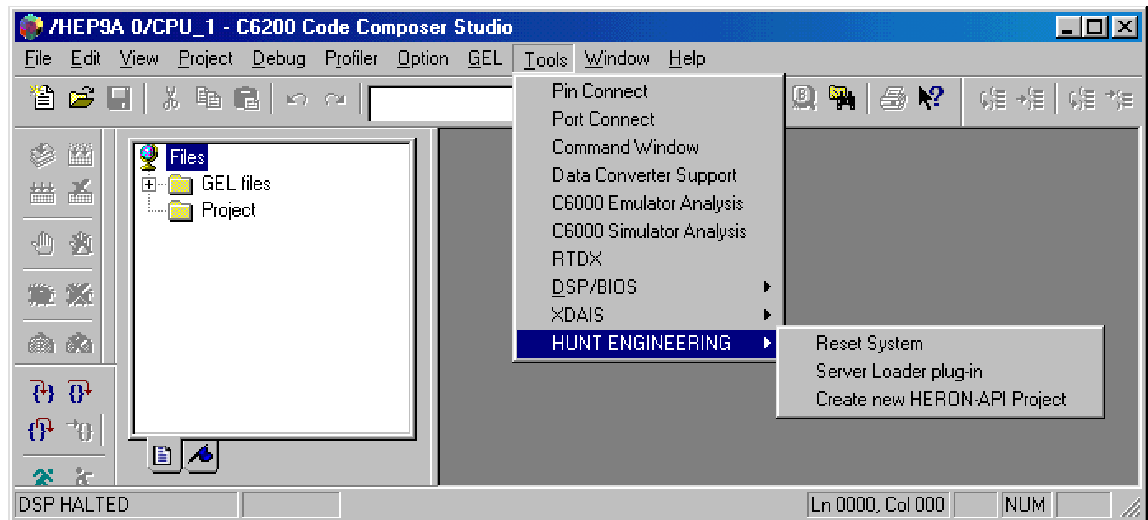
`SC_REG_CONTROL0                    0x0040`

The `HeJtag` functions encode the 16 bit JTAG register offset and the 16 bit JTAG data into one variable. The `HeJtagWrite` routine takes an `HE_DWORD` in which the low 16 bits is the data to be written and the high 16 bits is the required register offset from the table above.

The `HeJtagRead` routine takes a pointer to an `HE_DWORD`, which is initialised to have the register offset from the table above in the high 16 bits. The function returns the requested data in the low 16 bits of the `HE_DWORD` pointed to.

What is a plugin?

A plugin is a program that can be called from within Code Composer Studio. HUNT ENGINEERING provides several, and they are accessible from the 'Tools' menu in a Code Composer Studio's processor window.



The 'Server Loader plug-in' belongs to the HUNT ENGINEERING Server/Loader package and is further discussed in Server/Loader documentation.

Please note that the plugins only work with Code Composer Studio (as opposed to the plain Code Composer). This implies that only 'C6x HERON systems are supported.

A plugin is very tightly integrated with Code Composer Studio. Plugin source code can 'call' a number of selected Code Composer Studio 'functions', like for example 'set a breakpoint', 'run to breakpoint', and 'load a file'. Therefore, a plugin allows us to integrate our hardware and software with Code Composer Studio.

For example, the 'Reset System' plugin will reset HERON systems. This is a HUNT ENGINEERING specific action, not natively supported by Code Composer Studio. But with a plugin this HUNT ENGINEERING specific functionality is seamlessly added to Code Composer Studio.

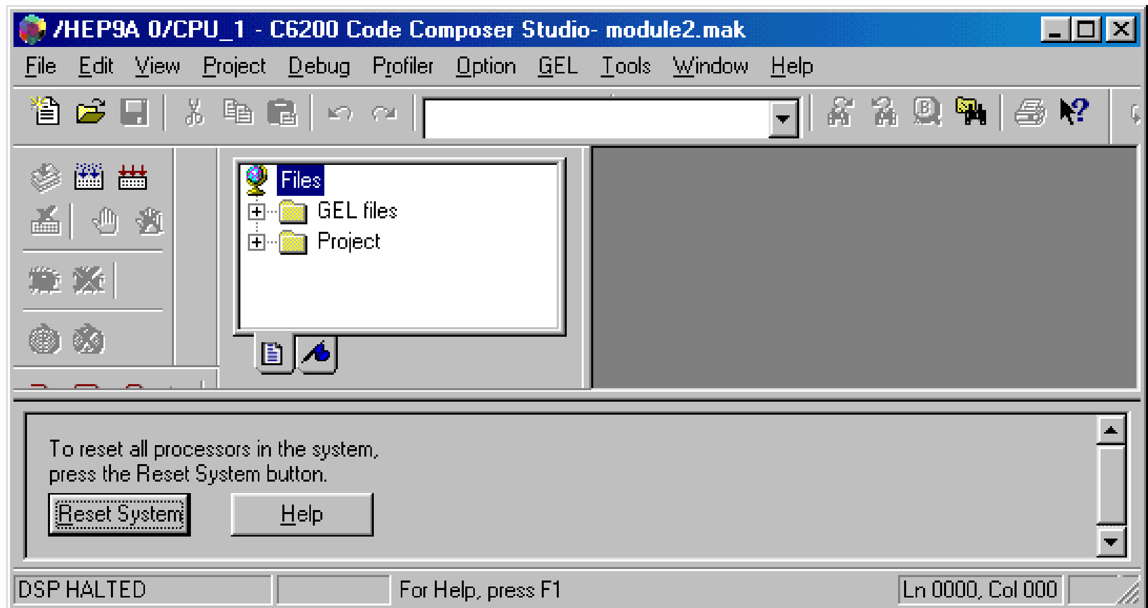
The 'Reset System' Plugin

What does it do?

The 'Reset System' plugin will reset the HERON system hardware. Optionally, it can then reload your source file(s) and run to 'main'. (See the options section later in this section.)

How do I start it?

Go to the Code Composer Tools menu and select Tools → HUNT ENGINEERING → Reset System. A new window will be created within the Code Composer Studio window, at the bottom of this window. The plugin will work with multiple processors.

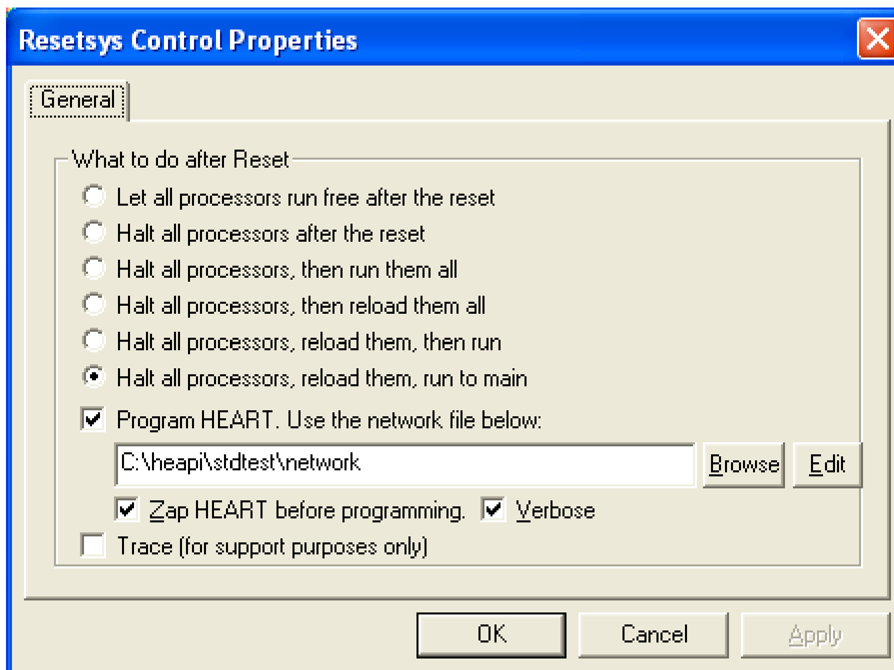


How do I use it?

Right-click on any part of the grey area of the plug-in, select "Property Page" from the menu that has now appeared. Select what you want the plug-in to do after the actual system reset (see the options section below for details). Then click "OK". Next, to reset the system, click 'Reset System'.

Options

There are a number of options for the plugin. You can reach the options window by right-clicking on the plugin's grey area. A menu will appear. Click 'Property Page'.



Let all processors run free after the reset

The plugin will do a 'RunFree' on all processors, then do a HERON system reset.

Halt all processors after the reset (default)

The plugin will do a 'RunFree' on all processors, do a system reset, then halt all processors.

Halt all processors, then run them all

The plugin will do a 'RunFree' on all processors, do a system reset, then halt all processors. Then it will put all processors in 'Run' mode.

Halt all processors, then reload them all

The plugin will do a 'RunFree' on all processors, do a system reset, then halt all processors. If a processor was previously loaded, it will attempt to do a 'Reload' so that the original file is again loaded. Execution will be at the entry-point of the file (ie 'c_int00').

Halt all processors, reload them, then run

The plugin will do a 'RunFree' on all processors, do a system reset, then halt all processors. If a processor was previously loaded, it will attempt to do a 'Reload', then a 'Run'.

Program HEART. Use the network file below.

After the reset, the plug-in can run 'HeartConf' to (re-)create HEART connections between the modules. Please be aware that a system reset will also clear all HEART connections. For non-HEART boards such as the HEPC8 this option isn't used.

Use the 'Browse' button to select a network file. Use the 'Edit' button to alter an existing or to create a new (template) network file.

Use 'Zap HEART before programming' by default. Only if you use jumpers to 'hard-wire' HERON FIFO's to certain timeslots, leave this tick box un-ticked. A jumpered connection will exist even after a reset. Simply adding HEART connections (from a network file) may cause unexpected connections. The 'Zap HEART before programming' will first erase all pre-existing HEART connections. If there's HERON module that uses a jumper, then this setting will not make any difference.

To create the HEART connections, the reset plug-in will execute 'HeartConf' in a DOS-box. If there's no error, the DOS-box closes again. Upon error, the DOS-box will remain open until you press a key or click the box away. In error situations it may be helpful to use the 'Verbose' option, as this will ask 'HeartConf' to output some progress information.

Trace (for support purposes only)

Ticking this box will create a file 'sl' in the root directory ('\') of the current drive. This is only there for support purposes, ie when for some reason the reset plug-in doesn't work properly and HUNT technical support needs a trace file to find out what the plug-in is doing.

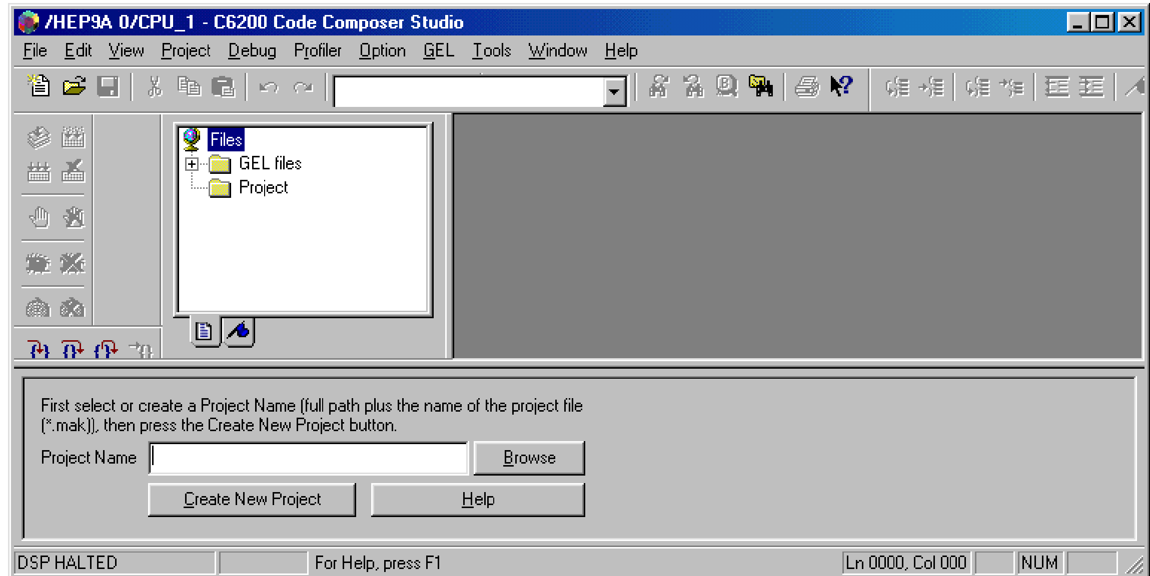
The 'Create new HERON-API Project' plugin

What does it do?

The 'Create new HERON-API Project' plugin will create a new Code Composer Studio project that is ready-for-use with HUNT ENGINEERING 'C6x HERON based systems.

How do I start it?

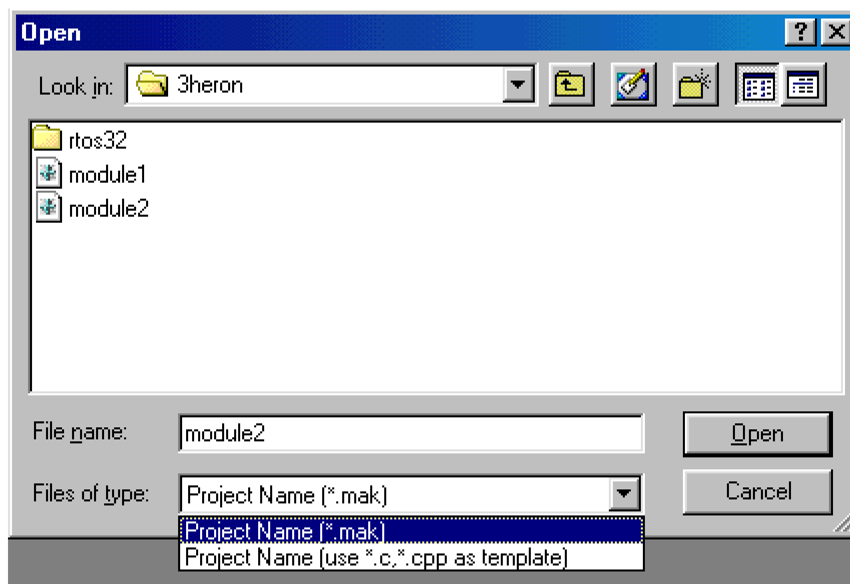
Go to the Code Composer Tools menu and select Tools → HUNT ENGINEERING → Create new HERON-API Project. A new window will be created within the Code Composer Studio window, at the bottom of this window.



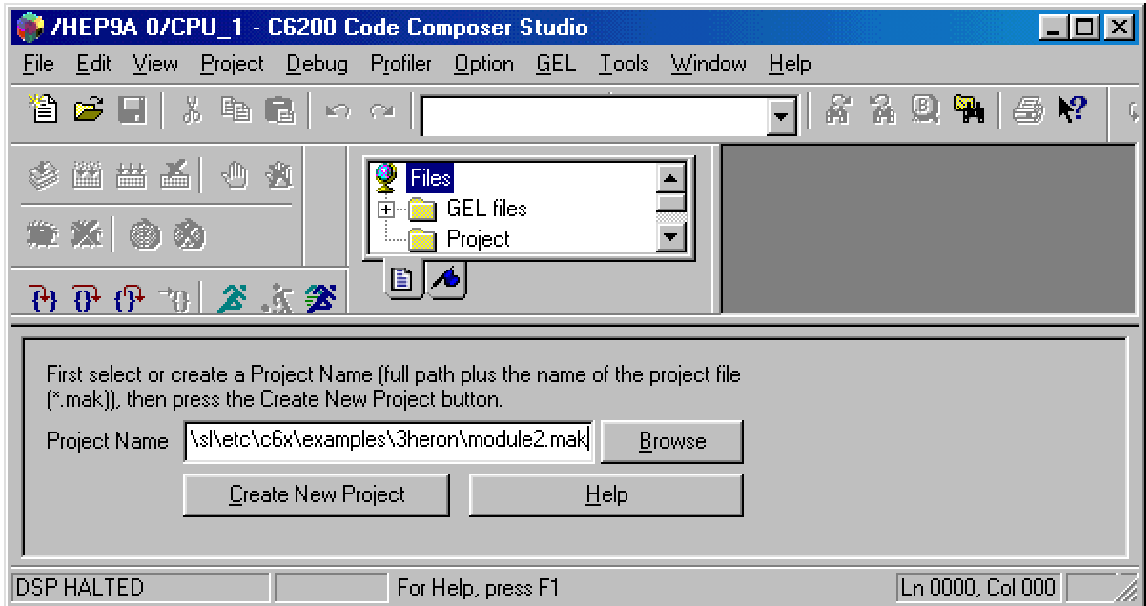
How do I use it?

First, you need to select a directory where you wish to create and keep your new project. If you want to create a new project using a HUNT ENGINEERING example, please first use e.g. Windows Explorer to copy the example from the HUNT ENGINEERING CD into any directory on your harddisk. For example, copy "cd:\software\examples\starting_development*.*" to "c:\ti\myprojects\demo".

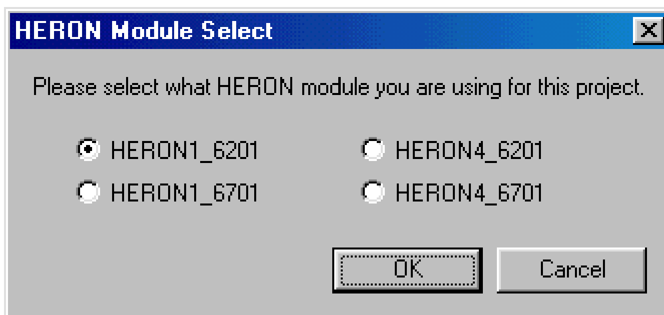
Type the full path + project name in the 'Project Name' edit box; or use the 'Browse' button to browse to your directory. Using the 'Browse' button, once you arrived in the directory of your choice, type the name you want to give to your project in the 'File name' edit box. You can also 'overwrite' an existing project by selecting an existing '.mak' file. You can also select a '.c' or '.cpp' file. The project name will be derived from the selected file name.



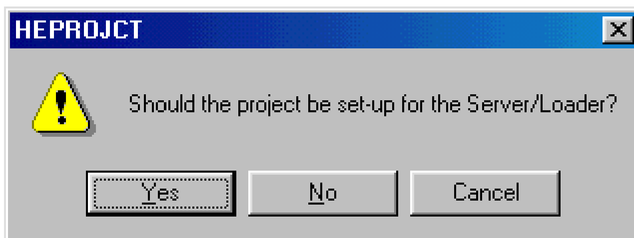
If you choose a name for the project that is the same as the name of a source file (apart from the extension), then this source file will automatically be added to the project. If there is no source file with an identical name, a new source will be created with the same name as the name of the project. In the case of the example above, the "Project Name" edit box would read "c:\ti\ myprojects\demo\demo.mak".



When a proper path plus project name has been typed in the 'Project Name' edit box, press the 'Create New Project' button. This will now create a new project. Two questions will be asked (when creating a brand-new project). First, it will ask you to select a HERON module type. Select the module type you want to create the project for; typically it would be the HERON module that the current Code Composer Studio processor window is running on.



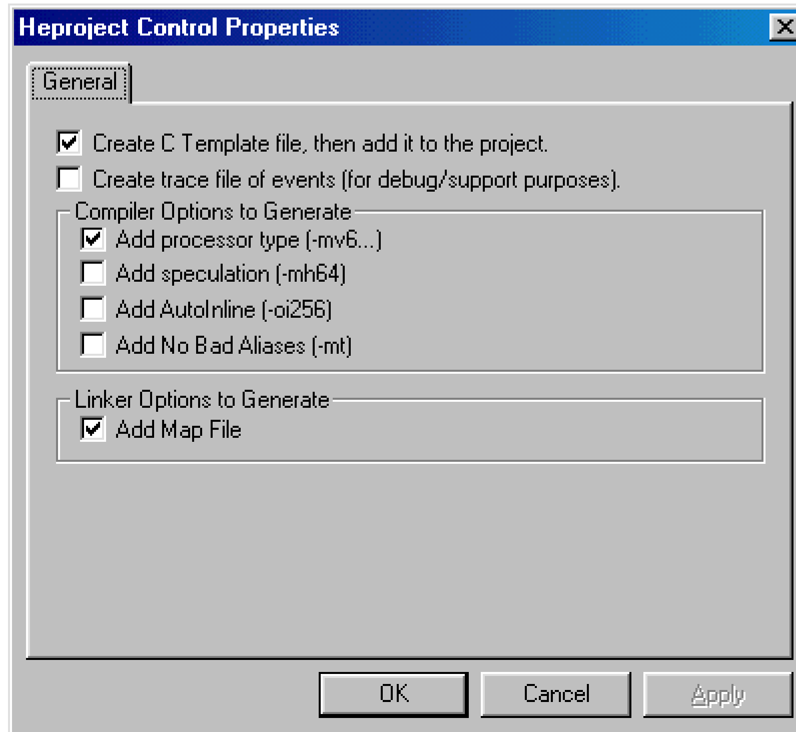
The second question will be if you want to create the project for the Server/Loader. The Server/Loader is HUNT ENGINEERING software that allows you to easily load and serve a network of DSP processors. The Server/Loader uses a small network file (ASCII format) that describes the DSP processor network. The Server/Loader works by having a call to a 'bootloader' function in the C file's 'main' routine. To debug 'Server/Loader' applications, you need to use the 'Server/Loader' plugin to start up a debugging session.



Click ‘No’ if you create a project that only needs to use Code Composer Studio. It is quite easy to upgrade a Code Composer Studio project to a Server/Loader application later on.

Options

There are a number of options for the plugin. You can reach the options window by right-clicking on the plugin’s grey area. A menu will appear. Click ‘Property Page’.



Create C Template file, then add it to the project

By default, the ‘Create new HERON-API project’ plugin will add a C file that has the same name as the project to the project. If there is no such C file, a template C file will be generated and added to the project. If you don’t want a template C file to be generated in such cases, please unclick this box.

Create trace file of events

For support purposes, it is possible to create a file that logs what actions are undertaken. The file created is named ‘’.

Add processor type

The ‘Create new HERON-API project’ plugin adds some compiler options to the project. It will add a ‘-mv6201’, ‘-mv6701’ etc. option depending on the HERON type you selected. Unclick this box if you want to have no such processor type compiler option added

Add speculation (-mh64)

The ‘Create new HERON-API project’ plugin adds some compiler options to the project. This option is used to get the compiler to create higher performance code. For more info on what the option does, please refer to TI’s compiler manual. Tick the box if you want the plugin to generate a new project with this option.

Add AutoInline (-oi256)

The 'Create new HERON-API project' plugin adds some compiler options to the project. This option is used to get the compiler to create higher performance code. For more info on what the option does, please refer to TI's compiler manual. Tick the box if you want the plugin to generate a new project with this option.

Add No Bad Aliases (-mt)

The 'Create new HERON-API project' plugin adds some compiler options to the project. This option is used to get the compiler to create higher performance code. For more info on what the option does, please refer to TI's compiler manual. Tick the box if you want the plugin to generate a new project with this option.

Add Map File

Be default the plugin will create a new project, configured to create a map file upon 'Build'. A map file is created by the linker, detailing what software sections are placed where, on what physical memory sections. A map file is helpful debugging stack, heap, section overflow and other kinds of problems. If you don't want the plugin to create projects that configure the new project to have map files, unclick this box.

What actions does the plugin perform?

Create a new project.

First, the plug-in will create a new, empty, project, and immediately close it again.

Add compiler and linker options to the mak file

It will then edit the make file (*.mak) and change some of the compiler and linker options. The plug-in will add the "-mi100", "-o3", and possibly other options (see the options section above), and will add include paths to the HERON-API and (optionally) the HUNT ENGINEERING Server/Loader include directories. After a successful edit of the make file, the project is then re-opened.

Add CDB file

Next, the plug-in will add a CDB file to the project. The CDB file will be chosen from the HERON-API's cmd directory, based on the choices you made. These CDB files have been created to reflect the hardware configurations of the different HERON modules.

Add linker command file

Then a linker command file will be added to the project. The linker command file will be chosen from the HERON-API's cmd directory, based on the choices you made. This linker command file includes (using the "-l" option) the "standard" linker command file created by the Code Composer Studio / DSP/BIOS configuration compiler (gconf). This is necessary in order to make the linker understand the non-default code and data sections created by the HERON-API.

Add libraries

The plug-in will then add the appropriate HERON-API and (optionally) Server/Loader library to the project.

Add C file or C template

Finally, a source file is added to the project. If there is a source file in the project directory with the same name (apart from the extension) as the project name, that source file will be added to the project. If there is no such source file, the plug-in will create a new source file (template) and add it to the project. For example, if the project name is "demo.mak", the plug-in will add to the project a C file named "demo.c".

If you don't want to have a template C file added to your projects, the Property Page has an option that allows you to disable this facility. The Property Page is reached by right-clicking on the grey area of the plug-in, and then selecting Property Page in the pop-up menu that should have appeared.

For most HUNT ENGINEERING example programs, you can now build the project. These example programs have only a 'maintask', and the CDB files are configured to include a 'maintask'. A few examples have more entities than just a 'maintask'. For example, the 'Starting Development' example uses two 'SWI' objects, and you would need to edit the CDB file and add two 'SWI' objects. Such details will be explained in the document that go with the example programs.

Example Programs

Examples are provided on the HUNT ENGINEERING CD. For LINUX, the examples are embedded in the tar file (the tar file is on the CD). In this case, the examples are in a sub-directory 'etc' under the main API installation directory.

To use the examples, copy the example directory tree across from the CD onto your harddisk. Use Windows Explorer to do so, or use a DOS box. In this case, to copy the whole example directory tree over into the API installation, type:

```
xcopy d:\software\examples\host_api_examples c:\heapi /s
```

(Assuming 'd:' is your CD drive, and your installation directory is 'c:\heapi'.)

The examples each have instructions on how to build and run them, per supported operating system. The instructions are in PDF documents located in the example directories. The HERON example programs will be in sub-directory 'c6x'. Legacy example programs will be in sub-directory 'c4x'.

Technical Support

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section http://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 http://www.hunteng.co.uk for the list of current re-sellers.

HUNT ENGINEERING technical support can be contacted by emailing support@hunteng.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.