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.demon.co.uk
URL: http://www.hunteng.co.uk

We are a
committed
member of the
Texas Instruments
3rd party
programme

For Sales and Support in North America Please Contact Our Strategic Partner:

Traquair Data Systems Inc, 114 Sheldon Road, Ithaca, NY 14850 USA

Tel 607 266 6000, Fax 607 266 8221

Email Traquair@traquair.com, URL http://www.traquair.com

For Sales and Support in Other Areas Please Contact Your Local Reseller.

HUNT ENGINEERING

API Reads Example

Description and Reference

Document Rev A
API Reads Example Rev 1.0
J.Thie 05-03-01

COPYRIGHT

WARRANTIES LIABILITY and INDEMNITIES

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

> Exclusions - If HUNT ENGINEERING decides that there is any evidence of electrical or mechanical abuse to the hardware, then the customer shall have no recourse to HUNT ENGINEERING or its agents. In such circumstances HUNT ENGINEERING may at its discretion offer to repair the hardware and charge for that repair.
>
> Limitations of Liability - HUNT ENGINEERING makes no warranty as to the fitness of the product for any particular purpose. In no event shall HUNT ENGINEERING'S liability related to the product exceed the purchase fee actually paid by you for the product. Neither HUNT ENGINEERING nor its suppliers shall in any event be liable for any indirect, consequential or financial damages caused by the delivery, use or performance of this product.

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

TECHNICAL SUPPORT

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/support.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.

TABLE OF CONTENTS

What does it do?

The ‘reads’ example is an example program that tests the PCI FIFO and HSB interface of a HERON carrier board. The example will try to boot a small and simple program onto the first processor (on the module in slot 1). This program will send a stream of known data to the host, sent via the PCI interface. The host example program will read the stream of data and tell you if everything worked or not. It will also give a very rough estimate of the transfer speed.

Supported Operating Systems

The HUNT ENGINEERING API supports several different platforms. For each of the supported operating systems there is a sub-directory in which a document explains how to compile and link the example for that particular operating system. For example, in the win32 sub-directory you will find instructions how to compile and link the example using a 32-bit Microsoft Visual C/C++ or Borland C/C++ compiler.

Using one source code set for all the different operating systems shows the platform independence of the HUNT ENGINEERING API. There are some platform dependencies in the example code itself, though, as different include files may be needed, or some operating systems need an initialisation routine, etc.

Supported Boards

The HUNT ENGINEERING API supports all HUNT ENGINEERING carrier boards. The COFF loader code used in the example originates from TI. The COFF loader code can ‘understand’ *.out files and will output ready-to-send code chunks that can be downloaded on to the processor. However, the COFF loader code supports either ‘C6x or ‘C4x, but not both at the same time. This example chooses ‘C6x. Therefore, even though the HUNT ENGINEERING API supports all HUNT ENGINEERING carrier boards, due to the COFF loader code, the example will only work with ‘C6x based systems, such as those systems that use the HEPC8 or the HEPC9.

DSP Code

The source code of the small program that gets loaded onto the first processor in the system is in sub-directory ‘dsp’. The code is there so that you can see what it does, and so that you can understand the example better. You must not use the code in sub-directory ‘dsp’ as an example how to create dsp programs. To create dsp programs, please use Code Composer Studio, DSP/BIOS and HUNT ENGINEERING tools. Please have a look at the examples on the cd: \software\examples\heron_api_examples.

Warning

Before you rush off, and use this example to write your own loader program, please be attended to the fact that the HUNT ENGINEERING Server/Loader will boot and serve a network of dsp processors. The Server/Loader software comes in both .exe and library format, so it is very easy using the Server/Loader within your own host application. If what you want is to boot a network of processors, then using the Server/Loader library will get you up and running much quicker. Please have a look at the examples in the Server/Loader examples directory (on the cd: \software\examples\server_loader_examples\c6x, especially 'mysl' and the two examples in 'sl_api').

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.

Open / Close

Before you can access a device, you must claim the device. Different devices must be claimed seperately. The ‘HeOpen’ and ‘HeOpen1’ calls are generic device open functions. The difference is that ‘HeOpen1’ expects a character string to identify the device you want to open, and ‘HeOpen1’ expects an integer identifier. A third open function is ‘HeOpenS’. This function is used to ‘hide’ operating system quirks. For example, in an operating system like VxWorks, the open function needs to know for ISA boards what interrupt and address to use. This information is can be transferred using ‘HeOpenS’, which takes an array of 32-bit specifiers as extra input parameter. Therefore, you would not usually need this function. The documents in the operating system specific sub-directories will show you when ‘HeOpenS’ needs to be used.

Writing

To write data to a carrier board, use the ‘HeWrite’ function. Please note that this function is asynchronous. The ‘HeWrite’ merely starts a write transfer, but not necessarily completes it. 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
```

The ‘HeWrite’ only initiates a write transfer. The actual write transfer will proceed somewhere ‘in the background’. This may be an independent thread, interrupts, or something else. Conceptually the transfer proceeds in parallel with your code. Therefore, there is a certain amount of time between the ‘HeWrite’ call and the completion of the write transfer. This time can be used by your application do do something else (typically some processing of data previously read from the carrier board).

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. Please note that this function is asynchronous. The ‘HeRead’ merely starts a read transfer, but not necessarily completes it. 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 ‘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’, 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
```

The ‘HeRead’ only initiates a read transfer. The actual read transfer will proceed somewhere ‘in the background’. This may be an independent thread, interrupts, or something else. Conceptually the transfer proceeds in parallel with your code. Therefore, there is a certain amount of time between the ‘HeRead’ call and the completion of the read transfer. This time can be used by your application do do something else (typically some processing of data previously read from the carrier board).

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
```

HSB (serial bus)

The serial bus is a relatively slow bus in addition to the fast FIFO interface. The serial bus is typically used to retrieve module information or to configure something (examples: FPGA and HEPC9 FIFO connections). The ‘reads’ example code shows how the serial bus can be used to retrieve processor module information.

To use the serial bus, you need to adhere to a protocol. Simply sending any buffer of data is not likely to work. Both sender and receiver adhere to a protocol. This protocol is defined in detail in the ‘API – Reference Manual’. Only if both sender and receiver are ‘your’ code, no pre-defined protocol is needed; essentially you define your own.

Generally, to send an HSB message, you first need to send the adressee ID. The ID is made up of 8 bits. Bit 7 is not used. The next highest 4 bits are the carrier ID (red switch on for example the HEPC8). Bits 2,1,0 are the slot id.

For small messages adhering to the pre-defined HUNT ENGINEERING protocol, the ‘level-3’ functions are easiest to use. However, if you want to define your own protocol, or you want to send not-so-small data buffers, use level-2 or level-1 HSB messages (selected by #define ‘LOW_ LEVEL’ in the example code). In ‘heapi.h’ there will be comments identifying HSB level functions:

```
/* Level 1 HSB functions */
HeHSBInit(HE_HANDLE hDevice, HE_BYTE hsb_id);
HeHSBMaster(HE_HANDLE hDevice, HE_BYTE hsb_id, int msec);
HeHSBSlave(HE_HANDLE hDevice, int msec);
HeHSBListen(HE_HANDLE hDevice, HE_DWORD *id, int msec);
/* Level 2 HSB functions */
HeHSBStartSendMessage(HE_HANDLE hDevice, HE_BYTE slot, int msec);
HeHSBSendMessageData(HE_HANDLE hDevice, void *data, HE_DWORD Count,
                     int msec);
HeHSBEndOfSendMessage(HE_HANDLE hDevice, int msec);
HeHSBStartReceiveMessage(HE_HANDLE hDevice, HE_DWORD *id , int msec);
HeHSBReceiveMessageData(HE_HANDLE hDevice, void *data, HE_DWORD size,
                        HE_DWORD *read, int msec);
HeHSBEndOfReceiveMessage(HE_HANDLE hDevice, int msec);
/* Level 3 HSB functions */
HeHSBSendMessage(HE_HANDLE hDevice, HE_BYTE  msg_type, HE_BYTE slot,
                 void *data, HE_DWORD size, int msec);
HeHSBReceiveMessage(HE_HANDLE hDevice, HE_BYTE *msg_type, HE_BYTE slot,
                  void *data, HE_DWORD size, HE_DWORD *read, int msec);
```

When using level-1 HSB functions, use ‘HeWrite’ and ‘HeRead’ to do the actual writing of buffers and reading of buffer data, repectively.

HeAlloc

Some operating systems require that any data buffers are 32 bit aligned. Simply declaring an array of 32 bit data items (HE_DWORDs) is not enough. Use ‘HeAlloc’ and ‘HeLock’ to get a pointer to a suitably aligned block of 32-bit words. The ‘HeAlloc’ will initialise a suitably-aligned area of memory; ‘HeLock’ will provide you with a pointer to the memory area. Use this pointer in all API reads and reads.

If you find that alignment is no problem for your particular operating system, an array or a malloc’d chunck of memory will work just as well. However, if you have to deal with any porting issues (eg you have to support more than 1 platform) then portability is enhanced using the ‘HeAlloc’ and ‘HeLock’ functions.

Include file

Source files that use HUNT ENGINEERING API function calls must have ‘heapi.h’ included in the file. There are no other include files that need to be included.

```
#include <heapi.h>
```

Library

Projects that use the HUNT ENGINEERING API should be linked with the HUNT ENGINEERING API library. This has a different name, depending on the operating system and/or compiler used. For example, when using a win32 Microsoft Visual C/C++ compiler creating a Windows executable, link with ‘hendrv.lib’. But when using the win32 Borland compiler, link with ‘hebdrv.lib’. There are no other libraries that need to be linked.

Environment variable

The HUNT ENGINEERING API installation program will generate an environment variable ‘HEAPI_DIR’ that points to the directory in which you installed the API.

Technical Support

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/support.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.