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
www.hunteng.co.uk
www.hunt-dsp.com

Using HERON-API to access HERON FIFOs from a C6000 module.

Rev 1.0 P.Warnes 22-7-03

The main function of HERON-API is to allow your C6000 program to access the HERON FIFOs that are used to interface with other nodes in your system.

Although there are many examples for using HERON-API and the different models it provides, users have requested that there is this tutorial to introduce the concepts.

The tutorial starts from the C code provided, that uses a simple waitio model, and makes changes to that to demonstrate some of the issues that should be understood.

This tutorial requires that you have a HEART based module carrier board like the HEPC9.

History

Rev 1.0	First written

Code Composer Studio.

Code Composer Studio is the development tool for C6000 programs, and you should have already followed the Code Composer Studio tutorials provided by TI, and the "Starting your development" tutorial provided by HUNT ENGINEERING.

DSP/BIOS

DSP/BIOS is the multi-threading environment provided as part of the Code Composer development Environment. It also provides services for configuring processor features such as hardware interrupts and timers. As it is included in Code Composer Studio, along with the Compile tools for the C6000, all users of HERON hardware will own it.

The program for each processor will use DSP/BIOS to configure the multi-tasking etc for that DSP. Simply program and use CCS to compile the application for each DSP separately. This results in a single .out file that contains all of the interrupt service routines etc for that processor.

Each processor can access it's end of a HERON FIFO independently from the other end of the FIFO that will be accessed by another node.

HERON-API

HERON-API is the communications library that HUNT ENGINEERING provides to perform the inter-processor and processor to I/O communications. Its purpose is to prevent the user from needing to intimately understand the communications mechanism, by providing an optimised way to use the limited DMA resources of the C6000 in a choice of ways.

It also serves the purpose of providing a common software interface to the various C6000 HERON modules that HUNT ENGINEERING produce or plan to produce. The hardware of those modules will be different but the HERON-API interface will not.

The main part of HERON-API is to communicate via the HERON FIFOs. For this it provides an asynchronous I/O model, protected by an open and close mechanism. To use a FIFO it must first be "opened" for read or write using HeronOpenFifo, and a read or a write can be ***started*** using HeronRead or HeronWrite. This just schedules that I/O to take place, using the DMA resources of the processor. Meanwhile the processor core is free to execute other tasks such as processing the last buffer of data.

Fifo_example

Copy the source file fifo_example.c from the CD (you can click on the files link from the CD menu, or look for the directory heron_api_examples\fifo_example .

Using the techniques that you have already learnt in the "starting your development" tutorial, make a new HERON-API project in Code Composer Studio that uses this source file.

If you have made the project properly using the CCS plug in, the line

#include "heronx.h"

will be set for the module type that you have.

This is how the correct HERON-API library is included in your project, e.g. if you have a heron4 this line will be

#include "heron4.h"

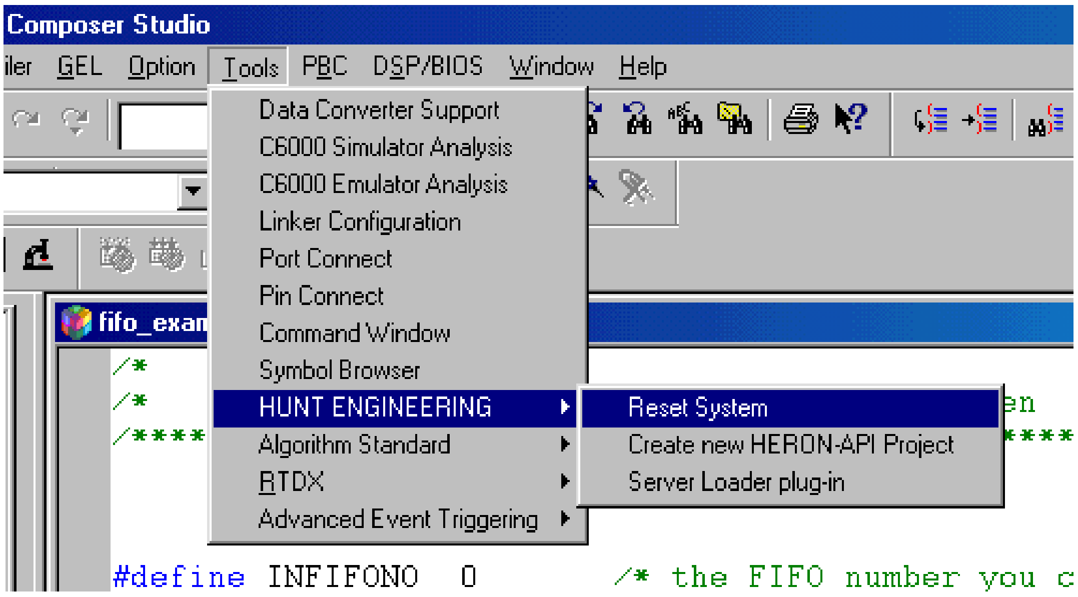
If you look in the source code you will find call to the open function and the write, read and waitio functions.

The example uses the #defined variables INFIFONO and OUTFIFONO to define which fifo numbers will be used.

The example will send data to an output FIFO, and read the results from an input FIFO before checking them. This expects there to be a connection between those FIFOs so that the same data that is sent is received. To make this connection we use the HEART communications system.

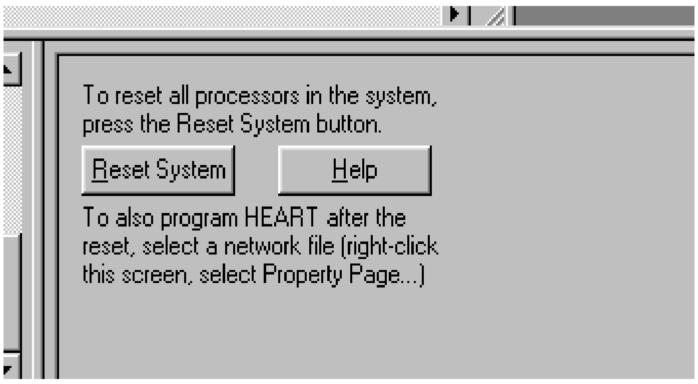
You should have already viewed the movie presentation about using HEART. If you have not done so - then you should do that now.

From within Code Composer open the Reset plug in by following



```
#define INFIFONO   0        /* the FIFO number you c
```

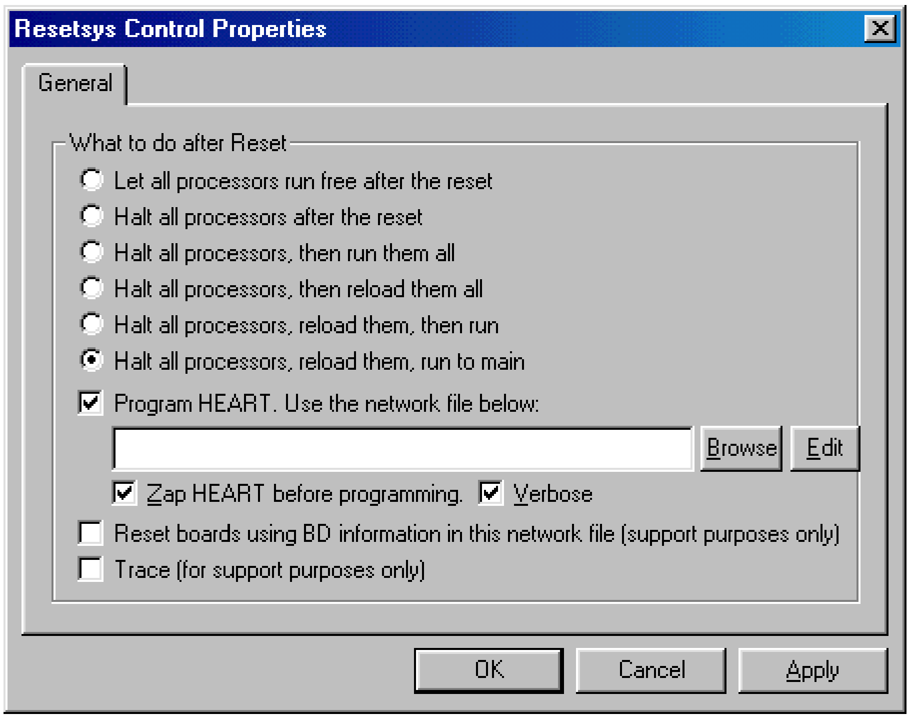
Then you will see the reset plug in



To reset all processors in the system, press the Reset System button.

To also program HEART after the reset, select a network file (right-click this screen, select Property Page...)

Where you can right click to see the properties page :-



What to do after Reset

- ○ Let all processors run free after the reset
- ○ Halt all processors after the reset
- ○ Halt all processors, then run them all
- ○ Halt all processors, then reload them all
- ○ Halt all processors, reload them, then run
- [✓ ●] Halt all processors, reload them, run to main

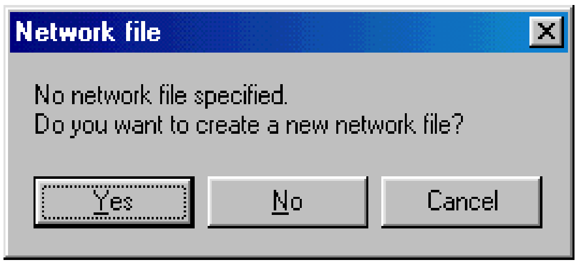
[✓] Program HEART. Use the network file below:

[✓] Zap HEART before programming. [✓] Verbose

[ ] Reset boards using BD information in this network file (support purposes only)

[ ] Trace (for support purposes only)

Notice that the "Program HEART" should be ticked. We need to generate a network file where we can define the FIFO connection that we need. If you have the filename box empty like in the picture you can click on the edit button. You will see :-



No network file specified.
Do you want to create a new network file?

If you click yes, you will be asked to enter a path and file name to be used for the network file. Choose a location and you will be shown a template network file. We need to edit this file to provide the FIFO connection for our example.

There should be a line

```
BD API hep9a 0 0
```

Which selects that we are using an HEPC9 with the switch set to 0.

We need to define the node that is our C6000 module. There is one commented out

```
# c6   0    dsp0   ROOT       0x01  filename
```

We need to uncomment this line, and to set the correct slot number to match our system. For example if the C6000 module is fitted to module slot 2 in our system then we need to have :-

```
c6   0    dsp0   ROOT       0x02  filename
```

The c6 is a keyword and must not be edited, the 0 is the board entry in our list of boards – in this case only one board so 0 is the only option. The name dsp0 is a user friendly name for the node – you can choose whatever you prefer. The 0x02 is the HERON-ID which should be a combination of the board number (in this case 0) and the slot number. The filename is only relevant if we use the Server/Loader. For full information about the network file please refer to the document "Network File Syntax" found under user manuals on the HUNT ENGINEERING CD.

Now we need to make a FIFO connection. To do this we edit one of the HEART lines to be :-

```
heart     dsp0  1     dsp0    0       1
```

which connects the output FIFO number 1 to the input FIFO 0, with one timeslot. Again refer to the documentation for details.

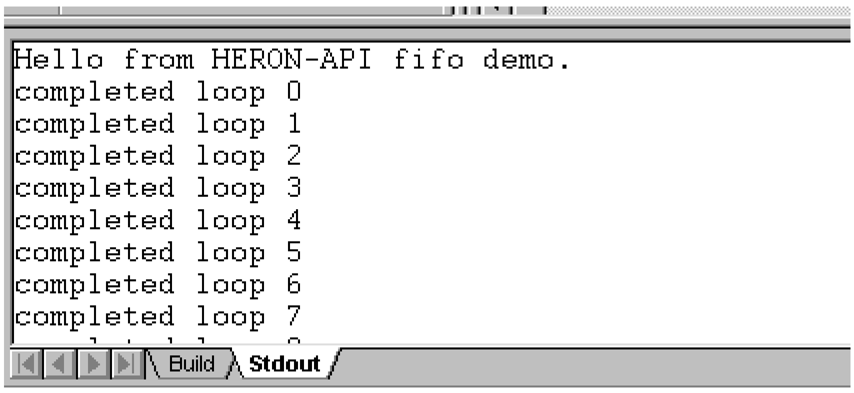
Save this file.

Notice that the reset plug in is able to re-load the processor, but not to load it in the first place. So now we should load the processor.

To do this choose File->Load Program from the Code Composer menus, and then select the file fifo_example.out from the debug directory..

Now you can use the Reset System button on the reset plug in. Using the option selected in the picture above this will reset the C6000 module hardware and the HEART FIFOs, will then connect the FIFO connections according to our network file, and re-load the DSP program. Then your source file will be opened at the main function.

Select run and you should see



```
Hello from HERON-API fifo demo.
completed loop 0
completed loop 1
completed loop 2
completed loop 3
completed loop 4
completed loop 5
completed loop 6
completed loop 7
```

If you do not then the FIFO connection made in your network file does not match the #defines in the source code for your C6000 program. It is very important that these setting match.

As a further exercise try increasing the buffer size to 1024. This doesn't work – can you work out why?

Look at the source code. In fact you can halt the processor and you will probably find that the processor is stuck in the waitio function.

The code currently makes the following sequence:-

Write

Wait for write to complete

Read

Wait for read to complete

But we are now sending more data than fits into the HERON FIFOs, so the write cannot complete. Hence we get stuck at the "Wait for write to complete". This shows how it is important to use the waitio function only when we are certain to complete.

We can make our example work again, by moving the line that starts the read to be immediately after the line that starts the write, i.e.

```
wstatus = HeronWrite(outfifo,pattern,BUFFER_SIZE);
if (wstatus != HERON_IO_IN_PROGRESS)
                {
                 printf("error from write\n");
                exit(0);
                }
/* now start to read the data back from the FIFO */
rstatus = HeronRead(infifo,results,BUFFER_SIZE);
if (rstatus != HERON_IO_IN_PROGRESS)
                {
                printf("error from read\n");
                exit(0);
                }
```

Now the read and write happen at the same time, and hence both can complete.

The danger with using the waitio case, is that that function will block the processor completely. If you have multiple tasks for example, a task that uses waitio will not be de-scheduled. For this reason it is better to use the Semaphore model.

Semaphore example

To demonstrate how the semaphore model can be used in this case, there is a second example in the source file semfifo_example.c.

Make another new HERON-API project like you did above, but this time use the C file semfifo_example.c as the basis for the project.

This example uses 2 tasks and semaphores to signify when the transfers are complete. For this reason you must modify the DSP/BIOS configuration before you can build the example.

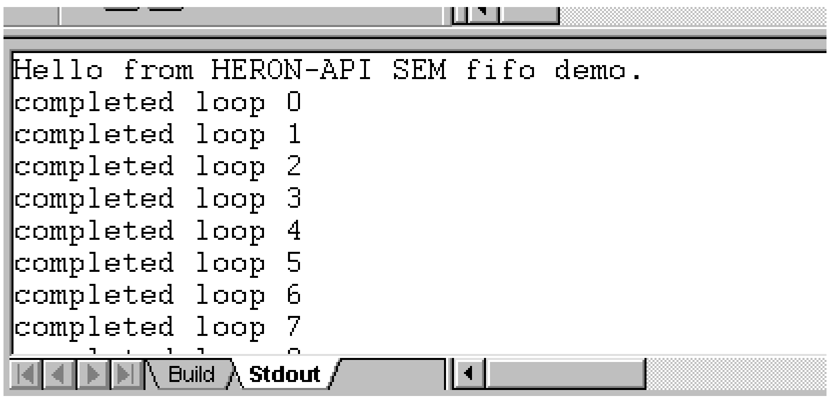
Open the .cdb file in Code Composer and add another task. Set the function property of this task to be _readtask (the underbar is important!)

Add two Semaphores, and rename then as “read0” and “write0”

Now you can build the example.

Remember to load the semfifo_example.out initially, but then to use the reset plug in to load the system. As long as you use the same definitions for INFIFONO and OUTFIFONO as in the earlier example, the same network file can be used as before. If not then you need to make the appropriate changes.

You should be able to run this example and see :-



```
Hello from HERON-API SEM fifo demo.
completed loop 0
completed loop 1
completed loop 2
completed loop 3
completed loop 4
completed loop 5
completed loop 6
completed loop 7
```

In this case the tasks are being de-scheduled when the semaphore is not set, allowing the other task to run until it too becomes de-scheduled.

So now if we increase the BUFFER_SIZE definition to be 1024 and recompile, the example still runs.

This shows the advantage of using a semaphore model rather than a waitio model. There are other models offered by HERON-API and DSP/BIOS, and ultimately you need to select the right model for your application.