



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



## Starting DSP software development for HERON C6000 products.

Rev 3.3 P.Warnes 27-3-07 (removed autoconf)

The HUNT ENGINEERING C6000 Software Developers Pack (SDP) is a development environment that allows you to develop applications for your HERON system. The Framework includes development tools provided by TI, development tools provided by HUNT ENGINEERING and various other components that are designed to make developing your DSP system software easier. Other parts of the framework are for use on the host machine. The aim of the framework is to allow you to develop your application quickly and easily, while not having to learn details of the hardware. By maintaining a level of hardware independence the framework protects your investment in development tools, learning time, and software development. By being non-hardware specific you can benefit from the continual development that HUNT ENGINEERING and TI are doing.

Each component of the framework comes with its own documentation and, in some cases, examples. This document is not intended to be a replacement for those other resources, but merely a description of how you would use them to start developing the DSP part of your application. All development will be made using Code Composer Studio, even if the final system will be deployed using only the Server/Loader.

Separate discussions and examples are provided for host side application development.

## **Code Composer Studio.**

At the heart of the framework is Code Composer Studio. This is provided by TI and is a very comprehensive development environment for the C6000. They provide tutorials and good documentation for CCS, which we recommend you follow to discover the full power of your development environment. Also there is extensive on-line help provide by CCS.

CCS however is a generic development environment for the C6000, and requires plug ins to extend it to be useful with your HERON hardware.

There are some things that are specific to using Code Composer Studio with a HERON system which we discuss here.

## **Installation**

Code Composer Studio should be installed on your system, using the installation programs supplied by TI on the CCS CD.

Following the C6000 SDP installation on the HUNT ENGINEERING CD will install the board drivers and the Server/Loader tool, make some confidence checks, and a number of HUNT ENGINEERING plugins that will be available from the Tools menu in Code Composer Studio.

The HUNT ENGINEERING installation program will also set an environmental variable HEAPI\_JTAG to indicate which board type the CCS drivers should access. The installation will set this to be the last board type that you have installed, but you can edit this setting in autoexec.bat if you wish.

## **Setting up Code Composer Studio**

To use Code Composer Studio with a HUNT ENGINEERING carrier board and C6000 modules, you need to use a third-party JTAG emulator, such as Texas Instrument's XDS510. To configure Code Composer Studio, please follow the instructions that come with your emulator.

## **Versions of CCS.**

All HERON C6000 system users will be offered the Software Developers Pack. This includes the complete version of Code Composer Studio. This has the part number TMDSCCS6000-1.

The HUNT tools will support older versions of CCS, although we recommend that you upgrade to the latest version, and at least to version 2.0 or higher.

In the past there was a version of Code Composer Studio that came with the TI EVM. This had part number TMDX3246855-07 and was called CCS-Compile Tools. This version will not work with hardware other than the EVM, and must be upgraded. Attempting to use this version with your HERON system will give an error message stating "hardware not supported".

If you have this version then it will be necessary for you to upgrade CCS to be able to use the HERON system.

## **DSP/BIOS**

DSP/BIOS is the multi-tasking and 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.

DSP/BIOS provides priority based scheduling of threads and tasks. The threads are called Software Interrupts or SWIs. You can have as many SWIs as you like in your system, and you can assign each a priority. There are up to 15 priority levels available, and you can have multiple SWIs of the same priority. The tasks are called TSKs, and can also have 15 different priority levels all of which are lower than the SWI priorities.

A SWI is "posted", that is, marked as able to execute when the right conditions exist, by the function SWI\_post(). If a SWI is posted that has a higher priority level than the one that is running, it will begin execution immediately, and the currently executing SWI will be marked as able to continue when the conditions are right for it to do so. A SWI that is the same or lower priority than the one currently running will not be executed until all SWIs of a higher priority are completed.

Because a SWI that is running can only be de-scheduled if it posts a higher priority SWI, it is important that each SWI should run to completion.

The only other time that a SWI will be de-scheduled is when a Hardware Interrupt (HWI) occurs. All HWIs are higher priority than SWIs. The priority of the HWIs is set by the processor hardware.

A TSK is started when the system starts, that is, marked as able to execute when the right conditions exist. Once a task completes it cannot be re-started, so it should loop for as many times as you want it to run. The task will be de-scheduled if a higher priority item is able to run, or if the task becomes blocked. One way of blocking is to use a semaphore. Task priorities are lower than SWIs and hence lower than HWIs.

A quirk of this scheme is that the DSP/BIOS is not started until the function main has exited. This

gives rise to a strange look to your program as "main" is normally almost empty, and the rest of your program is written as functions that do not appear to get called.

In fact the calling of these functions is done by setting up HWIs, SWIs and Tasks in the DSP/BIOS configuration tool.

## **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.

This is the component of the framework that makes it possible to use multiple processors and to interface with I/O modules.

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.

### **WaitIo/TestIo model**

The status of the I/O can be determined at any time by calling HeronTestIo which will indicate if the I/O is complete or still in progress. If your processor has no more work to do until the I/O is complete then you can use HeronWaitIo which will not return until the I/O has completed.

### **SWI/SEM model**

Another way of determining the completion of an I/O is for HERON-API to actively notify you of the completion. The functions HeronSwiOpenFifo and HeronSemOpenFifo take parameters that are the DSP/BIOS SWI and SEM objects that should be "posted" when the I/O is completed.

The use of a SWI allows a routine to be run each time an I/O completes i.e. using threads

The use of a SEM allows a Task to be blocked (using SEM\_Pend) until the I/O is complete.

### **SIO model**

Another way of handling the completion of I/O is to use a Streaming I/O driver. This model uses multiple buffers for data, and automatically moves onto the next buffer without user interaction. The user then requests buffers from the SIO object and will be passed the buffers in order. If there is no buffer available the SIO\_Get function will block until there is a buffer ready. There are some choices to be made about how the situation where all buffers are used will be handled (see HERON\_API documentation).

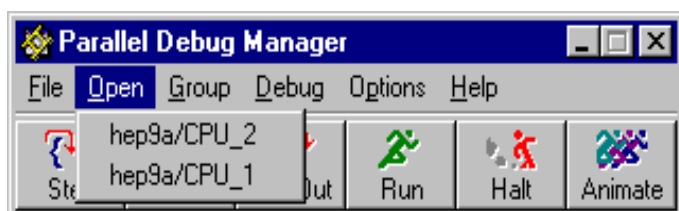
This model is intended for use communicating with I/O modules that cannot be stopped waiting for a communication (e.g. an A/D).

HERON-API is fully integrated and dependent upon DSP/BIOS. This means that you must learn how to set up the DSP/BIOS environment to properly use the HERON\_API.

For complete information of how to use the features of HERON-API refer to the user documentation for HERON-API.

## Starting Code Composer Studio

The installation of CCS will install icons on the desktop of your development machine. Clicking on one of those will start the Setup CCS program, the other Icon will start CCS. If you have configured it for multiple processors you will actually start the Parallel Debug Manager (PDM). From there you can use “open” and choose which processor from the list you wish to open. Notice that with CCS 1.2 in this list the processor names are reversed from that in the Setup CCS. So in this list the lower numbered HERON slot is at the end!



In the above picture the PDM of CCS 1.2 is shown. You can see CPU\_1 is listed bottom.



With CCS 2.x the PDM is as shown above. The list of processors is as you expect with CPU\_1 listed at the top.

Opening one of the processors will result in a window for that processor that will be the same as the window that will open if you have a single processor configured. In a multi-processor case you can open as many windows as you like.

The following sections of this document refer to the CCS window for each processor.

If you get an error message when starting CCS, it is probably because the JTAG circuitry has been left in an undefined state. To fix this use the “Start → Programs → HUNT ENGINEERING → API JTAG reset” function provided by the API installation. If you continue to experience problems then use the “Start → Programs → HUNT ENGINEERING → API board reset” function provided by the API installation followed by the API JTAG reset. If you still have problems then you have probably not configured CCS properly for the number of processors that you have in your system.

### **HINT!**

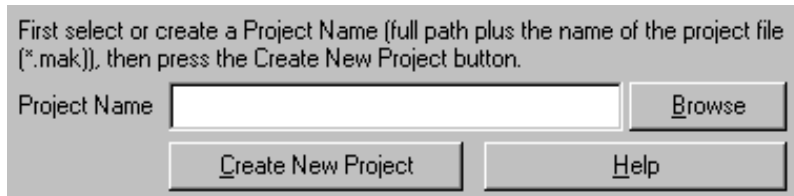
It is useful to have desktop icons for the “API board reset” and “API JTAG reset” programs. These are installed by the HUNT ENGINEERING software installation. These programs very briefly print a message that you do not have time to read. If you need to read them use the HUNT ENGINEERING → Confidence Checks → tools to perform the same action. Then the message remains on the screen.

### **FAQ**

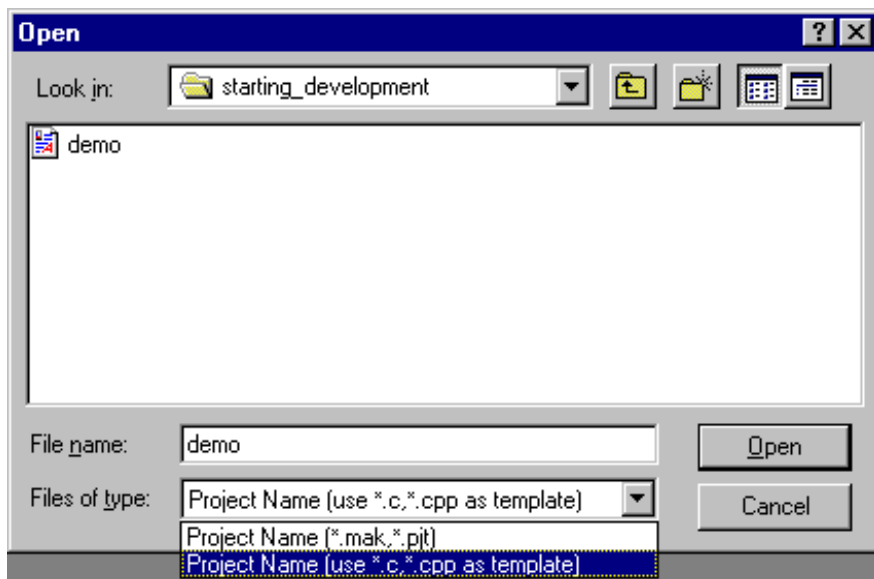
If you receive a message stating “Hardware not supported” then it is because you do not have the complete version of Code Composer Studio installed. You have the CCS-Compile Tools version that is supplied with the TI EVM board.

## Starting an example on a single HERON processor

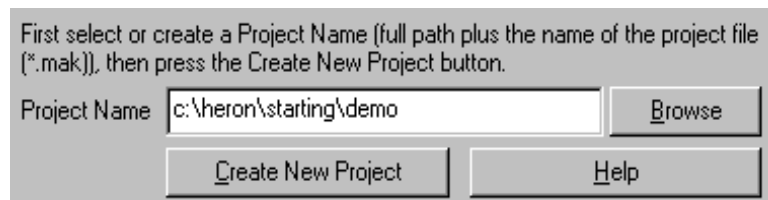
The example that we supply for this tutorial is a C file called demo.c. On the Hunt Engineering CD this is in the directory \software\examples\starting\_development. When you start with the demo example, simply copy the source file from the CD into a new directory. You can now use one of the HUNT ENGINEERING Code Composer Studio Plug-ins to configure a project for you. To do this, start Code Composer and choose Tools→HUNT ENGINEERING→Create new Heron-API project. This will guide you through setting up the project.



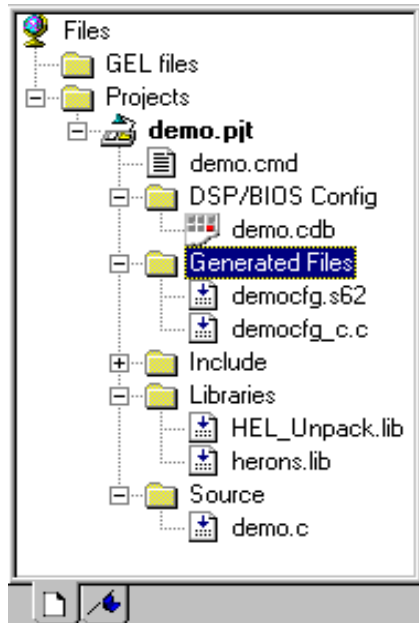
As long as you choose the name “demo” for the project, when creating the project, the plug-in will incorporate the “demo.c” file. One way you can do this, is to press the “Browse” button of the plug-in. Browse to the directory in which you copied the getting started example. Change “Files of type” to “Project Name (use \*.c, \*.cpp as template)”.



Then click “Open”. The plug-in will use the C file name to generate a project file name. Now you are ensured that your project name is called “demo.prj” (CCS 2.x) or “demo.mak” (CCS 1.2). Thus, when you actually create the project, “demo.c” will automatically be added to the project.

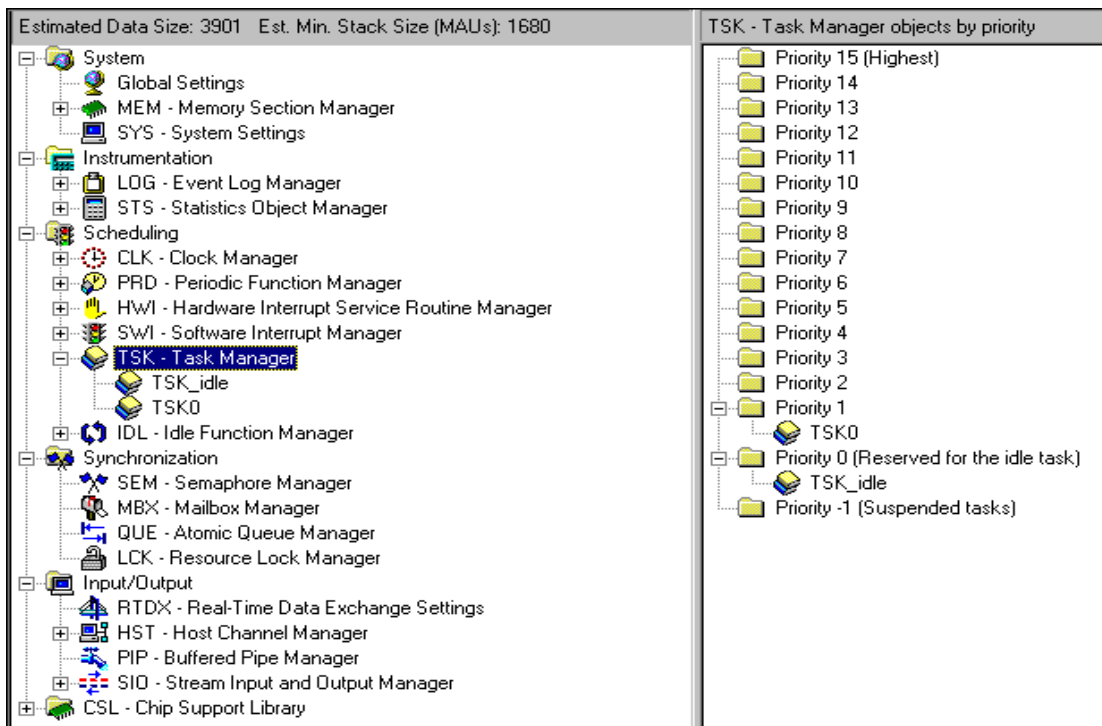


Create the project by clicking on the “Create New Project” button. The plug-in will ask you if you want to create the project for the Server/Loader. Answer “No” to this question. You’ll end up with a

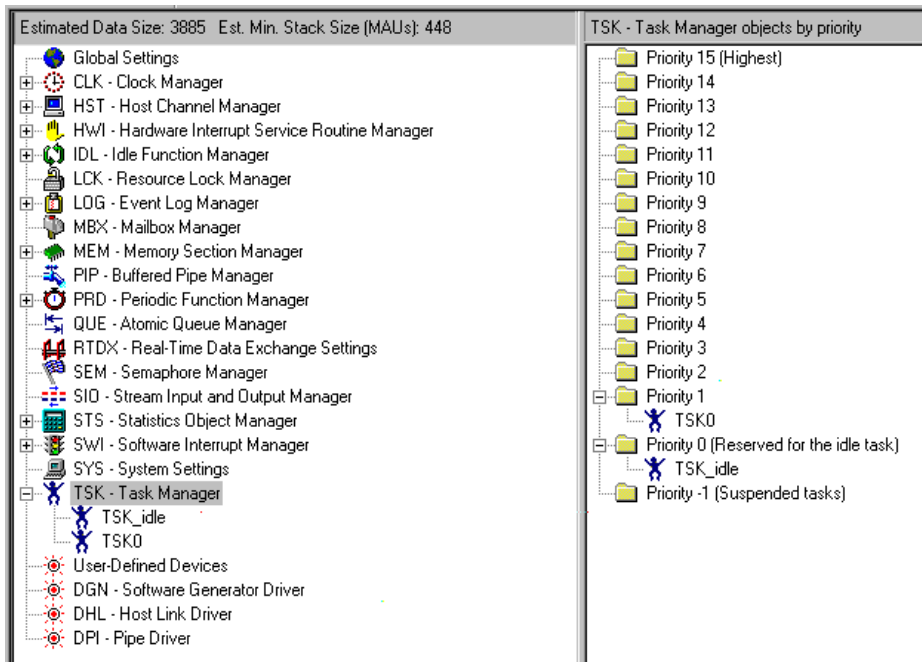


project something like in the picture below:

Then all you need to do is to open the .cdb file and insert the SWI’s and tasks required by your program to your configuration. You open the cdb file by going to the project manager, opening “Projects”, then “demo.pjt” (CCS 2.x) or “demo.mak” (CCS 1.2), then “DSP/BIOS config”. Finally, double click on “demo.cdb”. With CCS 2.x you’ll see something like:

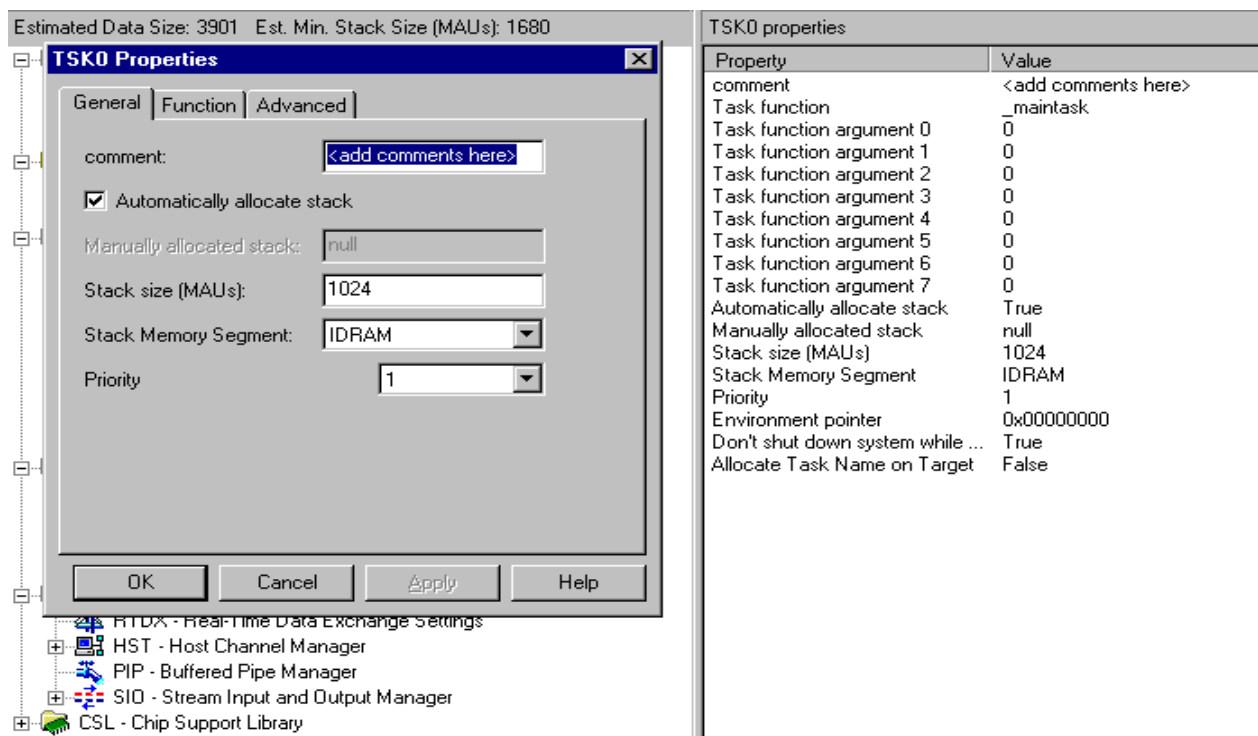


You may have to open several items to get to the same screen as shown above. With CCS 1.2 the screen looks a bit different, as shown on the next page.



## Adding Tasks

In this “simple” example we have one Task which is the core of our program. This Task is called “maintask”. The create new project plug-in has already added this Task for us. You can check this by selecting TSK – Task Manager -> TSK0. The Task function field should show “\_maintask”. This task is always added by the plug-in (even when there’s no “maintask” in your C code). This is so that most of our examples can be built immediately after creating a new project. (Most examples, but not this example, we need to add some SWI’s later). For CCS 2.x, TSK0 would look like:



## Adding SWIs

In this “simple” example we will add 2 SWIs. Do this by right clicking on the SWI manager (“SWI

– Software Interrupt Manager”) and choosing insert SWI, twice.

Now these SWIs must each be assigned a function to run. Right click on the individual SWI entries and choose properties. In the box that pops up, there is a dialog called “function”. Here you must enter your function name. If this is a C function it must be preceded by an underbar ( \_ ); an assembly function does not require this underbar.

Each function assigned to a SWI can be assigned a mailbox value and up to 2 arguments. This means a single function can be called by multiple SWIs and different arguments passed to them. Our example uses this so set the function for both of the remaining SWIs to `_procswi` but for SWI0 choose an `arg0` of 1, and for SWI1 choose an `arg0` of 2.

It is also possible to rename your SWIs to have more meaningful names for your project, but for the demo we will not bother.

You can now set the priorities of your SWIs by maximising the configuration tool window, and left clicking on the SWI manager. In the pane to the right you should see your SWIs listed along with some others that are used by HERON-API. By dragging down the SWI icons in that pane the tool will allow you to change the priority for each SWI.

If you look at the HWIs you can see that some of them have been assigned to HERON-API functions as they are used to manage the HERON FIFO interrupts of your module and the DMA engines.

If you look at the Global Settings properties, by right clicking on the “global settings” and choosing properties, you will see some settings for the processor. These should be correct for your module, unless you are clocking at a slower rate. Notice that the Program Cache Control setting is cache enable. This is a good general purpose choice allowing your code to be placed into the much larger available external memory of your module. If you did this without enabling the cache your program performance would be poor, but enabling the cache will give you the advantage of performance while using the external memory for your code storage.

If you look at the memory section manager you will see the various memory blocks of your module, and will see how various parts of your project are being assigned to different memory areas. The settings in the supplied `cdb` files are a good general purpose starting point but as your project progresses you may need to change some of these.

One thing that you MUST NOT do, is place any code section into IPRAM while the cache is set to be enabled. When the cache is enabled, the IPRAM section is used as cache and any code placed there will be overwritten. This also means that the “function stub” and “interrupt service table” memory of the HWI manager must not be placed there either.

Save your `cdb` file in your project directory by using File → Save.

### **Setting up the Project Manually**

For your information (or if there is some problem) here is how to set up the project yourself:-

In Code Composer, select ‘Project →new’ and choose the path and name for your project. In this case choose a name of “demo”.

Now you need to set up DSP/BIOS using the graphical editor. You will produce a `cdb` file, which contains the settings for DSP/BIOS in your project. As a starting point we provide “standard” `cdb` files for each HERON module type. These files contain the valid memory blocks for each module type, the settings that module needs for using HERON-API and some settings that are a good general purpose starting point for your development.

The HUNT ENGINEERING software installation copies the standard `cdb` files to your hard drive in

the directory %HEAPI\_DIR%\heron\_api\cmd. These need to be copied into the directory where the TI tools have been installed (normally c:\ti) and then \C6000\bios\include. Once these files are there you can proceed by choosing File → New → DSP/BIOS configuration. Choose the cdb file that shows the HERON module number that you have, and if there are several for that module number choose the one with the correct option for your module. For example if you have a HERON1 module with a C6701, you should select heron1\_c67...

This opens the cdb editor window.

In this “simple” example we have one Task which is the core of our program. We need to add a task to our configuration. Do this by right clicking on the Task manager and choosing insert Task.

Now this Task must be assigned a function to run. Right click on the entry TSK0 and choose properties. In the box that pops up, there is a dialog or tab called “function”. Here you must enter your function name (at “Task Function”). If this is a C function it must be preceded by an underbar ( \_ ), an assembly function does not require this underbar. For our example set TSK0 to be \_maintask which is our task function.

In our “simple” example we will add 2 SWIs. Do this by right clicking on the SWI manager and choosing insert SWI, twice.

Now these SWIs must each be assigned a function to run. Right click on the individual SWI entries and choose properties. In the box that pops up, there is a dialog called “function”. Here you must enter your function name. If this is a C function it must be preceded by an underbar ( \_ ) an assembly function does not require this underbar.

Each function assigned to a SWI can be assigned a mailbox value and up to 2 arguments. This means a single function can be called by multiple SWIs and different arguments passed to them. Our example uses this so set the function for both of the remaining SWIs to \_procswi but for SWI0 choose an arg0 of 1, and for SWI1 choose an arg0 of 2.

It is also possible to rename your SWIs to have more meaningful names for your project, but for the demo we will not bother.

You can now set the priorities of your SWIs by maximising the configuration tool window, and left clicking on the SWI manager. In the pane to the right you should see your SWIs listed along with some others that are used by HERON-API. By dragging down the SWI icons in that pane the tool will allow you to change the priority for each SWI.

Save your cdb file in your project directory by using File → Save as and choosing your project directory. The cdb file MUST have the same name as the .out file you are generating, in this case demo, so choose demo.cdb.

Saving your cdb file will automatically generate some other files for your project. One of these is a cmd file. This .cmd file gives the linker the correct information about the items configured in the cdb file, but unfortunately not about those other linker commands needed in your project.

Now you must also copy a .cmd file from %HEAPI\_DIR%\heron\_api\cmd to your project directory. Again choose the one that refers to your module number. The ones that have “slbios” are for Server/Loader applications – do not choose that one now.

When you have copied the correct .cmd file to your project, you must edit it to include the “other” cmd file generated by saving the .cdb file. To do this replace the \*\*\*\* in the file with the name of your .cdb file. Look in this file while you are editing it, and notice that two sections are being placed by that file. These are sections used by the HERON-API. Later you might need to add other sections there that are used in your own project.

Use “Project → add files to project” to add the source file demo.c to the project.

Use “Project → add files to project” to add the linker command file that you just edited.

Use “Project → add files to project” to add the cdb file to the project.

Use “Project → add files to project” to add the library file %HEAPI\_DIR%\heron\_api\lib\herons.lib to the project. (This is a general purpose library built for the small memory model. There are other choices that sometimes become necessary as you progress with your project, but if at all possible you should use this small memory model. See the HERON-API manual for more details of the other choices.)

Now use “Project → Options” to add the path %HEAPI\_DIR%\heron\_api\inc to the include search path. In fact you have the choice of entering the complete path name c:\heapi etc or you can reference the environmental variable %HEAPI\_DIR% by entering %HEAPI\_DIR%\heron\_api\inc.

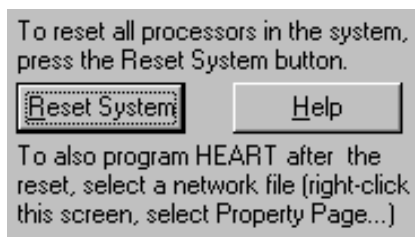
### Building and Running the Example

The project can now be built using “project → build” option. The project should build with 0 Errors and 0 Warnings.

The program can be loaded using “File → Load program” and choosing demo.out. With CCS 2.x this out file may be located in the “Debud” sub-directory. Then you can use “Debug → Go Main” to run the processor to the end of its initialisation code, i.e to the “main” of your program.

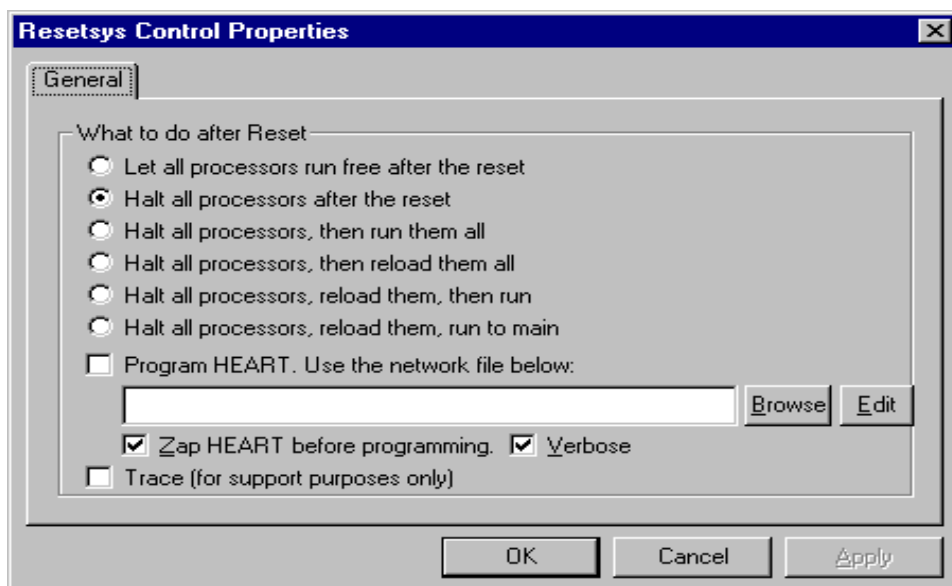
The program can now be stepped or run as required.

Try out the HUNT ENGINEERING CCS “System Reset” Plug-in by choosing Tools→HUNT ENGINEERING→Reset System.



When you use the “Reset” button it will actually perform a hardware reset of the system – ensuring that all FIFOs are reset, and that the Config signal is re-asserted. The Processor modules will re-boot from their FLASH ROM and hence will re-initialise their EMIFs. These are important steps to recover from a crash when debugging your code.

By right clicking on the background of the plug-in you can select from a menu of operations. This allows you to re-load your processors and run or go to main etc. Choose whichever you prefer. NOTE this cannot make the first load of your processors, as it uses re-load.



NOTE also that this plug in resets the “system” so will reload and run ALL processors in the system.

Users of HEART based carrier boards (such as the HEPC9) should take notice of the “Program HEART...” field in the Properties page. After a reset, all HEART connections disappear. This field will tell the reset plug-in to run HeartConf (the HUNT ENGINEERING HEART configuration utility) after the system reset.

HeartConf uses a network file to know what FIFO connections to make. The network file is a ‘pure’ ASCII file. In Windows, Notepad or Wordpad can be used. With Linux a simple editor such as ‘vi’ can be used. Editors used to create software code (such as Microsoft Visual C/C++) also work fine. To learn about the syntax of the network file, please have a look in the Server/Loader documentation. HeartConf is a derivative of the Server/Loader, using precisely the same syntax. The only difference is that HeartConf won’t actually boot DSP processors and FPGAs, even though in the syntax HeartConf will still insist on ‘something’ being on that spot.

You may already have spotted the “Edit” button. This allows you to quickly view the network file. But also, if no network file was selected yet, it will create a template network file.

Let’s have a quick look at the contents of a network file. First, define the board you’re using. For a HEPC9 with the red board switch set to 0, use:

```
BD API HEP9A 0 0
```

Where the first 0 is the board number, and the second 0 the fifo number (FIFO A is 0, FIFO B is 1, etc). Next, define the modules in your system and give them a nice, descriptive name. Note that the interface to the host also exists; this is to allow you to create FIFO connections to the host interface.

```
c6 0      heron  ROOT      (0)  00000001  filename
fpga 0     fpga1  normal                    2  rbtfile
gdio 0     gd12  normal                    00000003
ibc  0     ibc1  normal                    0x06
pcif 0     host1 normal                    0x05
```

The first field defines the module type. The ‘c6’ can be used for HERON1, HERON4, and other HERON processor types. The ‘fpga’ can be used for FPGA and HERON-IO modules. The ‘gdio’ can be used for any HEGDxx module. The ‘ibc’ stands for Inter Board Connector. The ‘pcif’ stands for PC InterFace. Note that the filenames are arbitrary. Although they must be there, the actual values are not used – by HeartConf. If you also use the same network file with the Server/Loader, then it’s probably useful to have a proper filename – which is then used by the Server/Loader.

Finally, it’s time to connect the different modules up. For example, to connect the C6x to the host:

```
heart      host1  0      heron  0      1
heart      heron  0      host1  0      1
```

A ‘heart’ means: make a one way FIFO connection. Next, specify the ‘from’ module (here “host1”) and ‘from’ FIFO (here “0”), then the ‘to’ module (here “heron”) and the ‘to’ FIFO (here “0”). These two connections create a duplex FIFO connection between the host and the C6x. The C6x can reach the host by reading/writing FIFO 0. The host can reach the C6x by reading/writing its FIFO 0.

As another example, let’s create a connection between the C6x (fifo 1) and the fpga (fifo 2).

```
heart      fpga1  2      heron  1      t=0
```

This creates a simplex connection from the FPGA module to the C6x. The FPGA can reach the C6x by writing to its FIFO 2. The C6x can read what the FPGA wrote by reading its FIFO 1. As a last example, let the FPGA read the GD output using FPGA FIFO 2:

```
heart      gd12   3      fpga1  3      3
```

As a last remark, the last field of a ‘heart’ statement is the number of timeslots. You can choose between 1, 2, 3, 4, 5, and 6. HeartConf will automatically allocate an actual timeslot. If you want to allocate a timeslot yourself, you can use the “t=” or “v=” specifiers – as shown above. For more details on how this works, please refer to the Server/Loader manual.

After all this, sorry, but we don’t need to use a network file for this demo. It only uses 1 processor but no FIFO’s. However, other examples do use FIFO’s – for example the GDIO examples, the multiple processor examples, and the host examples. With those examples you may very well want to use this functionality. You can still try it here, if you wish. Creating FIFO connections won’t alter the way this example works. Simple click – enable the “Program HEART ...” field, and choose or create a network file. Close the Properties window and do the reset.

## **Description of the demo.**

The file demo.c makes some includes, which are necessary for the use of DSP/BIOS. The std.h must always be there, and for each DSP/BIOS component you use you must include a header for it e.g. swi.h for using SWIs.

There is also a heronx.h includes, where x represents the HERON module type that you have. This is necessary to be able to use HERON\_API functions.

Now there are some extern definitions of the SWI objects. This is so that they can be “posted” from programs in this file.

The main program contains no code.

Then there are two functions, maintask that simply posts the first SWI, and procswi that determines the SWI that called it, posts the “other” SWI that calls procswi and does some work with counter.

Running the program shows nothing, but if after your “go main” you set a breakpoint in the beginning of maintask you can run to that. Also set a breakpoint in procswi, and notice that stepping over the SWI\_post does result in you reaching procswi. This is because procswi is a higher priority than the task.

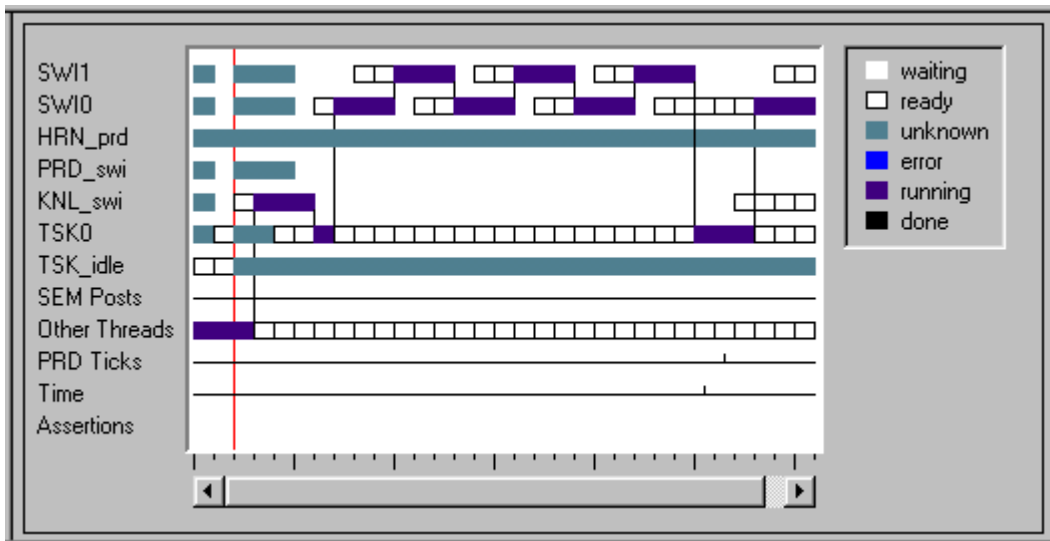
## **RTDX**

Built into DSP/BIOS is something called Real Time Data Exchange. If you stop your program and select “Tools → DSP/BIOS → RTA control panel” you will see a panel of options that can be enabled. Right click on the background of that panel, and choose “enable all”. Now select “Tools → DSP/BIOS → Execution Graph”. Another panel will open. If you run your program again, and then stop it, this panel will display a graph. This graph shows your task and SWIs, and you should see the dark blue “execution” passing continually from SWI0 to SWI1. The graph is quite short, because the system log length is set that way. This log is a buffer in the memory of your DSP where this information is being stored in real time. It only gets transferred to the host machine when the DSP has nothing to execute. In the case of our demo this never happens.

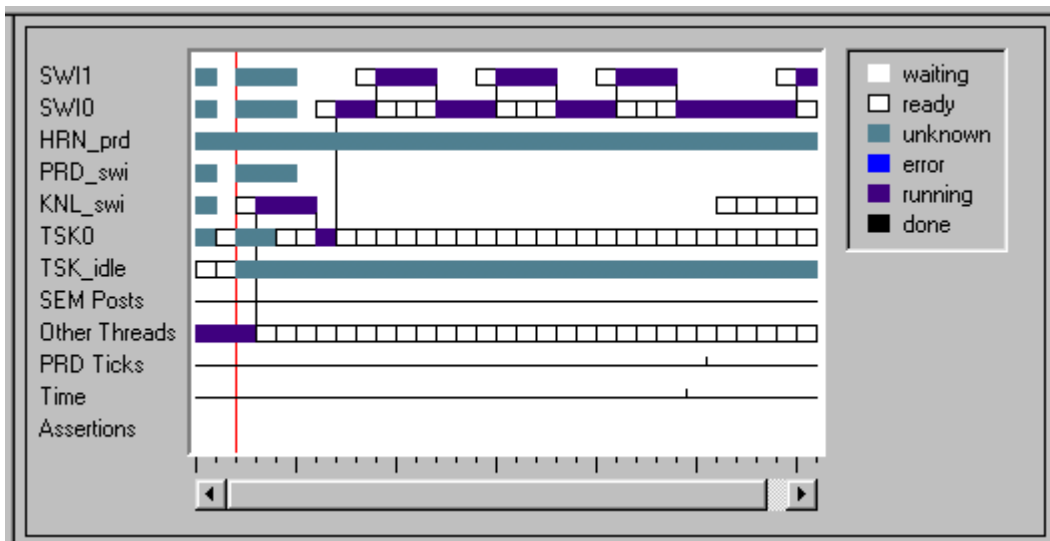
Change the length of the system log by opening the cdb file. You can do this by “opening” the project folders (click on the + sign) until you get to the cdb file. Then double click on the cdb file. Double click on the Event log manager in the cdb file. Then right click on LOG\_system and select properties. You will see that the buffer is placed into internal data RAM, and has a length of 32 words. Change this to be 128 words. Notice that the logtype is circular, i.e. it will be overwritten if there has not been an opportunity to send it to the host.

Select Apply and OK. The project must now be rebuilt with these new settings, and the program re-loaded. Running it now shows a lot more of the execution history. Sometimes you might even see the PRD0 and prd\_swi get run. Unless you were lucky, you will not see the one time that TSK0 was run. If you change the log type of the LOG\_system to be “fixed”, rebuild and re-run the system, you will find that after the LOG\_system buffer is filled once, it is not overwritten. If you move to the left of the graph

you will see the program starting. You will see that TSK0 is running, when SWI1 gets posted and it is immediately run. When SWI1 gets posted it is ready to run, but does not run yet. This can be seen by the white “ready” boxes.



If you change the priority of SWI1 to be priority 2, leaving SWI0 as priority 1, rebuild and re-run the graph changes to be:-



Notice that now, as soon as SWI0 posts SWI1 its execution starts. This is because it is now a higher priority. This clearly shows the use of priorities.

The RTDX data in our example so far has only been updated when we stop the program running. This is because we always have a thread ready to execute. This is one of the features of RTDX, in that it does not consume processor cycles if your application wants them – i.e. it is non-intrusive debugging.

If your program actually runs like this, but you want to update the RTDX information, you can force the processor to run through the functions that send the data to the host by adding a call to the function IDL\_run() to your loop. You can try this in the example.

Normally, however, a real time DSP program will spend some time in the idle thread of the program, and the data will be sent then without impacting your program's performance. In this case the execution of threads will be triggered by receiving data or a timer function.

There are many other features of DSP/BIOS and RTDX which can be learnt from the TI tutorials and documentation.

While you have not seen the use of HERON-API to access the FIFOS, this demo in fact does use HERON-API to control the config line and digital outputs. You have now successfully built and run an application that uses DSP/BIOS and HERON-API, and can proceed to learn about the use of the FIFO access functions and DMA management from other examples on this CD. Typically you need to choose an example that uses the I/O module that you have, or the multi-processor example.