



**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](mailto:sales@hunteng.demon.co.uk)  
URL: <http://www.hunteng.co.uk>



## The “testio” HERON\_API example.

Rev 2.0 P.Warnes 17-7-00 (changed to reflect CCS 1.2 and additional HUNT CCS plug ins)

The “testio” example is not really a program but more of a code segment to demonstrate how the HeronTestIo function can be used. It has been taken from a real system example provided to a customer to demonstrate the use of their system, and was functional in that case.

The DSP code uses HERON-API to manage the transfer of data over the HERON FIFOS.

The use of HERON-API means that the example is easily changed to use any HERON C6000 module. HERON-API uses DSP/BIOS internally so must be built using Code Composer Studio.

This document describes how to functions are used to overlap input and output of data.

### History

Example revision 2.0 made for HERON-API V2.3

Example revision 3.0 made for CCS V1.2

## **Example software**

The example that we supply is a C file called example.c. It needs to be changed to reflect your actual needs, and then built using Code Composer Studio and uses the HERON-API software that has been installed on your PC when you did the “install drivers and tools” from your CD.

## **DSP/BIOS**

DSP/BIOS is the multi-threading environment provided as part of the Code Composer development Environment. It also provided 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 be able to use it.

This example is configured and built using Code Composer and DSP/BIOS.

## **HERON\_API**

HERON\_API is the hardware independence layer that we provide to access HERON FIFOs and other features of the HERON modules. It allows the DMA engines of the processor to be used when transferring to and from the FIFOs without knowledge of the FIFO hardware, or the DMA engines.

## **HeronTestIo vs HeronWaitIo**

The I/O model used by HERON-API is an asynchronous one. That is the HeronRead and HeronWrite functions actually *start* the I/O, but it is left to the user to determine when the I/O is complete. The function HeronWaitIo and HeronTestIo functions are provided to facilitate this.

HeronWaitIo is a blocking function that will not return until the response is that the I/O has completed. While this is useful in some cases, it is more likely in a real-time system that the program merely wishes to “poll” for status, and will execute more work if the I/O is not yet complete.

## **Setting up the example**

The example is a HERON\_API project that can be set up using the create project plug in. Choose Tools→HUNT ENGINEERING→Create new Heron-API project. This will guide you through setting up the project and as long as you choose the name “example” for the project it will incorporate the example.c file. Then all you need to do is to open the .cdb file and insert the TSK0 and set it to be \_maintask.

## **The example**

The example program is taken from a real system, where an I/O board provided data continuously, and the data should be passed to a host computer for storage. Later some processing will be made by the DSP before the data is stored.

The problem is that the I/O board data cannot be stopped, and must be accepted at all times. The data stream to the host however can be disrupted for a number of reasons. The non real-time operating system may stop servicing our data while another task is being undertaken, and in fact disk caches often stop accepting data while the data is actually written to the media.

So we must have a method to store up data buffers on the DSP, which can be sent to the host when the disruption finishes.

The example manages some data buffers in a circular manner, and uses HeronTestIo on the input data to immediately start the next input. Then some buffer management is performed , and processing could be inserted here.

A second HeronTestIo is then performed on the output data stream, and if this succeeds we check to see if a new buffer is ready to be sent. If there is one ready we start the next output. If there is not a buffer ready to send we simply return to check the status of the input. In this case the call to HeronTestIo will again return its OK status the next time around the loop.