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

The "SIO" HERON-API example

Rev 1.0 R.Williams 1-9-00

The SIO example is not really a program but more of a code segment to demonstrate how to use the Streaming I/O (SIO) functions of the HERON-API. The Streaming I/O model of the HERON-API is provided for high performance FIFO access when reading or writing data to or from a high speed I/O module.

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 use the SIO functions of the HERON-API to achieve high performance FIFO access.

History

Example revision 1.0 made for HERON-API V3.0

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. It 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 Studio 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 be able to use it.

This example is configured and built using Code Composer Studio 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.

HeronSioRead and HeronSioWrite vs HeronRead and HeronWrite

The FIFO access models provided by the HERON-API perform asynchronous I/O. That is, in the case of the TestIo and WaitIo models, a FIFO transfer is started by a call to HeronRead or HeronWrite. As soon as the transfer has been started the function call completes and the transfer continues in the background.

The user must then call HeronTestIo or HeronWaitIo to detect the completion of the transfer. In the case of HeronWaitIo, the function will block until the transfer has completed. In the case of HeronTestIo, the function will return immediately, indicating whether the transfer has completed or whether the transfer is still in progress.

The Streaming I/O (SIO) FIFO access model of the HERON-API provides higher performance than the standard model, by automatically restarting the next read or write operation as soon as the previous transfer has completed. Instead of the user application starting a transfer with a specific data buffer and then performing a second function call to detect completion, the SIO model provides queues to manage the flow of data. These queues allow the automatic transfer of data in the background.

The HERON-API provides three SIO drivers (which are located in the User Defined Devices section of the CDB file). There are two SIO drivers for reading from a FIFO and one SIO driver for writing to a FIFO. An SIO driver is used to maintain the buffer queues, ensuring that data is kept flowing as efficiently and quickly as possible.

For a FIFO opened to use SIO, two queues are created. One queue is used to pass buffers from the user application to the SIO driver (the out-queue), and one queue is used to pass buffers from the SIO driver back to the user application (the in-queue).

For an application that is reading data from a FIFO, the SIO transfer is first started and data transfer begins. As each buffer is filled by the HERON-API it is placed in the in-queue. When the user application is ready it can take a filled buffer from the in-queue, and at the same time return an empty buffer to the out-queue. In this way, the HERON-API can continue to read from the FIFO filling buffers and placing them in the in-queue, independently of the user task.

Similarly for an application that is writing data to a FIFO, the SIO transfer is first started and data transfer begins. For each full buffer in the out-queue the HERON-API will transfer the buffer data to the HERON output FIFO. As each buffer is written to the FIFO, it is replaced as an empty buffer in the in-queue. Again, this allows the user application to transfer buffers of data independently to the HERON-API.

The Read Trash Buffer

The HERON-API provides two SIO drivers for reading from a FIFO. One read driver uses a ‘trash’ buffer, and one does not.

When reading from a FIFO using SIO, the user application should keep up with the data transfer. That is, when a buffer is filled and placed in the in-queue, the out-queue should already contain an empty buffer for the next transfer.

When the out-queue is empty the ‘FIFO Read With Trash Buffer’ driver will continue to read data into the trash buffer until an empty buffer is placed in the out-queue. This driver offers the better performance of the two FIFO read drivers, but may discard one or more buffers of data when the user application does not keep up with the data input.

This driver may discard whole buffers at a time, but as such it guarantees that the data is always aligned on buffer-size boundaries.

For the ‘FIFO Read Without Trash Buffer’ driver, when the out-queue becomes empty, the driver will

wait for a new buffer, and will not write data into the trash buffer.

For all SIO drivers, the function SIO_ctrl can be called to find out how many times the trash buffer was used.

For the 'FIFO Read With Trash Buffer' driver, this function will return the number of times data was written into the trash buffer. For all other drivers this function will return 0.

Setting up the example

The example is a HERON-API project that can be set up using the 'Create new HERON-API project' plug-in. To do this, 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` source file.

When the project has been created, you need to open the `.cdb` file and insert two objects. The first object is a task object called '`TSK0`' with the task function set to be '`_mainTask`'. The second object is an SIO object named '`sio_stream`'.

Open the properties window for the new SIO object and select the device '`Dx0`' (where `x` is the number of the HERON module type you are using, for example for a HERON1 use device '`D10`'). Ensure that the mode is set to input, and that the Allocate Static Buffers check box has a tick in it. Enter a number of buffers to use, and enter a buffer size (in bytes) to match the buffer size used in the source file `example.c`.

The example

The example performs a simple loop that receives N buffers and processes them. The example is based around HeronSioRead, and as such is intended for reading data from a high speed I/O module.

The program could easily be converted to write data to a FIFO by using HeronSioWrite in place of HeronSioRead.

Streaming IO manages data flow by passing pointers to buffers rather than by copying each buffer of data. For each call to HeronSioRead, a pointer is passed into the function, and a new pointer is returned using the second function argument.

The pointer that is supplied to the function is a pointer to an empty buffer. An empty buffer is a buffer whose data is no longer required by the user application. This empty buffer must be returned to the HERON-API so that it can be used for a future transfer.

The pointer that is returned when the function call returns is a pointer to a buffer that has been filled by reading from the FIFO.

When the processing loop completes the program calls the function `SIO_ctrl`. This function call will interrogate the HERON-API SIO driver being used to find out how many times the trash buffer was used.

If you are using the 'FIFO Read With Trash Buffer' driver (Dx0), then this call will return the trash count into the integer pointed to by the third function argument. The number returned will be 0 if the trash buffer was never used, indicating that the processing loop kept up with the data transfer process. If the trash buffer was used, the function will return the number of whole buffers that were written to the trash buffer.