



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



TI Third Party Network
Member



Using the HEGD9 Examples for HERON Systems

Rev 3.3 P.Warnes & R.Williams 22-02-02 (included HEART based carriers)

The HEGD9 is a multi-channel A/D module, which multiplexes the data for all channels into a single data stream to be written to a FIFO.

This document describes two separate examples for the HEGD9. The examples are for use with HERON modules and show how to read the data using HERON-API and sort it using some assembly sorting routines supplied.

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.

For the purposes of each example the code is then loaded onto the processor and demonstrated using Code Composer Studio, but the same software can easily be used in an application that uses the Server/Loader. To find out how to do so, please reference the separate examples of writing Server/Loader applications.

History

Example revision 2.0 made for HERON-API V2.3

Example revision 2.1 tidied and moved DMA sorting to a separate example

Example revision 3.0 made for CCS V1.2

Example revision 3.1 added a second example using Streaming I/O (SIO)

Example revision 3.2 updated to use HEL_Unpack library for data unpacking

Example revision 3.3 added HEART based carriers (document change only)

Example Software

There are two examples supplied for the HEGD9. The first example uses HeronRead and HeronWaitIo to read the data from the HEGD9, and provides two methods for unpacking the data, one C source method and one linear assembly method. This example is located in the sub-directory 'demo' below the 'hegd9' example directory. The example is provided in a C source file called 'gd9_a.c'.

The second example uses the Streaming I/O (SIO) function HeronSioRead to read the data from the HEGD9. This example uses the linear assembly method of data unpacking. This example is located in the sub-directory 'realtime' below the 'hegd9' example directory. The example is provided in a C source file called 'gd9_b.c'.

Both examples show the use of linear assembly functions for unpacking the channel data. These functions are taken from the unpacking/packing library HEL_Unpack that is installed as part of your standard software install.

Both examples need to be built using Code Composer Studio and both need to use 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.

Starting

We assume that a user of this example has previously installed Code Composer and followed the confidence checks. They should also be familiar with using Code Composer.

Configuring the Examples

HUNT ENGINEERING provides several Code Composer Plug-in tools that allow you to make your development faster. The first plug-in is one that sets up Code Composer ready for your hardware, so you don't need to configure which device drivers are needed. This plug-in can be found in Start→Programs→HUNT ENGINEERING→AutoConfigure CCS.

These examples assume that Code Composer Studio has already been set up in this way. Note, the plug-in also copies the HERON-API template cdb files etc into the correct locations.

Configuring the First Example

When you start with the first HEGD9 example, simply copy the source files from the ‘demo’ directory on the CD into a new directory. Then start Code Composer and choose Tools→HUNT ENGINEERING→Create new HERON-API Project. This will guide you through setting the project up, and as long as you choose the name “gd9_a” for the project, it will incorporate the file gd9_a.c. Check the .cdb file to make certain there is a TSK object named TSK0 which has the function property set to be ‘_maintask’. This is part of the template .cdb file, and is how you will add tasks to your project as you continue from this example.

The default .cdb file will actually place all code into external memory, and switch on the program cache. This is a good general purpose setting, but might need to be changed for your application.

Building the First Example

You can now build the first example by choosing Project→Rebuild All. There should be no errors, but there will be one warning caused by the endless loop in maintask.

To run the example, you need to understand which FIFO the DSP will use to access the HEGD9.

HEART carrier boards (such as the HEPC9)

For an HEPC9, and other HEART based carrier boards, you need to create the FIFO connections. You can use the heartconf utility to make the connections how you want, or you can use the default routing jumpers to make connections. For full instructions on how to use the heartconf utility, or the jumpers, you should refer to the documentation for the HEPC9, but lets look at a simple example.

If you have a DSP module in Slot1, and the HEGD9 module in slot2, you can use a network file to connect whichever FIFO you choose, i.e.

```
# Using API
BD API HEP9A 0 0
# Nodes description
# ND BD_nb ND_NAME ND_Type CC-id HERON-ID filename(s)
#-----
pcif 0 host1 normal 0x05
c6 0 dspmod normal 0x01 none.out
gdio 0 HEGD9 normal 0x02
#-----
# from:slot fifo to:slot fifo timeslot
#-----
heart dspmod 2 HEGD9 0 1
heart HEGD9 0 dspmod 2 1
```

Would connect FIFO #2 of the DSP module to FIFO #0 of the HEGD9 module. You can use this network file from the heartconf entry under the programs→HUNT ENGINEERING group. But as you will be using Code Composer Studio to build and load the DSP example code it is probably better to use the settings on the Reset plug in to configure the network for you after each reset. (Because the FIFO connections get cleared by the reset.)

Alternatively, you could also use the default routing jumpers. Using the same module set up (C6 in slot 1, HEGD9 in slot 2) fit the input and output jumpers to 0 on both modules. This creates a connection between FIFO #0 of the DSP module and FIFO #0 of the HEGD9 module in both directions. (But note that the connection is created upon system reset.)

HEPC8

For an HEPC8 the connections are determined by which slots your modules are fitted to. There is probably a document shipped with your system that explains how your system is configured, but if you have lost that or changed your configuration you will need to refer to the user manual for the module carrier you are using to work it out.

For example if the DSP is in slot 1 of an HEPC8, and the HEGD9 is in slot 2, the DSP will use FIFO

#2, and the HEGD9 will use FIFO #

Running the First Example

After building, the program can be loaded and started by choosing Debug→Go Main. To observe the example running you should set a breakpoint in the maintask and run to it. You should be able to animate the program and it should continually reach this breakpoint.

If you wish to see data use the View→Graph option to display one of the result buffers `chx_data` as unsigned integers. Set the DC value to be `0x7ff` and the maximum to be `0xffff`, and make sure the display length is the same as the buffer in the example.

Setting up the First Project Manually

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

Make sure that you have copied all of the `.cdb` files from the directory `%HEAPI_DIR%\heron_api\cmd` into the directory `C6000\bios\include` under the directory where your Code Composer Studio installation is (usually `c:\ti`).

In Code Composer, select 'Project→New' and choose the path and name for your project. Remember this name as you must use the same name when you save the `.cdb` file.

Select 'File→New→DSP/BIOS Configuration' and choose the correct `.cdb` file for your hardware. This will have a name that uses your HERON module number and possibly an option that is available for that module.

Open the project `.cdb` file and right click on Global Settings, and check that the `CLKOUT` property is set to the frequency of your processor module. This is used by DSP/BIOS to calculate the correct settings for the timer period.

This `.cdb` file has some items set up which are for HERON-API. DO NOT CHANGE THESE!

For the first example you need to set up a task that is called `TSK0`. Under its properties set its function to be `"_maintask"`.

Use 'File→Save' to save the `cdb` file to the project directory. This must have the same name as the project name you have used.

Saving the `.cdb` file will generate a `.cmd` file, but that file will not place the sections `heronapi_code` and `heronapi_data`. For this reason there is a `.cmd` file supplied by us, in the directory `%HEAPI_DIR%\heron_api\cmd` that will be called by your heron module number and have `_bios.cmd` at the end, i.e. `heronx_bios.cmd`. You need to copy this to your project directory.

Now add the source file, the `.cdb`, and also the `heronx_bios.cmd` to the project. Edit this `cmd` file so that it includes the `cmd` file generated when you saved the `.cdb` file.

Add the HERON-API library `"herons.lib"` from the directory `%HEAPI_DIR%\heron_api\lib` to the project.

Go to Project Options and add `%HEAPI_DIR%\heron_api\inc` to the include path.

Add the Unpacking library `"HEL_Unpack.lib"` from the directory `%HEAPI_DIR%\heron_unpack` to the project.

Go to Project Options and add `%HEAPI_DIR%\heron_unpack` to the include path.

Add the compiler option `-mtw` and `-mi100` to the project. These set the correct optimisation and interruptability for the linear assembler file. Select `-o3` optimisation from the compiler optimisation menu.

The default `.cdb` file will actually place all code into external memory, and switch on the program cache. This is a good general purpose setting, but might need to be changed for your application.

Now build the example as in the section above – Building and Running the First Example.

Configuring the Second Example

When you start with the second HEGD9 example, simply copy the source files from the ‘realtime’ directory on the CD into a new directory. Then start Code Composer and choose Tools→HUNT ENGINEERING→Create new HERON-API Project. This will guide you through setting the project up, and as long as you choose the name “gd9_b” for the project, it will incorporate the file gd9_b.c. Check the .cdb file to make certain there is a TSK object named TSK0 which has the function property set to be ‘_maintask’. This is part of the template .cdb file, and is how you will add tasks to your project as you continue from this example.

The second example uses Streaming I/O. As such, you must also add a SIO object named ‘sio_stream’. Edit the properties of the SIO object as follows:

Set the Device property to ‘Dx0’ (where x is the number of the HERON module type you are using). This selects the ‘HERON-API FIFO read driver, with trash buffer’. Set the Mode to input. Set the Buffer size to 1024. This buffer size is in bytes, and must be 4 times the setting of the #define BUFFER_SIZE_WORDS used in the C source file. Set the Number of buffers to 4, and tick the box ‘Allocate Static Buffer(s)’.

The default .cdb file will actually place all code into external memory, and switch on the program cache. This is a good general purpose setting, but might need to be changed for your application.

Building the Second Example

The project should build without any changes. To run the example, you need to understand which FIFO the DSP will use to access the HEGD9.

To run the example, you need to understand which FIFO the DSP will use to access the HEGD9.

HEART carrier boards (such as the HEPC9)

For an HEPC9, and other HEART based carrier boards, you need to create the FIFO connections. You can use the heartconf utility to make the connections how you want, or you can use the default routing jumpers to make connections. For full instructions on how to use the heartconf utility, or the jumpers, you should refer to the documentation for the HEPC9, but lets look at a simple example.

If you have a DSP module in Slot1, and the HEGD9 module in slot2, you can use a network file to connect whichever FIFO you choose, i.e.

```
# Using API
BD API HEP9A 0 0
# Nodes description
# ND BD_nb ND_NAME ND_Type CC-id HERON-ID filename(s)
#-----
pcif 0 host1 normal 0x05
c6 0 dspmod normal 0x01 none.out
gdio 0 HEGD9 normal 0x02
#-----
# from:slot fifo to:slot fifo timeslot
#-----
heart dspmod 2 HEGD9 0 1
heart HEGD9 0 dspmod 2 1
```

Would connect FIFO #2 of the DSP module to FIFO #0 of the HEGD9 module. You can use this network file from the heartconf entry under the programs→HUNT ENGINEERING group. But as you will be using Code Composer Studio to build and load the DSP example code it is probably better to use the settings on the Reset plug in to configure the network for you after each reset. (Because the FIFO connections get cleared by the reset.)

Alternatively, you could also use the default routing jumpers. Using the same module set up (C6 in slot 1, HEGD9 in slot 2) fit the input and output jumpers to 0 on both modules. This creates a connection

between FIFO #0 of the DSP module and FIFO #0 of the HEGD9 module in both directions. (But note that the connection is created upon system reset.)

HEPC8

For an HEPC8 the connections are determined by which slots your modules are fitted to. There is probably a document shipped with your system that explains how your system is configured, but if you have lost that or changed your configuration you will need to refer to the user manual for the module carrier you are using to work it out.

For example if the DSP is in slot 1 of an HEPC8, and the HEGD9 is in slot 2, the DSP will use FIFO #2, and the HEGD9 will use FIFO #3.

You can now build the second example by choosing Project→Rebuild All. There should be no errors and no warnings.

Running the Second Example

After building, the program can be loaded and started by choosing Debug→Go Main. If you have a HEPC9, or other HEART based carrier board, you should now first run heartconf or do a system reset (using the reset plugin).

Set a breakpoint at the HeronSioClose function – this will be found if an error occurs.

Setting up the Second Project Manually

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

Make sure that you have copied all of the .cdb files from the directory %HEAPI_DIR%\heron_api\cmd into the directory C6000\bios\include under the directory where your Code Composer Studio installation is (usually c:\ti).

In Code Composer, select 'Project→New' and choose the path and name for your project. Remember this name as you must use the same name when you save the .cdb file.

Select 'File→New→DSP/BIOS Configuration' and choose the correct .cdb file for your hardware. This will have a name that uses your HERON module number and possibly an option that is available for that module.

Open the project .cdb file and right click on Global Settings, and check that the CLKOUT property is set to the frequency of your processor module. This is used by DSP/BIOS to calculate the correct settings for the timer period.

This .cdb file has some items set up which are for HERON-API. DO NOT CHANGE THESE!

For the second example you need to set up a task that is called TSK0. Under its properties set its function to be “_maintask”.

The second example uses Streaming I/O. As such, you must also add a SIO object named 'sio_stream'. Edit the properties of the SIO object as follows:

Set the Device property to 'Dx0' (where x is the number of the HERON module type you are using). This selects the 'HERON-API FIFO read driver, with trash buffer'. Set the Mode to input. Set the Buffer size to 1024. This buffer size is in bytes, and must match the setting of the #define BUFFER_SIZE_WORDS (when divided by 4) used in the C source file. Set the Number of buffers to 4, and tick the box 'Allocate Static Buffer(s)'.

Use 'File→Save' to save the cdb file to the project directory. This must have the same name as the project name you have used.

Saving the .cdb file will generate a .cmd file, but that file will not place the sections heronapi_code and heronapi_data. For this reason there is a .cmd file supplied by us, in the directory

%HEAPI_DIR%\heron_api\cmd that will be called by your heron module number and have _bios.cmd at the end, i.e. heronx_bios.cmd. You need to copy this to your project directory.

Now add the source file ,the .cdb, and also the heronx_bios.cmd to the project. Edit this cmd file so that it includes the cmd file generated when you saved the .cdb file.

Add the HERON-API library “herons.lib” from the directory %HEAPI_DIR%\heron_api\lib to the project.

Go to Project Options and add %HEAPI_DIR%\heron_api\inc to the include path.

Add the Unpacking library “HEL_Unpack.lib” from the directory %HEAPI_DIR%\heron_unpack to the project.

Go to Project Options and add %HEAPI_DIR%\heron_unpack to the include path.

Add the compiler option -mtw and -mi100 to the project. These set the correct optimisation and interruptability for the linear assembler file. Select -o3 optimisation from the compiler optimisation menu.

The default .cdb file will actually place all code into external memory, and switch on the program cache. This is a good general purpose setting, but might need to be changed for your application.

Now build the example as in the section above – Building and Running the Second Example.

Notes About the Example Programs

The first example actually makes a single “snapshot” of data. It does this by following a HeronRead immediately by a HeronWaitIo. Normally in a real time system you would overlap the acquisition of data with processing. Then you would be more likely to use HeronRead followed by your processing and finally a call to HeronWaitIo or perhaps HeronTestIo followed by some more processing. For examples of methods of how to use these techniques please refer to the HERON-API examples on the HUNT ENGINEERING CD.

Using the “snapshot” method of the first example, data is viewed from within Code Composer Studio using the View→Graph function. This method however loses data when the HERON FIFOs overflow. This is because in order to display the graph, Code Composer will first halt the processor and will then load the data for the graph over the JTAG serial bus.

While the processor is halted, the reading of data from the FIFO will also halt. The FIFO will quickly become full and begin to overflow, losing data from the HEGD9 as it does so. So when you use the animate feature with the first example, data is lost each time the processor is halted and the graph is updated.

For this reason the HeronRead in the first example includes a “flush” number of items. The flushed data items are used to remove stale data from the overflowed FIFO and ensures that the displayed data is a continuous buffer of data. In a real-time application, the FIFO should not overflow, and so the flush technique would not be needed.

The second example uses the Streaming I/O (SIO) functions of the HERON-API. By using HeronSioRead to read from the FIFO, data can be read at a faster rate than for the first example. Streaming I/O works by continuously transferring data to or from FIFOs in the background. As each buffer is transferred it is put in a queue to be used by the user application. This leaves the user application free to focus on processing the data while the streaming I/O driver continues data transfer as quickly and efficiently as possible in the background.

For a real-time application, you would not typically want to interrupt the reading and processing of data by halting the processor, as this causes data to overflow the FIFO, and presents the problem of flushing stale data from the FIFO as discussed above.

However, for situations where this occurs, the second example provides a function that is used to automatically resynchronise the data flow if the processing loop is halted. By calling this function after the processing loop has been temporarily stopped, the SIO read operation can be re-synchronised to ensure that each new buffer of data is correctly aligned such that channel 0 of the HEGD9 data is stored in the first element. For a further example of SIO, refer to the HERON-API SIO example on the HUNT ENGINEERING CD.

Both examples also need to unpack the data received from the HEGD9. The first example provides a choice of ‘method 1’ (C code) and ‘method 3’ (linear assembler). Method 1 is in C code and is therefore clearer to follow, and can be stepped through in Code Composer Studio. Really the first method is just by way of an explanation, and the linear assembly method would be better used in a real system.

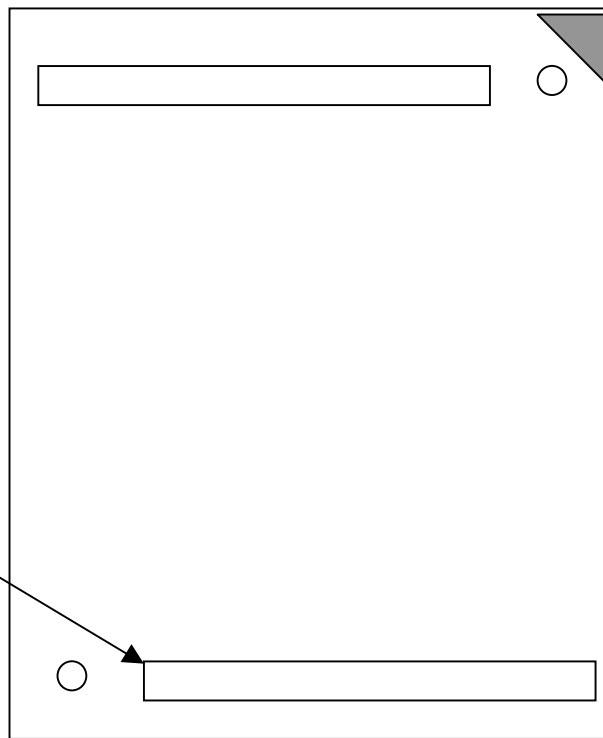
For the second example, only the linear assembly method is used.

DATA LOSS

It is possible for the system to lose data in two ways. The most common is that the FIFO is not being read as fast as it is being written by the A/D. In this case the FIFO becomes full and data will be lost. The only way to detect that type of loss is at the A/D module.

In fact the HEGD9 has an LED that lights if the FIFO becomes full, but as this will often happen for only a short time it can in fact result in just a “glimmer” from the LED. The only reliable way to detect this loss is to use an oscilloscope or logic analyser to check the FIFO full flag. This can be found on the HERON connector pins, and should be fitted with a longer pin to make it easier to probe.

**Corner pin is FIFO
full flag (low when
full)**



When using streaming I/O there is another way of losing data. This is because one of the SIO driver models uses a “trash” buffer. This means that if the DMA is able to read the data from the A/D fast enough, but the user application does not read the buffers from the driver fast enough, the DMA will continue to run but will discard whole buffers of data. In this case the FIFO flag will never be asserted, but the SIO driver will be discarding data. This case is not tested in the simple examples for the A/D, but is demonstrated in the “Streaming I/O example” for HERON-API.