



**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>



## Using the HEGD14 example for HERON systems.

Rev 3.0 P.Warnes 12-3-02 (changed to include HEART based carriers)

The HEGD14 is a multi-channel D/A module, which multiplexes the data for all channels into a single data stream taken from a FIFO.

This example is for the HERON modules and shows how to send the data using HERON-API and prepare 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 the 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 that please reference the separate examples of writing Server/Loader applications.

### History

Example revision 1.0 made for HERON-API V2.3

Example revision 2.0 made for CCS V1.2 and plug ins

Example revision 2.1 added use of HEL\_Unpack library

Example revision 2.2 added a second Streaming I/O example

Example revision 3.0 added support for HEART based carriers (documentation change only)

## **Example software**

The example that we supply is a C file called GD14.c. It needs to be built using Code Composer Studio and uses the HERON-API software, and packing/unpacking library that has been installed on your PC when you did the “install drivers and tools” from your CD.

There are two examples supplied for the HEGD14. The first example uses HeronWrite and HeronWaitIo to write the data to the HEGD14. Sample data is initialised to generate a sine wave on each of the output channels. It shows how to use a HEL\_Unpack library function to pack the data, ready to be sent to the HEGD14. This example is located in the sub-directory ‘demo’ below the ‘hegd14’ example directory. The example is provided in a C source file called ‘gd14\_a.c’.

The second example uses the Streaming I/O (SIO) function HeronSioWrite to send the data to the HEGD14. This example uses the linear assembly method of data packing. This example is located in the sub-directory ‘realtime’ below the ‘hegd14’ example directory. The example is provided in a C source file called ‘gd14\_b.c’.

Both examples show the use of linear assembly functions for packing 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 first example**

HUNT ENGINEERING provide several Code Composer Plug-in tools that allow you to make your development faster. The first one sets up Code Composer ready for your hardware, so you don’t need to configure device drivers etc and can be found from the Start→Programs→HUNT ENGINEERING→AutoConfigure CCS.

We assume that this is already set up, but this plug in also copies .cdb files etc into the correct locations.

When you start with the HEGD14 example, simply copy the source files from 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 “gd14\_a” for the project, will incorporate the gd14\_a.c file.

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.

You can now build the demo by choosing Project → re-build all. There should be no errors, but there will be one warning caused by the endless loop in maintask.

## 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 HEGD14.

## HEART based carriers e.g. 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 HEGD14 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 HEGD14 normal 0x02
#-----
# from:slot fifo to:slot fifo timeslot
#-----
heart dspmod 2 HEGD14 0 1
heart HEGD14 0 dspmod 2 1
```

Would connect FIFO #2 of the DSP module to FIFO #0 of the HEGD14 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, HEGD14 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 HEGD14 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 HEGD14 is in slot 2, the DSP will use FIFO #2, and the HEGD14 will use FIFO #3.

## Building the First 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 need to set the FIFO connections in your system as discussed above. You can do this by running heartconf after each reset. It is probably better to use the reset plugin for CCS, where you can choose to run HeartConf automatically after each reset.

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 what data is sent to the HEGD14, use the View→Graph option to display one of the channel buffers chx\_data as unsigned integers. Set the DC value to be 0x1fff and the maximum to be 0x3fff, and make sure the display length is the same as the buffer in the example.

If you wish to see the output data, you should connect an oscilloscope to the HEGD14 outputs.

## Manually Setting up the Project

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 Config' 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.

In the DSP/BIOS config tool, right click on Global properties, 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 this 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 files to the project and the .cdb, and also the heronx\_bios.cmd. 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. Add the HEL\_Unpack library "HEL\_Unpack.lib" from the directory %HEAPI\_DIR%\heron\_unpack to the project.

Go to Project Options and add %HEAPI\_DIR%\heron\_api\inc 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.

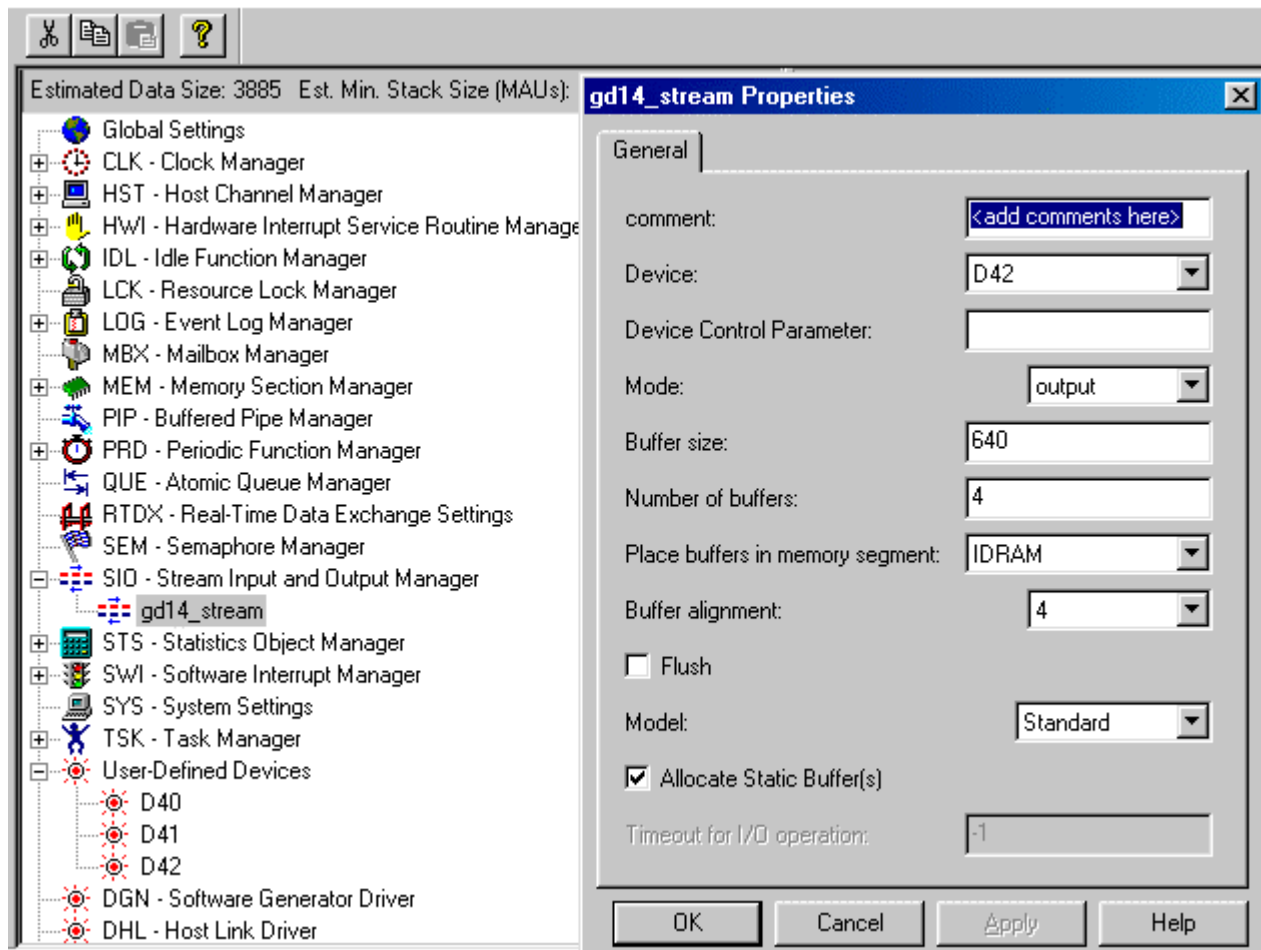
The default .cdb file will actually place all code into external memory, and switch on the program cache. This is a good general purpose setting, but might need to be changed for your actual application.

You can now build the demo by choosing Project → re-build all. There should be no errors, but there

will be one warning caused by the endless loop in maintask.

## Configuring the second example

When you start with the second HEGD14 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. As long as you choose the name "gd14\_b" for the project, it will incorporate the file gd14\_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 'gd14\_stream'. Edit the properties of the SIO object as follows:

Set the Device property to 'Dx2' (where x is the number of the HERON module type you are using). This selects the 'HERON-API FIFO write driver'. Set the Mode to output. Set the Buffer size to 640. 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 demo as-is uses 8 channels, 40 samples per channel (thus uses 8\*40 shorts, or 640 bytes). If you want to change the number of channels and/or samples per channel, please be aware that you also need to change the gd14\_stream's buffer size!

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

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

and no warnings.

To run the example, you need to understand which FIFO the DSP will use to access the HEGD14.

### HEART based carriers e.g. 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 HEGD14 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 HEGD14 normal 0x02
#-----
# from:slot fifo to:slot fifo timeslot
#-----
heart dspmod 2 HEGD14 0 1
heart HEGD14 0 dspmod 2 1
```

Would connect FIFO #2 of the DSP module to FIFO #0 of the HEGD14 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, HEGD14 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 HEGD14 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 HEGD14 is in slot 2, the DSP will use FIFO #2, and the HEGD14 will use FIFO #3.

### Building 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 need to set the FIFO connections in your system as discussed above. You can do this by running heartconf after each reset. It is probably better to use the reset plugin for CCS, where you can choose to run HeartConf automatically after each reset. Set a breakpoint at the HeronSioClose function – this will be found if an error occurs – and run (Debug→Run) to the breakpoint.

### 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 'gd14\_stream'. Edit the properties of the SIO object as follows:

Set the Device property to 'Dx2' (where x is the number of the HERON module type you are using). This selects the 'HERON-API FIFO write driver'. Set the Mode to output. Set the Buffer size to 640. 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)'.

The demo as-is uses 8 channels, 40 samples per channel (thus uses 8\*40 short integers, or 640 bytes). If you want to change the number of channels and/or samples per channel, please be aware that you also need to change the gd14\_stream's buffer size!

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 demo**

1. This first example actually spends its whole time waiting for data to be output. It does this by following a HeronWriteFifo immediately by a HeronWaitIo. Normally in a real time system you would overlap the output of data with the processing. Then you would be more likely to use HeronWriteFifo, followed by your processing and finally a HeronWaitIo or perhaps a HeronTestIo and then 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.

When the processor is halted, the writing of data to the FIFO will also halt. The FIFO will quickly become empty and begin to underflow, and the HEGD14 will run out of data to output. So when you halt the processor and/or view some variable or array, data output continuity is lost each time the processor is halted and the variable or array is updated.

2. The second example uses the Streaming I/O (SIO) functions of the HERON-API. By using HeronSioWrite to write to the FIFO, data can be written 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 or underflow the FIFO, and presents the problem of missing data in the FIFO as discussed above.

Both examples also need to pack the data to be sent to the HEGD14. Both the first example and the second example use a linear assembler function provided by Hunt Engineering's HEL\_Unpack library.

## **STALE DATA OUTPUT**

It is possible for the system to receive data too slowly relative to the sample clock. This happens if the FIFO is not being written to as fast as it is being read by the D/A. In this case the FIFO becomes empty and stale data will be output. The only way to detect this is at the D/A module. The GD14 has a Data LED that will light up if this situation occurs. At every underflow it will light up for a second.