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

The HERON_IO2 Version 2 example2

Rev 1.1 P. Warnes 29-04-05

The HERON-IO2 module is a module that has an FPGA and 2 channels of fast A/D along with 2 channels of fast D/A.

Most users will use the FPGA to provide either a custom I/O capability or a processing resource that uses the FPGA for that processing. In that case the Example2 project can form a starting point for the development of that.

Other users may want to simply use the HERON-IO2 as an I/O module in which case the example2 bit stream for HERON-IO2 is provided and the functionality of that bit stream is described in this document.

History

Example revision 1.0 10-05-01 first made for Version 2 of the HERON-IO2 Documentation change only

Example revision 1.1 29-04-05 Removed reference to specific ISE versions

What the bitstream does

Example2 (Transient analysis/Pattern generation) for the HERON-IO2 is supplied on the HUNT ENGINEERING CD, and web site. The FPGA source code is supplied along with bit streams that can be loaded directly onto the HERON-IO2. There are also example programs that you can use on the Host PC or a C6000 DSP module.

This example can therefore be used as it is, if it suits your needs, or can be used as a starting place for you to modify and make something that suits your needs.

If you make changes to the project and re-build it you can change the functionality to be whatever you want, but if you use the supplied bit stream you need to know what it is doing. This document describes that for you.

The standard clock soldered to “User Osc3” of the HERON-IO2 is 100Mhz.

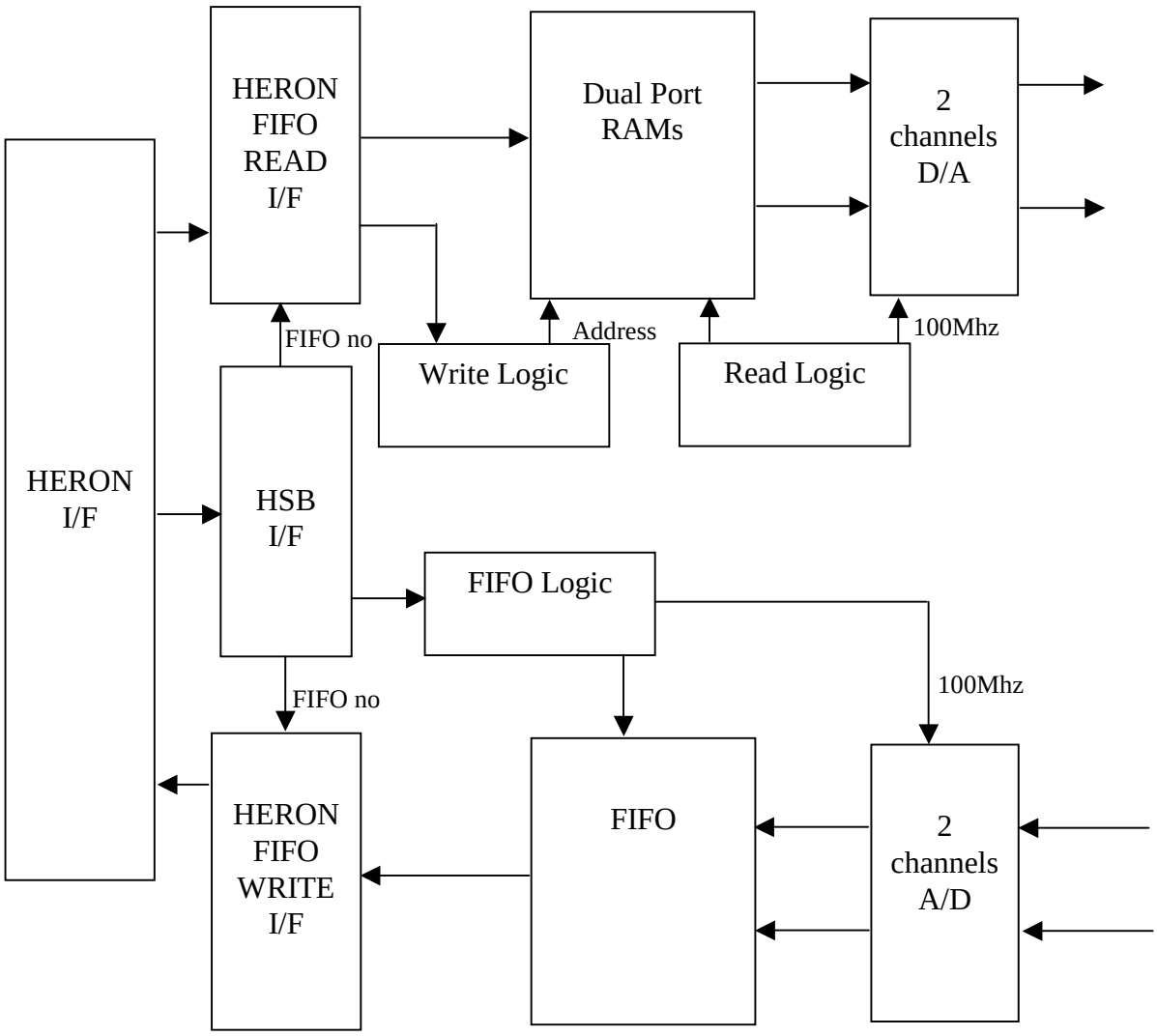
Example2 always uses this clock to drive the sampling of the A/Ds and D/As.

For an HEPC8 the HERON FIFO clocks must be driven at less than 60Mhz, whereas for the HEPC9 they must be driven at between 60Mhz and 100Mhz. For this reason there are different bit streams supplied for HEPC8 and HEPC9 as follows :-

2v1000fg456.rbt	HERON-IO2V fitted to HEART based carrier e.g HEPC9 (100Mhz FIFO clocks)
2v1000fg456_pc8.rbt	HERON-IO2V fitted to HEPC8 (50Mhz FIFO clocks)

When the bit stream is loaded onto the HERON-IO2, the FIFO connections can be selected by sending an HSB “user message”. The registers are address 0 = the fifo number that the FPGA reads from, and address 1 = the fifo number that the FPGA writes to. These are normally (but not necessarily) the same.

FUNCTIONAL BLOCK DIAGRAM



A/Ds

The use of the 100Mhz clock for the A/Ds means that 400Mbytes/sec of data is being produced from the A/Ds. A DSP or Host program cannot consume data at that rate, so it is useless to write that data directly to the HERON FIFOs.

Example 2 provides a FIFO inside the FPGA that has 511 locations. Into each of those locations a 32 bit word is stored that is formed as follows :-

Bits 31 &30	Undefined
Bit 29	Ch B THU – set if top half of range is used
Bit 28	CHB OTR – set if overrange
Bit 27..16	Ch B data – signed 12 bit
Bits 15 & 14	Undefined
Bit 13	Ch A THU – set if top half of range is used
Bit 12	Ch A OTR – set if overrange
Bits 11..0	Ch A data – signed 12 bit

Normally no data is placed into the FIFO and no data is written into the HERON FIFOs. When the FPGA receives an appropriate HSB message from the input FIFO it will capture a number of data items into the FIFO and immediately start to send that data to the HERON FIFOs.

The HSB accesses to trigger the capture are:-

User write to address 2 writes the lower 8 bits of the count

User write to address 3 writes the upper bit of the count – the action of writing this register triggers the capture.

The count is used to specify the number of samples to be captured from both channels, and will result in that "count" number of words to be sent to the FIFO. The maximum count is 511.

This makes Example2 an ideal candidate for "Transient Analysis" applications.

D/As

The use of the 100Mhz clock for the D/As means that 400Mbytes/sec of data is being consumed by the D/As. A DSP or Host program cannot consume data at that rate, so it is useless to take that data directly from the HERON FIFOs.

Example 2 provides 2 dual port RAMs inside the FPGA that have 512 locations each. Into each of those locations RAMs you can write a pattern, which will be continually cycled through and the values presented to the DACs.

The RAMs are written via the HERON FIFO, by sending 32 bit words as follows :-

Bits 31 &30	Don't care
Bit 19	Set to reset address pointer for ChB RAM
Bit 18	Set if data is for Ch B RAM
Bit 17	Set to reset address pointer for ChA RAM
Bit 16	Set if data is for Ch A RAM
Bit 15 & 14	Don't care
Bit 13-0	14 bit Data to be written to RAM.

On power up the ram is undefined, the first write to the RAM will be stored in the first address, and that address auto incremented ready for the next value. The two RAMs can be written and re written at

any time, but if you require the two DAC outputs to have a known phase it is recommended that the reset pointer bit is used on the first word of each new pattern. The RAMS are read using a common address so the DACs are both presented with the data from each address during the same clock.

This makes Example2 an ideal candidate for “Pattern Generation” applications.

This makes the normal use of the example bit stream as follows:-

- load bit stream for example2
- Send HSB message to configure the input and output FIFOs
- Send a pattern for the DACs over the HERON FIFOs,
- Send an HSB request for ADC data.
- Read that number of samples from the HERON FIFO
- Send the next ADC block request
- Receive the next block

Etc etc. Which is what the examples provided show.

Where are the bitstream and examples?

The bit streams for this example can be found on the HUNT ENGINEERING CD under \fpga\io2v2\Transient_analysis(ex2). The name of the .rbt file reflects the FPGA part number and the Carrier board type, as explained earlier in this document.

The DSP and host examples can be found in the dsp and host subdirectories.

An easier way to navigate to the correct directory is to select the “Files” link next to the “Transient Analysis/Pattern Generation” link under the IP sections of the CD browser.

The source files for the FPGA example can be found in the \src subdirectory. The sources in the \io2v2\common directory are also required. There is a project for ISE in the ‘ISE’ subdirectory.

What example do I use?

We have provided two pieces of software as examples to use with the example2 FPGA bit streams.

One runs on your Host PC, and requires a FIFO connection between the Host PC and the HERON-IO2.

The other runs on a HERON-C6000 module and requires a FIFO connection between the DSP module and the HERON-IO2.

If you do not have a DSP module then you will need to use the Host based example program, but if you have a DSP module then you can choose which example type you want to use. Choose whichever example is closest to your needs.

The following sections describe how to use the Host based and DSP based examples. Choose which one of these you want to follow and simply skip the other section.

Host based Example software

HEART carrier like HEPC9

If you have a HEART based module carrier like HEPC9, you can run the Host based example code with your HERON-IO2 module fitted to any slot. Then you need to configure the HEART connections between the module and the Host using HeartConf. This is done from inside the hegraph program for you using the network file that is in the same directory as the host example software.

HEPC8 carrier

If you have an HEPC8, then you must fit your HERON-IO2 module to the first HERON slot as this is the only slot that has a FIFO connection to the Host machine. In this case the FIFO number the module uses is FIFO #1.

Host example : What it does

We supply a pre-compiled Windows program for the PC, which can be used to communicate with the HERON-IO2, and to gather the data captured onto the PC where it is displayed.

The program is called 'hegraph.exe' and it is located in the 'host' sub-directory of this example. This example can be used both as a confidence check and a starting point for your development.

The 'hegraph.exe' program is supplied as an exe file, but we have included the Microsoft Visual C/C++ 6 project of this program as well. This allows you to change it and re-compile it as you want.

After you have loaded the correct bit stream into your HERON-IO2 module, the "DONE" LED should be switched off showing that the configuration was successful. Also The USER LED4 should flash about once per second, showing that the clocks are properly running in the FPGA. If the DONE LED is off, but the LED is not flashing it may be because the Delay Locked Loop used in the clock circuit of the FPGA needs to be reset. You can do this using the utility under "programs → HUNT ENGINEERING → API board RESET" if you want, but such a reset will also be made when you start the hegraph program.

When you run the hegraph program you will see a window appear, that has some control menus available. Using these menus you will be able to start the program (which will send HSB messages to correctly set the FIFO numbers that the FPGA should use, and also configure HEART if you have a HEART based module carrier). It will also capture and display the data coming from the A/Ds, and to set patterns to be output on the DAC outputs. Optionally you can also display an FFT of the data coming from the A/Ds (which is calculated on the PC).

Host example : Using it

First the correct example2 bit-stream for your module should be programmed into the FPGA using the standard Windows FPGA programmer found under "Programs → HUNT ENGINEERNG → Program HERON FPGA". What bit-stream to choose is discussed in the above sections and depends on the type of HERON-IO2 you have and the type of module carrier you have.

If you don't know how to load a bit-stream onto the HERON-IO2, please review example1 once more. Verify that after the loading process the "Done" LED goes off. Finally, execute the 'hegraph.exe' program.

If you are using an HEPC9, you must first select the slot number of the module you want to use with the example program. Do this by choosing "Options → Module slot number". If you have an HEPC8 carrier this option will be ignored.

Selecting “File → Start” will execute function0, which will start a loop of acquiring buffers of data from the ADCs and displaying them. The loop will run at whatever speed is possible given the PC and operating system setup that you have. Within each buffer the ADCs will be sampling at 100Mhz.

The Graphs default to being set up as autoscale off with DC at 0, and max y as 2047. This is correct for the HERON-IO2. You can change these settings if you want by following “Graph → Channel x graph”.

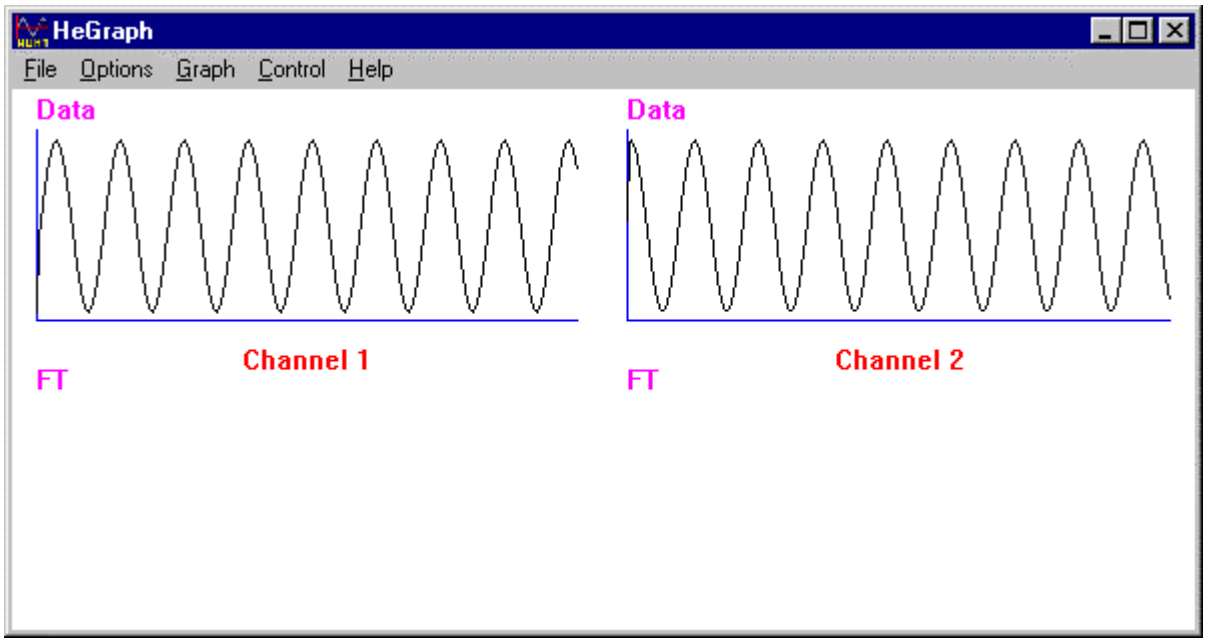
Any signals that are applied to the inputs of the ADCs will be displayed in the window. For the purposes of this demonstration you can connect the DAC outputs to the ADC inputs for each channel. This is OK for the HERON-IO2 as the output and input voltage ranges match.

The Frequency spectrum (FFT) of the signal can be displayed if required by enabling that feature from the “Graph” menu.

When you started the program, values for the DAC outputs were calculated and written to the DAC RAM in the FPGA. Now they will continually be “played” on the DAC outputs. This is known as pattern generation. The starting values were calculated as a Sine wave on channel A and a Cosine wave on channel B. As these will be output by the FPGA over and over again, you will see continuous waveforms at the DAC outputs.

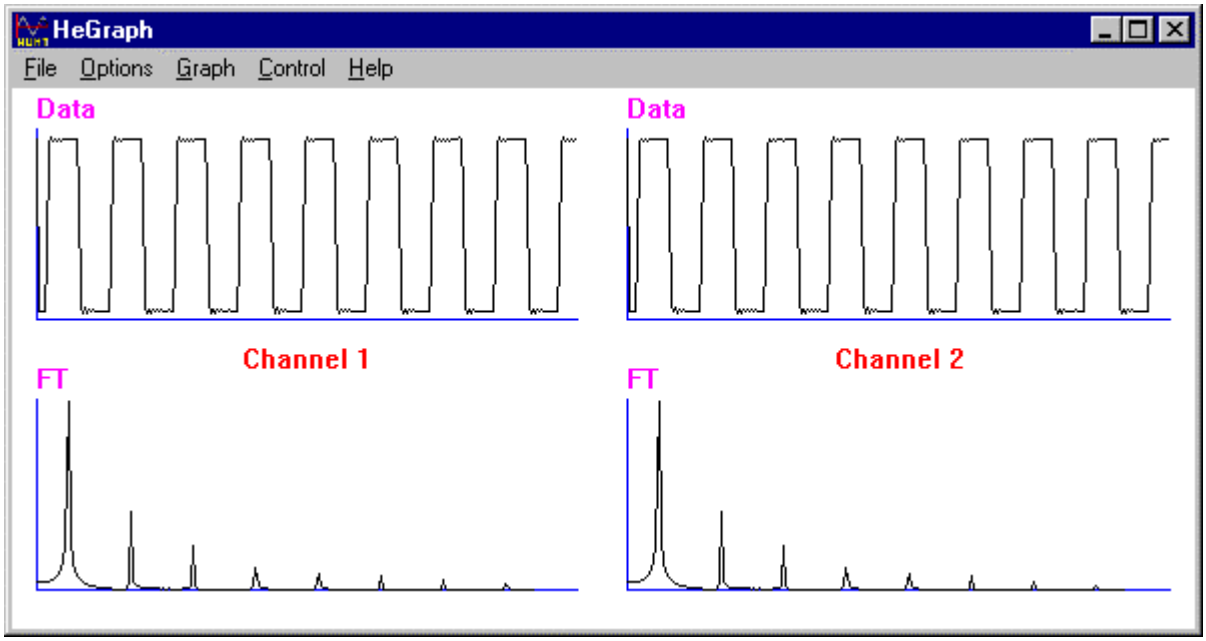
The capture of the ADC values will occur when the Host example program sends a request to the FPGA. This means that each time data is sampled the waveform will be in a different position in the graph.

In the graph you can see that the two inputs have different phases because one is a sine, and the other a cosine.



After selecting File -> Start, the ‘hergaph’ program should display a waveform as shown above. If you don’t see this, verify that you have connected your inputs to your outputs correctly. Also, as already stated you must have successfully loaded the correct HERON-IO2 bit-stream for example 2. Verify that the “Done” LED on the HERON-IO2 is switched off and the User LED4 is flashing slowly.

Selecting “Control → function2” will execute function2, which will re-calculate the values for the DAC RAM to be identical square waves. It then sets a semaphore that indicates to the loop in function0 that the DAC RAM should be updated. The Host writes the pattern to the RAM in the FPGA once and the DACs “play” the data continuously at 100Mhz.



When you have changed the outputs to be square waves, and enabled the Frequency plot you should see the graph above.

You can return to the sine/cos outputs again by selecting "Control → function1" if you want. This will re-calculate the values for the DAC RAM to be the original Sine and Cosine waves. It then sets a semaphore that indicates to the loop in function0 that the DAC RAM should be updated. The Host writes the pattern to the RAM in the FPGA once and the DACs "play" the data continuously at 100Mhz.

Host example: Changing and Building it

The project can be copied from the CD to your hard drive. Then each file needs to have the permissions changed so that they are not read only. The project should build.

The 'HeGraph.exe' program is used to run the demo, control some settings and to display the ADC data graphically. The project contains a single C file that contains the functions that access the Hardware. This makes it easier to look at the example code without getting confused by the graphical display part of the windows program.

The C file is called example2.c and contains a "function0" that captures data from the A/Ds and sends the data to the graphical display program.

There is a "function1" that generates a sine and cosine pattern that is written to the DAC RAMs.

Also a "function2" that generates an identical square wave on both DAC outputs by writing the DAC RAMS with that data. The functions 3 and 4 are not used in this example.

Functions 1 and 2 use a semaphore to indicate to the main thread "function0" that a DAC RAM update is needed.

Now you can make changes to the file example2.c as you wish.

If you no longer require the graphical interface for the demo, you can simply use the C code from example2.c in a C or C++ program for the Host.

C6000 DSP based Example software

HEART carrier like HEPC9

If you have a HEART based module carrier like HEPC9, you can run the DSP based example code with your HERON-IO2 and C6000 modules fitted to any slots. Then you need to configure the HEART connections between the modules using HeartConf, or alternatively use default routing jumpers to make the connection.

If you use HeartConf, then a network file that connects the two modules as follows

```
# For HUNT ENGINEERING's Device Driver API use:
# BD API        Board_type      Board_Id        Device_Id
#-------------------------------------------------------------------------
# Using API
BD API HEP9A 0 0
#
# Nodes description
# ND  BD_nb  ND_NAME  ND_Type  CC-id HERON-ID  filename(s)
#-------------------------------------------------------------------------
  c6   0      dspmodule  ROOT     (0)   00000001  mydspprog.out
  fpga 0      heronio2   normal         00000002  nofile

  ibc  0      ibc1       normal         0x06
  pcif 0      NodeC      normal         0x05

#-------------------------------------------------------------------------
#       from:slot  fifo  to:slot   fifo   timeslots
#-------------------------------------------------------------------------

heart      dspmodule  0      heronio2   0     1
heart      heronio2   0      dspmodule  0     1
```

could be used to make a connection from FIFO#0 of the DSP module to FIFO#0 of the HERON-IO2. Of course you need to modify the HERON-IDs to correctly show which slots your modules are fitted to.

If you prefer to use the default routing jumpers then you can simply fit the jumpers to 0, on both modules for both input and output FIFOs. In this case you are selecting to use timeslot 0 to connect between the FIFO#0 of each module to the other.

HEPC8 carrier

If you have an HEPC8, then you can also fit your HERON-IO2 and C6000 modules to any HERON slots. In this case the FIFO numbers that each module uses will depend on the module slots you have chosen. An example would be if the C6000 module is in slot1, and the HERON-IO2 is on slot3, then the DSP will use its FIFO#2, and the HERON-IO2 would use its FIFO#3.

DSP example : What it does

We supply a c source file for a C6000 based HERON module. Before you can use it you need to make a new project that matches the C6000 module type that you have.

Once the DSP program is compiled and loaded onto the DSP it will be used to gather the data captured by the HERON-IO2 A/Ds into the DSP memory where it can be displayed.

The program is called ‘example2.c’ and it is located in the ‘dsp’ sub-directory of this example. This example can be used both as a confidence check and a starting point for your development.

After you have loaded the correct bit stream into your HERON-IO2 module, the “DONE” LED should be switched off showing that the configuration was successful. Also The USER LED4 should flash about once per second, showing that the clocks are properly running in the FPGA. If the DONE LED is off,

but the LED is not flashing it may be because the Delay Locked Loop used in the clock circuit of the FPGA needs to be reset. You can do this using the utility under "programs → HUNT ENGINEERING → API board RESET" if you want, but such a reset should be made using the HUNT ENGINEERING Reset Plug in for Code Composer.

If you have a HEART based carrier (like an HEPC9) you will need to set the FIFO connections that will be used in your system. You will do this using the Heartconf program, which can be called from the HUNT ENGINEERING reset plug in for Code Composer Studio. This is probably the best way as it will re-configure your connections every time you reset the system. If you are not already comfortable doing this you should review the HEART movies and documentation again.

When you run the example2 dsp program it will ask you some questions about your system configuration. You need to correctly answer these questions regarding the way that your modules are connected.

The program will send HSB messages to set the FIFOs that the HERON-IO2 FIFO will use according to your answers. The program will then calculate a Sine and Cosine pattern that is sent to the DAC RAM in the FPGA. This will be continually output from the DACs of the HERON-IO2. The program will then start to capture blocks of data from the HERON-IO2 A/Ds into the DSP memory. Then the DSP will "unpack" this data into a separate array for each channel, with the correct sign extension for displaying a graph of the data using Code Composer's Graph routines. Optionally you can also display an FFT of the data coming from the A/Ds (which is calculated by Code Composer Studio).

DSP example: setting it up

If you have an HEPC8 your FIFO connections will be determined by the position of the modules on your carrier board.

If you have an HEPC9 however the FIFO connections are determined by the settings that you made.

The example that we supply is a C file called example2.c. It needs to be changed to reflect your actual needs, and then built using Code Composer Studio.

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" from inside Code Composer Studio. This will guide you through setting up the project and as long as you choose the name "example2" for the project it will incorporate the example2.c source file.

This project will incorporate the correct HERON-API library for your module and Module carrier combination.

You must review some settings at the top of the source file, and change them to reflect your system setup.

The first line is

```
#define FIFONO                  2  /* FIFO through which module communicates with FPGA */
```

This is the fifo that the C6000 uses to communicate with the HERON-IO2. For example if you have a DSP module in slot1 of an HEPC8, and the HERON-IO2 is in slot2, the setting needs to be 2. If you have an HEPC9 you must set whatever you have chosen, which is 0 in both of our HEPC9 configuration examples given above.

The next thing is to set the line

```
#define FPGA_TO_DSP          3
```

This is the fifo that the HERON-IO2 uses to communicate with the C6000. For example if you have a DSP module in slot1 of an HEPC8, and the HERON-IO2 is in slot2, the setting needs to be 3. If you have an HEPC9 you must set whatever you have chosen, which is 0 in both of our HEPC9 configuration examples given above.

Finally you need to change the line

#define FPGA_SLOT 2

to show which module slot the HERON-IO2 can be found in. This is used to address the HSB messages.

Then you can build the example using Code Composer Studio.

DSP Example: using it

To run the example, load the program onto the DSP using the File→Load command of Code Composer. At this stage it is advisable to open the HUNT ENGINEERING Reset Plug in, and set the option to "halt processors, reload them and then run to main". If you have a HEART based carrier and you will use HeartConf, select this also in the reset plug in. Then use this reset to reset the system, and reload it, which empties the FIFOs, and configures HEART if you need to.

But before you can run the example, you must have remembered to load the bit-stream onto the HERON-IO2. What bit-stream to choose is shown above. If you don't know how to load a bit-stream onto the HERON-IO2, please review example1 once more. Verify that after the loading process the "Done" LED goes off, and now the system has been reset (using the plug in) the USER LED4 should be flashing.

It doesn't really matter in what order the DSP and the HERON-IO2 is loaded. You may just as well first load the bit-stream and only then load the DSP. Finally, do a "Debug->Run" in Code Composer Studio to start the example.

The example2 starts by configuring the FIFOs using the HSB. The fifo number that is programmed is set by the "#define FPGA_TO_DSP 3" near the top of the program. You should have changed this to reflect which FIFO you want to use.

The program then enters a loop that requests a buffer, receives a buffer and uses a function from the HEL_Unpack library to split the data up from the two channels. Note the use of the Se version of the unpack function so that the sign bit is extended making the data be the C type "signed short".

The array "errors" is used to perform an "OR" of the top bits – allowing us to detect if the THU or OTR bits have been set anywhere at all in the buffer.

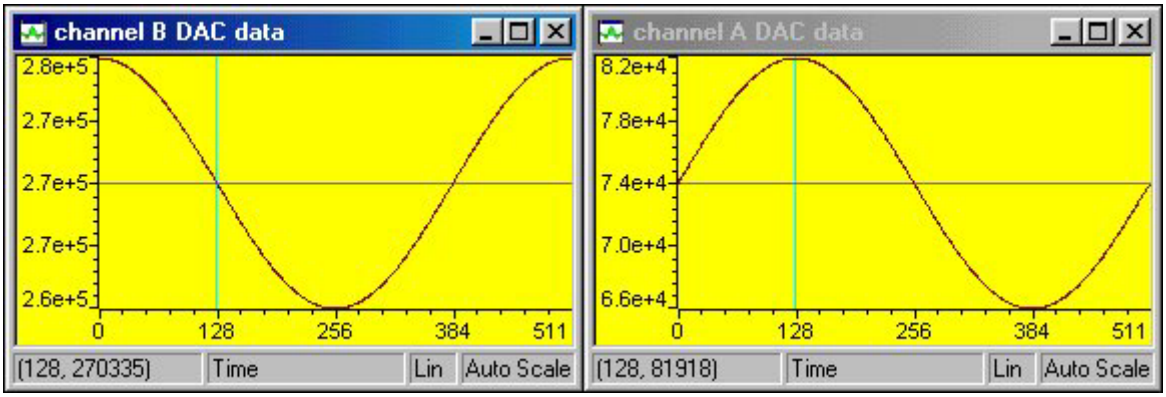
In Code Composer you can check the data that is being sent to the DAC rams by displaying a graph of each DAC data array. If you do not know how to use the Code Composer Graph functions please review the Code Composer documentation and tutorials again.

The DAC arrays are calculated in the lines

```
for (i=0;i<512;i++)
{
chadata[i]=0x10000 |(int)(0x1fff+ 0x1fff*( sin(i*(PI/256))));
chbdata[i]=0x40000 |(int)(0x1fff+ 0x1fff*( cos(i*(PI/256))));
}
```

In Code Composer you can add a break point on the "HeronDigioOut " line.

You can display a graph of channel a data by selecting "chadata" or "chbdata" as the address, an "acquisition buffer size" and "Display data size" of 512, and "32 bit signed integer" for the type. Finally you need to deal with the control bits that have been added, which are actually just a DC offset. For channel a this is 0x10000 + 0x1fff, for channel b this is 0x40000 + 0x1fff. So select the DC value box as 0x11fff and ox41fff respectively.



The graphs show the data that will be output to the DACs. Notice the vertical bar marking the 128th sample. For the purposes of this demonstration you can connect the DAC outputs to the ADC inputs for each channel. This is OK for the HERON-IO2 as the output and input voltage ranges match.

Any signals that are applied to the inputs of the ADCs will be captured into the DSP memory using the HERON-API functions:-

```
status = HeronRead(rfifo,readdata,ARRAY_SIZE);
if (status != HERON_IO_IN_PROGRESS)
{
        printf("error cannot start the read\n");
        exit(-1);
}
status = HeronWaitIo(rfifo);
```

When you started the program, values for the DAC outputs were calculated and written to the DAC RAM in the FPGA. Now they will continually be “played” on the DAC outputs. This is known as pattern generation. As these will be output by the FPGA over and over again, you will see continuous waveforms at the DAC outputs.

The capture of the ADC values will occur when the example program sends a request to the FPGA over the HSB:-

```
SendUserDatanr(FPGA_SLOT, 2 ,(ARRAY_SIZE&0xff)); /* bottom part of count */
SendUserDatanr(FPGA_SLOT, 3 ,((ARRAY_SIZE>>8)&0xff)); /* top part of count and request to start*/
```

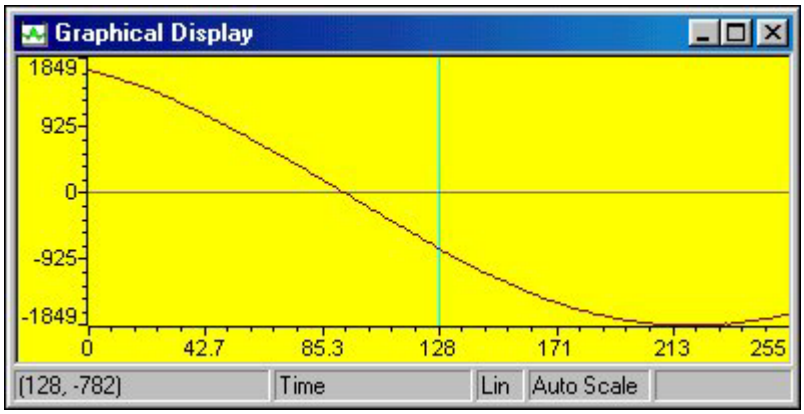
This means that each time data is sampled by the ADCs the waveform will be in a different position in the graph.

The line

```
HEL_UnpackSe16bit2chan ((short *)readdata, DATA_MASK, shortdata ,SAMPLES_PER_CHAN, errors);
```

uses the unpacking function to strip the two data channels into different arrays, to remove the control bits and perform a proper sign extension on the data.

You can display that graph by selecting “shortdata” as the address, an “acquisition buffer size” and “Display data size” of 256, and “16 bit signed integer” for the type. Because the data has already had the control bits removed the DC value in this case is 0.

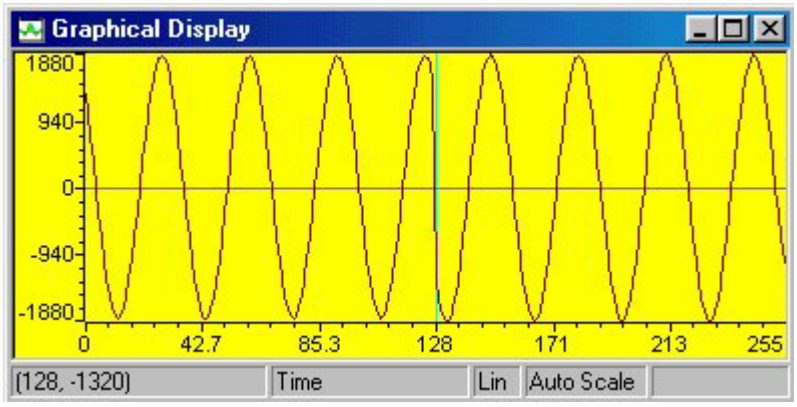


The array “shortdata” actually contains both channels of ADC data, with A in the first 128 samples and B in the second 128 samples. In the graph above you cannot see the change, which is because 128 samples of the Sin exactly lines up with the first 128 samples of the Cosine. Look at the original graphs to check it.

For our demo we’d like to have more cycles of the signal , which we can achieve by increasing the frequency of the DAC data. We can do this by editing the lines to be

```
for (i=0;i<512;i++)
{
chadata[i]=0x10000 |(int)(0x1fff+ 0x1fff*( sin(i*(PI/16))));
chbdata[i]=0x40000 |(int)(0x1fff+ 0x1fff*( cos(i*(PI/16))));
}
```

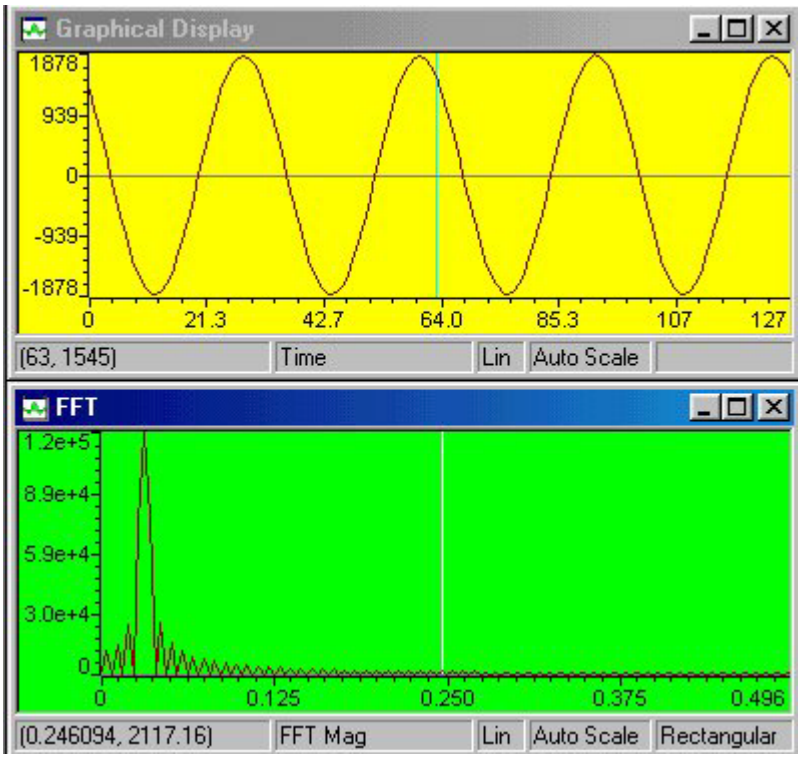
Rebuilding and resetting/reloading the system changes your results graph to be :-



Now you can see the discontinuity between the graphs of the two channels.

If we want to look at the frequency content, we should look at only one channel, so reduce the an “acquisition buffer size” and “Display data size” to 128. Then we will see channel A only.

We can add a new graph, of type FFT magnitude, but the same settings as the results graph.

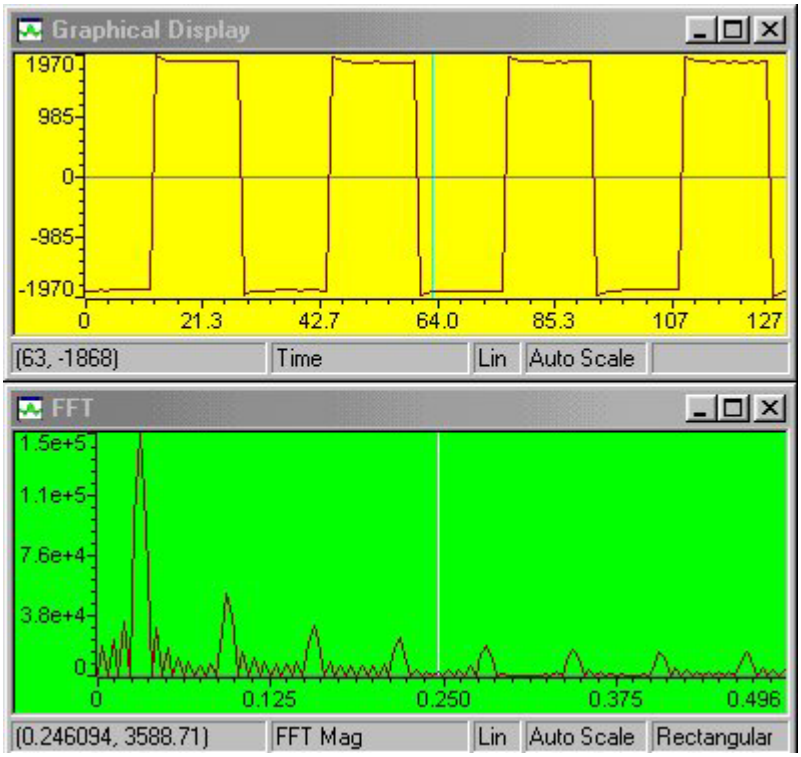


The frequency spectrum is “nearly” a single point, the smaller items are being introduced by the sampling process on the output of the DACs and the input of the ADC.

If you now change the your DSP program again to have

```
 for (i=0;i<512;i++)
 {
            if ((i&0x10)==0)
            {
                        chadata[i]=0x10000 |(int)(0);
                        chbdata[i]=0x40000 |(int)(0);
            }
            else
            {
                        chadata[i]=0x10000 |(int)(0x3fff);
                        chbdata[i]=0x40000 |(int)(0x3fff);
            }
}
```

for the DAC data calculation, you will have a square wave on the outputs, and the new results will be:-



Again the square wave is not perfect as the system bandwidth is not infinite, but now you can see the various frequency harmonics that are contained in the square wave signal.

DSP example: Changing and Building it

In the sections above you have already changed and built the DSP code a number of times. It uses DSP/BIOS and HERON-API.

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.

FPGA example code

You should understand the HUNT ENGINEERING VHDL support for HERON modules before looking at this section. If you do not then please review example1 again (the getting started example for FPGA modules).

This section only discusses therefore the things that are unique to example2.

Example2 has some options in the user_ap2.vhdl file that allow you to select some options for the FIFO clocking.

Just like in Example1 (the getting started example) you can select if the 100Mhz is divided by 2 to provide the FIFO clock frequency. This allows example2 to be able to clock the FIFOs at 50Mhz (suitable for HEPC8) or 100Mhz (suitable for HEPC9). You must also remember to set the HIGH_FCLK_G option to show if this clock is higher or lower than 60Mhz.

For example2 the correct options for an HEPC8 are :-

DIV2_FCLK	FCLK_G_DOMAIN	HIGH_FCLK_G	HIGH_FCLK_RD	HIGH_FCLK_WR
True	True	False	n/a	n/a

So both input and output FIFOs will be clocked at 50Mhz.

For example2 the correct options for an HEPC9 are :-

DIV2_FCLK	FCLK_G_DOMAIN	HIGH_FCLK_G	HIGH_FCLK_RD	HIGH_FCLK_WR
False	True	True	n/a	n/a

So both input and output FIFOs will be clocked at 100Mhz.

Example2 also has some options that need to be set for the A/D clocking.

For example2 the correct options are :-

SCLK_G_DOMAIN	INTERNAL_SCLK_A
True	True

And the EXT CLK jumpers MUST NOT be fitted

There is a Dual Port RAM placed for the DAC RAM, and a FIFO that is taken from the Core Generator used in this example. If you want to understand using Core Generator cores in a design then review the “DSP with FPGA” tutorial.

You need to consider the timing constraints that are defined in the .ucf file for your design. Actually if you use a time specification that is more strict than needed there is no problem, so the standard .ucf file have the clocks specified at 100Mhz. If the project builds (as example2 does) with this specification it is still guaranteed to work at lower clock speeds. If you add new clock nets into your design then you need to add new timing constraints into your design.