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

HUNT ENGINEERING

C6000 imaging demo/framework

An Example of capturing images with HERON-FPGA modules with standard IP, image processing on the C6000 using the Hunt Engineering Imaging Library and displaying the results on the host PC's graphics display. Originally written for Windows 98, but should work on any 32-bit Windows system.

Software Version 2.1
Document Rev B
R. Weir, J.Thie 16/12/05

TABLE OF CONTENTS

Introduction

Modern DSPs can be used to build powerful image processing systems. Many standard imaging functions are easily programmed for a DSP.

HUNT ENGINEERING have produced an image processing library for the C6000 that performs many of these standard functions. This demo/framework is provided to a) demonstrate that library and b) act as a starting point for customers that want to develop image processing applications using HERON systems.

There are standard FPGA designs for the HERON-FPGA modules that offers the interface to a variety of camera types. This is used to feed the image data to a C6000 based HERON module that performs image processing and feeds the results to a HERON FIFO. This is combined with a PC based windows program that controls that DSP program and displays the results in a Windows window. Both of these items are provided as source code allowing them to be used as a starting point for your own development.

Limitations of display

In many imaging systems there is a requirement for display. Sometimes this is a "hard" requirement, where the display needs guaranteed update rates, or high resolution / high frame rates; however in many systems, the display is not crucial to the system's operation.

Examples of such systems include a display for focusing or aligning cameras, a confidence checking display, or a monitor to display processed results. In such cases, it is possible to use the host PC for display.

This is an example of using HUNT ENGINEERING imaging boards in this way. It captures video, performs some processing, then transfers the resulting video to the PC for display on the PC's monitor. This is done entirely using Windows function calls, so should be portable across all 32-bit Windows systems.

The example also demonstrates use of the HUNT ENGINEERING imaging library to perform basic logical, arithmetic and filter operations. It can self-run, or be set to each task.

The software has been written using the HUNT ENGINEERING software infrastructure (Server/Loader, HE-API and HERON-API). This allows it to be easily ported from one platform to another.

Performance of the system depends very much on the bus bandwidth of the PC being used. The demo can be easily ported to any platform using the HUNT ENGINEERING software infrastructure.

Notes:

1. It is possible to re-compile to operate without a camera or framegrabber. In this mode it transfers test images to the PC for display; this can assist in debugging.
2. The demo uses 8-bit monochrome video. The most efficient display settings for the application are 24-bit or 32-bit colour – this reduces colour space conversion to three copy operations rather than colour search / match operations. Frame rate may drop if you use 16-bit or 8-bit mode.

Inside the Demo

Overview

The demo as-is will work with a HUNT ENGINEERING system that has an HERON4-6701 or HERON2-6203 module in slot 1, and a HERON-FPGA3, HERON-FPGA4, HERON-FPGA5 or HERON-FPGA7 module in slot 2. The demo will program the HERON-FPGA3 with a standard frame-grabber bit stream. The demo works with the HEPC9 and with the HERON-BASE2.

The demo should run on any PC with any 32-bit Microsoft Windows installed on it; however, it will operate better on faster machines with accelerated graphics cards.

Camera's supported are Camera Link and RS422 area-scan camera's, with at least 384 x 384 visible pixels. The pixel size should be 14 bits or less. The demo assumes 8 bit pixels. Pixels larger than that will be displayed as-if they are 8 bits. (The pixels will be right-shifted by such an amount that 8 bits are left, i.e. the most significant 8 bits will be used.).

The demo consists of a Windows host program, a DSP program using DSP/BIOS, and bit-streams for the FPGA modules. The bit-streams used are the standard Camera Link IP and RS422 Camera IP from the HUNT ENGINEERING CD.

There are different batch files, that run the demo depending on whether you use HERON2 or HERON4, which FPGA module you use, and whether you use a Camera Link camera or a RS422 camera. The naming of the batch files should indicate what modules and camera the batch file expects. The batch files are located in the Release directory of the \software\examples\utilities_etc\windows_display_demo directory on the HUNT ENGINEERING CD. It is designed so that the demo can be run off the CD by double-clicking the appropriate batch file.

Use of APIs etc

The demo is based around HUNT ENGINEERING's standard APIs and utilities. These give it great portability, allowing us to switch easily from one hardware platform to another. We discuss them a lot in the following outline; here is an introduction to their function.

Host API / HEAPI	HE-API is the standard software interface to any Hunt Engineering board. It provides a simple mechanism to control the board and transfer blocks of data to and from the DSP. In this application it is used to pass images from the DSP to the host.
Server/Loader	A more sophisticated tool, Server/Loader, gives the host far more control over the DSP board, plus adds extra services that the DSP can use. In this application we use Server/Loader to reset the DSP, load the software and start the processor.

Server/Loader also offers the DSP the ability to access the host's file system, or perform text-based I/O to a console window. These facilities were not used in this application.

With Server/Loader, switching from one board to another is simply a case of changing a text-based control file, which defines the system.

HERON-API	This API provides the DSP processor with easy to use access to the hardware. It includes functions for communications, with 'peripherals' (such as the HERON-FPGA3), with the host, or with other DSPs. These communications routines use the DSP's DMA controllers, so can be used to perform very efficient I/O. Other functions available in the HERON-API provide control over hardware facilities for digital I/O, UMI, etc. By using HERON-API, DSP code is portable – to switch the code used here from the HERON4 module to the HERON2 or vice versa is a matter of changing an include file (#include "heron2.h" instead of "heron4.h") and rebuilding – no other code changes would be required.
DSP-BIOS/2	The DSP code is built on DSP-BIOS/2, using its semaphore model to control the hardware, synchronise tasks to the framegrabber and so forth. This is a powerful real-time kernel included as standard with Code Composer Studio. HERON-API makes use of it extensively.
HEL_Imglib	This is a library of image processing primitives, allowing imaging systems to be put together quickly. It is highly optimised, with all processing routines written in assembler, and extensive use being made of the DMA to copy blocks of data on & off chip.

DSP Hardware

The demo has been tested on a variety of hardware, as follows:

- Various Camera Link digital cameras from Pulnix and Jai
- Pulnix TM 300 RS422 digital camera
- HERON-FPGA3 with standard Camera Link IP bit stream and with RS422 Camera IP bit-stream
- HERON-FPGA4 with standard Camera Link IP bit-stream and with RS422 Camera IP bit-stream
- HERON-FPGA5 with standard Camera Link IP bit-stream and with RS422 Camera IP bit-stream
- HERON-FPGA7 with standard Camera Link IP bit-stream and

with RS422 Camera IP bit-stream

- HERON4-C6701 DSP processor module
- HERON2-C6203 DSP processor module
- HEPC9 PCI carrier board
- HERON-BASE2 carrier board

Full use of the APIs provided with HUNT ENGINEERING systems allows easy portability to other platforms; however bear in mind that other cameras may require different configuration, while different HERON modules would require different versions of the HERON-API.

PC Hardware

The demo should be compatible with any 32-bit Windows system. Obviously, it uses considerable bus bandwidth for transferring video, and performs rapid bit-blits to place the video on the screen. Faster PC systems will perform this better, but the demo should still operate on older systems with no graphics acceleration and slower processors.

This raises a design issue we must be aware of – in many circumstances the DSP and framegrabber will be able to supply images far faster than the PC can display them. This is taken into account in the DSP software through video buffering.

Configuration File

There's a text file included in the demo that holds some parameters that are used by the demo to configure the camera used. This allows you to use the demo for some variety of Camera Link and RS422 cameras, and also gives you some degree of control over what the demo does.

The file is called 'camera.dat'. In here you can find the following parameters.

```
visible = 1
```

This parameter determines what part of the camera image is displayed. If visible is set to 1, then the demo will display the centre 384x384 pixels of the visible area. If visible is set to 1, then the demo will display the centre 384x384 pixels of all of the camera image. The HUNT ENGINEERING Camera Link and RS422 Camera IP will try to figure out the 'visible' area, that part of the image that isn't black. When visible is set to 1, the demo makes use of this feature.

```
logfile = 1
```

This parameter tells the demo whether to keep a log file or not. This maybe helpful in troubleshooting if something doesn't work. The log will be written to a file called "wddemo.log" in the c:\ root directory.

```
imglib       = 1
```

With this parameter you can tell the demo to do the actual Imaging Demo, thus for example rotations, filters, etc. But you can also tell the demo to simple display the images without doing any image processing.

```
cam_control  = 0xf
```

This is a parameter that represents the camera control value.

```
cam_use_dval = 0
```

Different Camera Link and RS422 camera's may use slightly different protocols. One of them is whether the 'dval' signal is used or not. With this parameter you can change that setting.

```
cam_polarity = 1
```

Different Camera Link and RS422 camera's may use a different 'polarity'. In one case, a logic 1 may mean 'active', while for another camera a logic one may mean 'not-active'. With this parameter, you can indicate the polarity of the signals of the camera used.

DSP Code

The DSP code performs the following tasks:

- Create and initialise multiple image buffers. Each is set to a test pattern which allows the system to be tested with no camera present. The buffers are in off-chip memory (typically SDRAM).
- Initialise the HERON-FPGA module. Some camera specific data is read in from a file (camera.dat) via the Windows host program. This is done to suit the camera in use. Edit this file (camera.dat) or modify this section of the code for different cameras.
- There's a read task that continuously captures images into three image buffers. One of the image buffers is in use by the processing task. If processing and data exchange with the Windows program is slower than the read task can read images, it will 'jump' between two image buffers not used by the processing task. Once the processing task is ready, it will use the last-read image buffer, whilst 'freeing' the image buffer it just used. Semaphores are used to synchronise this image buffer swapping between read and processing tasks.
- There's a processing task that continuously processes images read in by the read task. Overlapping with the processing itself, this task sends processed images to the Windows PC program. Two image buffers are used here: one that gets sent to the Windows PC program, another to receive processed data. After one loop, these image buffers are swapped between processing and sending.
- There's also a control task. This can receive, at any time, control data from the Windows PC program. Typically, the control data asks the DSP program to do certain operations (filter, copy, …) depending on what phase the Windows PC program is in. This so that what the Windows PC program writes as text on screen is synchronised with the processing on the DSP.
- The read task has the highest priority. This way, even if the communication between PC and DSP is slow, or even stops, the read

task can always read. If not, buffer overflows may occur (the FPGA to DSP FIFO may get full) and the image starts to 'drift'. The control task has the highest priority after the read task. We don't want control task to mess up reading images from the FPGA, but there's no problem interfering with the processing (task).

Image Buffers

The buffering system used was designed to allow the DSP to drop frames whenever the PC cannot sustain the display rate. This happens under many circumstances, such as PCI bus overload, where the images cannot be transferred in real time; or when the PC is performing other operations, such as running another application.

The buffering system uses five buffers to give the best possible transfer rate to the PC, along with reliable framegrabbing.

For capture, three buffers are used. The framegrabber continuously captures into these, ensuring that it does not capture into a buffer currently being used for processing. Essentially this means that the framegrabber will ping-pong between two buffers, while the processing routine operates on a third; when the processing is complete, it can then grab the freshest capture buffer, and free the source buffer it had been using.

Display works a bit different. Processing and display are done within the same loop. While processing is done on one image buffer, the other (previously processed) buffer is being sent to the Windows program (in the background, using the C6x's DMA engines). Just two imaging buffers need to be used for this processing and display task.

This set-up is essential where we don't know how fast the PC will be able to display the images, but we also want a robust system. With this buffering scheme, if the processing is too complex to sustain full frame rate, there will be an elegant degradation with dropped frames rather than a system failure!

Code Structure

The code has four components:

- initialisation
- read task
- processing and write task
- control task

There's a task called 'maintask' in the 'demo.c' C code file. This task first does the initialisation and then does image buffer processing and sending buffers to the host PC. The initialisation sets up the imaging buffers, opens the FIFO's, sets up the camera (and up-keeping the log file all the time). If everything is ready, it starts the processing and write sub-routine.

The processing and write task first starts the control task (be means of a sema-phore) then the read task (also by means of a semaphore). It then waits for a signal from the read task that a buffer is ready. Every time a buffer is ready it starts a write transfer of the previously processed buffer to the host PC. After starting the transfer, thus, while the write transfer is in progress, the task will process the buffer just received from the read task. After the processing is done, it checks whether the write transfer has completed. Once this is done, it loops round to wait for the read task to give it the next read buffer.

The read task, once started, simply loops around as fast as it can, reading frames sent to it by the FPGA IP. The very first time it loops it sends a message to the IP, to indicate it should start catching frames and send it. As soon as a frame has been read, it sets a semaphore to indicate to the processing and write task that a buffer is ready to be processed.

The read task uses three buffers. One of them is 'allocated' by the processing and write task. If the processing and write task is slower, the read task always has 2 buffers remaining to capture into. If one buffer was just filled, it captures next into the other buffer. And then fills the first one again. Once the processing and write task is ready, it 'allocates' the last-filled buffer. The read task can now 'ping-pong' between the now de-allocated buffer and its current capture buffer.

The control task is waiting, most of the time. When the host PC demo switches to a new 'mode', it sends a message to the DSP to do a different 'type' of processing. The control task reads the message and sets the necessary parameters.

The read task has the highest priority. We need to schedule the next HeronRead as fast as possible. The time in between the end of one read transfer and the next HeronRead the FIFO's fill up. The faster the data rate is (between FPGA and DSP) the shorter the time is you have (to start the next HeronRead).

Host Code

The host code is a Windows application. Understanding it requires some knowledge of the Windows operating system.

The application is split into three main sections:

1. User Interface / application code, which deals with the user interface, updating the display, moving the screen and so forth;
2. WndPROC, the window message handler. This function is registered as the handler for all messages for our display window.
3. DSP code, which boots the DSP card, and performs all data transfers to and from the board.

The application is multi-threaded. The main thread performs the User Interface code, while the second thread handles the DSP board. Splitting the application in this way greatly improves the usability of the system; user interface options (eg moving the window, selecting a menu, closing the application) can be performed while the application is reading the DSP card. This is a great benefit where a debugger like Code Composer is in use.

Note WndPROC is called by the Windows operating system itself – there are no calls to this function in the code.

User Interface / Application Thread

This is the main thread of the application. It performs the following tasks:

1. Open a window for image display
2. Create two bitmaps and initialise with a test pattern
3. Start the DSP code thread
4. Start the timer
5. Process Windows system messages

Once these tasks are performed, this thread simply polls the Windows message queue and despatches messages. In the main, these are either handled by the system or by the WndProc function.

Key messages arrive from the timer and from the user. Timer messages are used to switch the DSP to its next operating mode, while the user can request a switch manually.

WndProc

This function is called to handle all messages to our display window, from whatever source. Note that we don't call this function ourselves – it is registered as the handler for the window, and it is called by Windows directly.

There are three messages handled here that are worth noting:

WM_PAINT	This is the standard "Window Paint" message. Windows will send this message whenever the window needs an update – perhaps because it has been moved, or more of the window has been displayed; or part of the window has become valid. This paints the window. Display is performed using a Windows BitBLT call – this copies the image directly to the screen. StretchBLT would perform scaling – a call to StretchBLT is in the code, commented out, should you wish to try this. Note that to use StretchBLT you will need to change the window class – it is currently defined as non-resizable.
DSP_IMAGE	This message is sent by the DSP code thread. It indicates that a new image is available for display. The message causes the display window to be invalidated, causing a WM_PAINT event (See above).
DSP_STATUS	This sets a flag for WM_PAINT, indicating that the status text has been changed. It also invalidates the screen. This causes WM_PAINT to redraw the status bar.

DSP Thread

The host's interface code for accessing the DSP board is contained in a separate thread, started by the main application. This is not strictly necessary – it would have been possible to write the application with a single thread. However, multi-threading makes the code simpler and more robust for the user.

The code uses the HUNT ENGINEERING Server/Loader for starting the system. This performs all booting operations, such as resetting the board, loading the code and starting execution. The Server/Loader uses a network file in which the board type and modules in your system are defined; they are the *.net files, in the Release\nets directory. Each HERON DSP and HERON-FPGA combination has its own network file. The carrier board defined in the network file is the HEPC9. To avoid have two copies of each network file, the same network file is used for the HERON-BASE2. What the demo will do is scan the "BD API" line, and replace the board definition by the default board. The default board is set using the Confidence Checks program.

Once booted, the host closes the Server/Loader and opens an API session for the board. This allows us to create our own data passing mechanism across the interface, allowing us to send commands to the board and receive images back. Had this been a text-based application, we would probably have kept Server/Loader running, giving us access to stdio functions like printf, fscanf and so forth.

Once booted, the code continuously reads images from the board. When a complete image is read, the DSP thread sends message DSP_IMAGE to the display window – this forces an update. The code also initiates a read of the next image.

Images for display are double-buffered. This ensures that we always have one good image ready for display, while a second image is being received from the DSP. Thus, should the display need updating for any reason other than a new image being available, the old image can be redrawn. This covers the various screen updates that Windows can throw in – for our window being moved, or for a screensaver for example.

File Functions & Locations

The example is supplied as a Microsoft Visual C++ project. This contains the following directory structure:

Directory	Description
….\	Root directory for the project; contains project management files. Also contains documentation for this example.
….\DSP Code	Code for the DSP. Several files, whose function is outlined below.
….\Host Code	PC code for controlling the system and displaying the images. Details of the files are given below.
….\Shared	Shared header files for the project, containing information common to both PC and DSP code.
….\Release	The application itself. Visual C++ will build the application in this directory; the DSP make-file will also place the code here.

DSP Code Directory

File	Description
Demo1.pjt	Code Composer build file to configure this project (HERON4)
Demo1.cdb	Code composer configuration file (HERON4)
Demo2.pjt	Code Composer build file to configure this project (HERON2)
Demo2.cdb	Code composer configuration file (HERON2)
camspec.h	Definitions for the FPGA framegrabber
Def.h	Definitions for the camera
Demo.c	Main source code for demo.
Demo1_stub.c	Stub file to 'connect' Server/Loader and HERON-API (HERON4)
Demo2_stub.c	Stub file to 'connect' Server/Loader and HERON-API (HERON2)
DSPprocess.c	Processing harness & logic processing demo routines
DSPfilt_process.c	Filter processing routines
DSProt_process.c	Rotation demo routine
HRNRotCopy.c	Rotate functions – flip/mirror image

Host Code Directory

File	Description
Img_Display.cpp	Source code for demo. Note that although this is a .cpp file, it is almost straight C –the Server/Loader code is in C++ to be able to support CCS plug-in functionality.
Img_Display.rc	Resource file for the demo. This file defines the menus that appear on the application's menu bar.
Resource.h	Resource header associated with the resource file.

Release Directory

The Release directory is where the application is built into. It contains the following:

File / Directory	Description
demo.exe	Main PC application. Start the demo by using one of the batch files described below in this table. Starting 'demo.exe' on its own will have the demo look for a network file called 'network', which doesn't exist in this directory. The batch files each use a network file stored in the 'nets' sub-directory.
DSP_Camera4.out	DSP executable for HERON4. This is built by Code Composer Studio (v. 2.1). It is downloaded to the DSP by the Server/Loader (library). Source code is in the DSP Code directory.
DSP_Camera2.out	DSP executable for HERON2. This is built by Code Composer Studio (v. 2.1). It is downloaded to the DSP by the Server/Loader (library). Source code is in the DSP Code directory.
*.bat	Batch files that start the demo with parameters as indicated by the naming. Choices are HERON4/HERON2, FPGA3s/FPGA3v/ FPGA4, and Camera Link ('_cl') or RS422 ('_rs422') camera.
Camera.dat	Text file with demo and camera specific parameters.
Nets directory	Server/Loader network files that match the different batch files. Each network file in this directory specifies what to load onto the DSP and what to load onto the FPGA. It also defines and creates HEART connections on a HEPC9.

Note that in addition many temporary files are created – listings, object files and so forth. These may not be included in the distribution of the demo to reduce size.

Shared Directory

The Shared directory contains data shared between the host and DSP applications. It contains the following files:

File	Description
Image_defs.h	Definitions of the image size in use. Shared between the host and DSP.
Msg_Defs.h	Definitions of the messages passed between the DSP and host. Shared between the host and DSP.

Rebuilding the CDB File

The DSP projects provided are created by and for Code Composer Studio version 2.1. If you have a different Code Composer Studio version, you cannot use these projects. You will have to create a new project (using the Create New HERON-API Project plug-in, and create the new project for use with the Server/Loader).

The demo uses more than 2 tasks (besides 'maintask') and several semaphores. You also need to change some stack sizes, because the processing task stores some quite large frames on the stack. It does this because stack gets allocated on (fast) IDRAM. After you created a new project, you will have to re-insert the different items.

In your new project, open the CDB file, and then add 2 tasks:

1. Add another task, e.g. TSK1. Change 'task function' to '_read_task'. Set Priority to 3.
2. Add another task, e.g. TSK2. Change 'task function' to '_control_task'. Set Priority to 2.
3. Make sure that 'maintask' (probably TSK0) has Priority 1.

In itself, you can allocate priorities as you like. But make sure that 'maintask' has the lowest priority of the three tasks, 'read_task' the highest, and 'control_task' in between these two.

Next, make sure enough stack is allocated for 'maintask':

1. For 'maintask' (TSK0) set 'Stack size (MAU's)' to at least 32256.
2. For 'read_task' (TSK1) the default 1024 is enough.
3. For 'control_task' (TSK2) set 'Stack size (MAU's) to at least 512.

Depending on the HERON type, with the standard CDB you will probably not be able to cram all this into IDRAM (when you build the application, CCS will complain at link time). So you need to make some other sections smaller or swap them from IDRAM to external memory. For example, you can change the IDRAM heap size 0x1000. This already gives you 0x3000 bytes extra compared with the default 0x4000 size. If this doesn't do it, swap unused CDB objects out of IDRAM. The standard project (as created by the Create New HERON-API Project plug-in) is configured to create a map file when you build. The map file will show you where software sections are mapped onto, and also gives you an idea how much data/objects are stored where.

Next, you have to add some semaphores:

1. Insert a semaphore named 'ctrl_done', and set 'Initial semaphore count' to 0.
2. Insert a semaphore named 'read_buf_ready', and set 'Initial semaphore count' to 0.
3. Insert a semaphore named 'read_done', and set 'Initial semaphore count' to 0.
4. Insert a semaphore named 'read_mutex', and set 'Initial semaphore count' to 1.
5. Insert a semaphore named 'start_control', and set 'Initial semaphore count' to 0.
6. Insert a semaphore named 'start_read', and set 'Initial semaphore count' to 0.
7. Insert a semaphore named 'write_done', and set 'Initial semaphore count' to 0.

Once done, save the cdb file and build the project in the usual way. Be aware that the demo uses a DSP executable in the demo's Release directory. The standard project will probably create an out file somewhere else.

Rebuilding the Project

The DSP projects provided are for HERON4 (demo1.pjt) and HERON2 (demo2.pjt). They use the same source code. However, when rebuilding, make sure that you change the heron include files to the proper type. Thus, if you want to build for HERON4, look through all C source files, and replace any

#include “heron2.h”

by

#include “heron4.h”

And vice versa; if you want to build for HERON2, look through all C source files, and replace any

#include “heron4.h”

by

#include “heron2.h”

More in general, if you have a HERON type not used here, then replace any such include file by an include file that matches the HERON type you are using.

Known Limitations / Possible Enhancements

There are a number of features and facilities we could add:

1. Direct Draw support. The current version uses standard 32-bit Windows calls to perform display. DirectDraw offers a potentially faster way of implementing display, but at the expense of lack of portability.

   Many older machines do not have DirectDraw support installed, requiring a more complex installation program for the demo. Also, the main benefit to using DirectDraw would be accessing a hardware bit-blitter and colour space converter. Not all graphics cards have these, so we may get relatively little gain from the use of DirectDraw on older machines. It would also add to the complexity of the demo, making it much harder to understand!

2. Image Storage could be added, archiving video to disk.
3. Multiple Window Display