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

Developing a complex real time system with HERON and HEART.

Rev 1.0 P.Warnes 23-12-02

The HERON and HEART product range is designed for use in real time systems. HUNT ENGINEERING provides the hardware and the tools *you* need to be able to develop *your* application with them.

While each application is actually trying to solve a different problem, each application will be attempting to use the same data flow model with the HERON and HEART hardware. As such they will all have similar requirements, and the techniques that can be used in developing the system will be similar.

This document takes a system that has an FPGA module, a C6000 module, a HEART carrier and a PC, and shows how to develop test and optimise each stage before moving onto the next.

Actually the hardware used was an HEPC9, HERON-IO4 and a HERON2. The techniques used will be the same with other combinations of modules, but the performances that can be achieved may differ. The absolute measurements have been included in this document to give a feel for the difference each change makes to the system performance.

The author is a hardware engineer, so hardware test tools are easily available. For example extensive use of a logic analyser for measuring timing is used. This does make it simpler (in my opinion) but it is not essential to have these tools to hand. There are also software timing techniques provided by the development tools. It may also be possible for you to achieve your goals simply by looking at the issues raised in this document.

History

Rev 1.0	First written

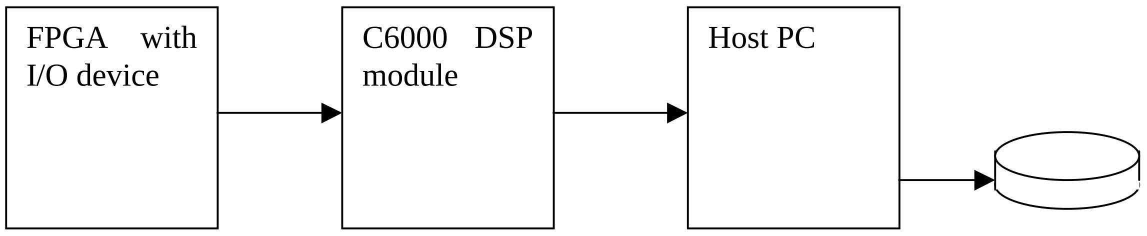
Background

This document is discussing the techniques that you can use to make your system operate at high speeds and in real time. Obviously before you can attempt to do that you must have learnt how to use the system and development tools. This document assumes that you already understand those things and have worked through the getting started tutorials and some basic examples that relate to your hardware. As such we will not discuss the details of each command needed to make each developmental step.

If you have not worked through those getting started tutorials and examples then do so before you read this document.

System architecture

The system that we will attempt to make In this tutorial is shown below:-



The input device will deliver samples at a constant rate and may perform some processing on those samples with the FPGA before sending the data to the C6000. This device cannot be stopped at any time because it is sampling real time data.

The C6000 will receive the data, and process it. The C6000 is really designed for processing the data in blocks, while it can receive and send data using DMA engines. Data will then be sent from the C6000 to the PC, where it will be stored on disk.

FPGA to C6000

The system will be dealing with data that comes from the real world. To begin with we will replace that data with a simple count so that we can check it’s correctness as it passes through the system.

I chose to use an external clock to generate the data samples. Then I could easily change the data rate each time I wanted to establish the limit of performance of the system.

The problem faced by all data samplers is that they cannot be stopped, so they will continue to generate samples at a fixed rate even if another part of the system cannot keep up. The architecture of a HERON/HEART system is that there is a FIFO used for the connection between the modules. This can absorb some differences in flow, but can never fix the situation where more samples are generated than can be processed.

The C6000 module will be programmed to read and process blocks of data. This will cause the flow of data to be “bursty” i.e. peaks and lulls will occur in the reading of the FIFO. Actually the transfer of blocks will be controlled by Interrupts that are issued to the processor core which must run some code to re-start the DMA. The response to those interrupts will not be constant, and the worst possible interrupt response time will govern the highest rate that samples can be received.

When an interrupt is issued, the FIFO is not being read, but samples are still being written to the FIFO by the acquisition device. So if we know the length of time that it will take to service the interrupt, and the length of the FIFO, we can work out the maximum sample rate that can be used. If we exceed that sample rate then we will attempt to write to a FIFO that is full and data will be lost. This is not acceptable in a real time system. The HEART FIFO length should not be relied upon, but is approximately 470 words at each node. As you can see from the above explanation the length of the FIFO can have an effect, so it is often a good idea to provide a further FIFO inside your FPGA design. Our standard examples often include a further 511 words of FIFO using a Core Generator FIFO block. Actually if you continue to extend the length of the FIFO you can reach a point where the limit of speed becomes something else and there is no need to extend the FIFO more. It is up to you to determine what the optimal FIFO length is for your system.

So we need an FPGA design that has a FIFO in it, that will be filled if samples are generated after the HERON FIFO becomes full. I used a 31 bit counter, which is clocked by the external clock I provide from a signal generator. If the FIFO inside the FPGA is not full, the value of the counter will be written into the FIFO, and the counter incremented on each clock. If the FIFO inside the FPGA is full when the clock occurs, the counter will be incremented and the previous value lost. This will allow us to detect if data has been lost.

Actually to be certain that if data is lost it is because of this FIFO overflowing, I use a registered version of the FULL signal to set bit 31 of the data written, on the first count written after the FIFO becomes not full.

The overflow bit is generated by :-

```
process(RESET, SCLK)
  begin
    if RESET='1' then
      OVERFLOW <= '0';
    elsif rising_edge(SCLK) then
      OVERFLOW <= ADCFIFO_FULL;
    end if;
  end process;
```

and the FIFO data is generated :-

```
DVALID <= RUN;
  ADC_FIFO(30 downto 0) <= std_logic_vector(ADCcount);
ADC_FIFO(31) <= OVERFLOW;

  process(RUN, SCLK)
  begin
    if RUN='0' then
      ADCcount <= (others=>'0');
    elsif rising_edge(SCLK) then
      ADCcount <= ADCcount + 1;
    end if;
  end process;
```

This means that if we get a sequence e.g. 0x00000608, 0x00000609, 0x00000700, 0x00000701 then the data lost is not due to the FIFO in the FPGA overflowing, but if we get 0x00000608, 0x00000609, 0x80000700, 0x00000701 then it is.

Notice that there is a signal in the above VHDL called “RUN”. This is to solve the issues that we can see at startup. If we are not careful then as soon as the FPGA is loaded, it will start to count and put data into the FIFO. Then when we reset the system, the FIFOs will be cleared and that data lost, the FIFO will again be filled until they are full. When the C6000 is loaded and run, the first values that it reads will be from just after the reset, then there will be lost data, before the true data starts. This makes it very difficult to detect true data loss without also gaining errors because of this startup problem.

The RUN signal is designed to cure that as follows :-

```
process(RST,START)
  begin
    if RST='1' then
      RUN <= '0';
    elsif rising_edge(START) then
      RUN <= '1';
    end if;
  end process;
```

where the signal “START” is actually connected to UMI0. So on reset the counter is cleared and will not run until a rising edge occurs on the UMI0 signal. This allows us to correctly prepare the C6000 for receiving data before releasing the FPGA to send the count, which will always start at the value 0.

As a further check for when the FIFO overflows I used one of the LEDs on the module to show when the internal FIFO has overflowed. If we simply connect the overflow signal to the LED it will not be possible to see overflows that have happened for just a few samples, so I used as counter to hold the LED on for some time.

I used the standard led.vhd from an IO4 example3 to do that for me. Then I could see the LED light up each time the FIFO is overflowed.

C6000 program

The C6000 program will use HERON-API to control the DMA engines, allowing the CPU to process data that has been received previously.

For that I started from a DSP example3 for a HERON-IO4. My main loop is below :-

```
 HeronUmi0_Out(HERON_UMI_LOGIC0);
loop=0;
count=-1;

while(1)
{

     /* start to fill buffer a*/
  status = HeronRead(rfifo,inputsa,SAMPLES);
  if (status != HERON_IO_IN_PROGRESS)
               {
               printf("error from read \n");
               exit(0);
               }

if (start==0)
  {
  /* process the input data buffer b here if it is not the first time */

HeronDigioOut(1);
  /* do your processing */
               for(i=0;i<SAMPLES;i++)
               {
                              if (count==0x7fffffff)count=-1;
                              if (inputsb[i]!=(count+1))
                              {
                                             printf("B   loop   %x   error   got   %x   when   count   was
%x\n",loop,inputsb[i],count);
                                             exit(0);
                              }
                              count=inputsb[i];
               }
  loop++;
HeronDigioOut(0);
  }
  else
  {
    HeronUmi0_Out(HERON_UMI_LOGIC1);
  start=0;

  }
  /* wait for buffer a to be filled*/
  status = HeronWaitIo(rfifo);
  if (status != HERON_OK)
  {
  printf("Cannot read fifo %d. Error %d,\n",INFIFONO,heronerr);
  exit(0);
  }
HeronDigioOut(0);
  /* start to fill buffer b */
                status = HeronRead(rfifo,inputsb,SAMPLES);
  if (status != HERON_IO_IN_PROGRESS)
                {
                printf("error from read \n");
                exit(0);
                }

  /* process the input data buffer a here */

HeronDigioOut(1);
  /* do your processing */
                for(i=0;i<SAMPLES;i++)
                {
                                if (count==0x7fffffff)count=-1;
                                if (inputsa[i]!=(count+1))
                                {
                                                printf("A  loop  %x  error  got  %x  when  count  was
%x\n",loop,inputsa[i],count);
                                                exit(0);
                                }

                                count=inputsa[i];
                }
  /* wait for buffer b to be filled*/
  HeronDigioOut(0);
  status = HeronWaitIo(rfifo);
  if (status != HERON_OK)
  {
  printf("Cannot read fifo %d. Error %d,\n",INFIFONO,heronerr);
  exit(0);
  }

HeronDigioOut((0));
}
```

Notice the use of “`HeronUmi0_Out(HERON_UMI_LOGIC1);”` to provide the rising edge on the UMI0 line.

I chose to use a dedicated DMA for the transfer because high performance was important to me. To do this I opened the device as “rd”. Refer to the HERON-API documentation if you want more details.

I made a new HERON-API project for my module type and compiled this program. When I run it using the reset plug in I could look at the LED on the FPGAS module to see if the FIFO was overflowing. If an error is detected the `exit(0);` will cause the DSP program to halt, and the FPGA FIFO will be permanently overflowed. Running this program I found that the maximum clock rate I could use was 5.5Mhz (22Mbytes/sec). This seems too low to me.

So this is where I use the logic analyser. If you notice the loop has `HeronDigioOut(1);` and `HeronDigioOut(0);` around the processing parts. This means if I look on a logic analyser (you could use an oscilloscope) when the DigOut0 line is high the C6000 is in the processing loop. This allowed me to see that the processing time was 188us for 1024 samples.

The immediate thing I can do to help that is to go to the Project Build Options as set optimisation to –o3. The default project has no optimisation set, so that the source code can be single stepped through using Code Composer. When –o3 is used the compiler actually combines lines of C code into the same assembler sequence. This makes it efficient but very difficult to logically debug. Actually I used the debugger to get the silly logical mistakes out of the code above. Now I know it works I can use the optimiser to get faster code.

With –o3 the processing loop is reduced to 81.5us for the 1024 samples.

In build options I selected -k option to keep the generated assembler files. Looking in that assembler file the C line "if (inputsa[i]!=(count+1))"

generates

```
  .line                    133
        LDW    .D2T2   *++B5,B12        ; |320|
        NOP           4
        SUB     .D2    B12,B10,B4       ; |320|
        CMPEQ   .L2    B4,1,B0          ; |320|
[!B0]   B       .S1    L31              ; |320|
        NOP           5
```

Notice that there is a NOP 4 and a NOP 5 in there – essentially 9 wasted cycles in each loop. Also there are no || symbols so in each clock only a single execution unit is active (the C6000 has 8).

To attempt to improve that I changed the loop to be

```
  if (inputsb[i]!=(count+1))
  {
                printf("B loop %x error got %x when count was %x\n",loop,inputsb[i],count);
                exit(0);
  }
  if (inputsb[i+1]!=(inputsb[i]+1))
  {
                printf("B loop %x error got %x when count was %x\n",loop,inputsb[i+1],inputsb[i]);
                exit(0);
  }
  if (inputsb[i+2]!=(inputsb[i+1]+1))
  {
                printf("B loop %x error got %x when count was %x\n",loop,inputsb[i+2],inputsb[i+1]);
                exit(0);
  }
  if (inputsb[i+3]!=(inputsb[i+2]+1))
  {
                printf("B loop %x error got %x when count was %x\n",loop,inputsb[i+3],inputsb[i+2]);
                exit(0);
  }

  count=inputsb[i+3];
```

This made more complicated assembler with less NOP instrunctions, and a few || symbols. Now the processing runs faster at 59.5 us. This is because the looping overhead is reduced, but we need to get more instructions to be run in parallel.

I changed the loop to be

```
  for(i=0;i<SAMPLES;i+=4)

{

  temp0=inputsb[i]-count;

  temp1=inputsb[i+1]-inputsb[i];

  temp2=inputsb[i+2]-inputsb[i+1];

  temp3=inputsb[i+3]-inputsb[i+2];
  if ((temp0!=1)||(temp1!=1)||(temp2!=1)||(temp3!=1))
  {
  printf("B loop %x error got %x %x %x %x when count was %x\n",loop,inputsb[i],inputsb[i+1],inputsb[i+2],inputsb[i+3],count);
  exit(0);
  }
  count=inputsb[i+3];
  }
```

Then the assembler became

```
   .line          82
         LDW    .D1T2  *A0,B4          ; |269|
         LDW    .D1T2  *+A0(4),B6      ; |269|
         LDW    .D1T2  *+A0(8),B7      ; |269|
         LDW    .D1T2  *+A0(12),B5     ; |269|
         NOP           1
         SUB    .D2    B4,B10,B8       ; |269|

         SUB    .D2    B6,B4,B8        ; |269|
||       CMPEQ  .L2    B8,1,B9         ; |269|

         SUB    .D2    B7,B6,B0        ; |269|
||       XOR    .S2    1,B9,B8         ; |269|
||       CMPEQ  .L2    B8,1,B9         ; |269|

         SUB    .D2    B5,B7,B0        ; |269|
||       CMPEQ  .L2    B0,1,B1         ; |269|
||       XOR    .S2    1,B9,B9         ; |269|

         XOR    .S2    1,B1,B9         ; |269|
||       CMPEQ  .L1X   B0,1,A4         ; |269|
||       OR     .L2    B9,B8,B8        ; |269|

         XOR    .S1    1,A4,A4         ; |269|
||       OR     .S2    B9,B8,B8        ; |269|

         OR     .S1X   A4,B8,A1        ; |269|
  [ A1]  B      .S1    L13             ; |269|
         NOP           5
```

for all four samples. Notice that the number of NOPS is greatly reduced, and there are several groups of three parallel instructions

This loop runs in 22.5us when timed by the digital I/Os.

Going further and making the loop be a loop of 8 changes the assembler to be

```
   .line          86
         LDW    .D1T2  *A4,B0          ; |273|
         LDW    .D1T2  *+A4(4),B4      ; |273|
         LDW    .D1T2  *+A4(8),B5      ; |273|
         LDW    .D1T2  *+A4(12),B9     ; |273|
         LDW    .D1T2  *+A4(16),B8     ; |273|

         SUB    .D2    B0,B10,B6       ; |273|
||       LDW    .D1T2  *+A4(20),B7     ; |273|

         LDW    .D1T2  *+A4(24),B6     ; |273|
||       SUB    .D2    B4,B0,B1        ; |273|
||       CMPEQ  .L2    B6,1,B2         ; |273|
        SUB     .D2     B5,B4,B2         ; |273|
||      CMPEQ   .L2     B1,1,B1          ; |273|
||      XOR     .S2     1,B2,B3          ; |273|

        SUB     .D2     B9,B5,B1         ; |273|
||      CMPEQ   .L2     B2,1,B2          ; |273|
||      XOR     .S2     1,B1,B12         ; |273|
||      LDW     .D1T1   *+A4(28),A0

        SUB     .D2     B8,B9,B3         ; |273|
||      OR      .S2     B12,B3,B2        ; |273|
||      XOR     .L2     1,B2,B1          ; |273|
||      CMPEQ   .L1X    B1,1,A5          ; |273|

        SUB     .D2     B7,B8,B3         ; |273|
||      CMPEQ   .L2     B3,1,B2          ; |273|
||      OR      .S2     B1,B2,B1         ; |273|
||      XOR     .S1     1,A5,A5          ; |273|

        SUB     .D2     B6,B7,B3         ; |273|
||      CMPEQ   .L2     B3,1,B1          ; |273|
||      XOR     .S2     1,B2,B2          ; |273|
||      OR      .S1X    A5,B1,A5         ; |273|

        CMPEQ   .L2     B3,1,B2          ; |273|
||      OR      .S1X    B2,A5,A5         ; |273|

        SUB     .L2X    A0,B6,B1
||      XOR     .S2     1,B1,B3          ; |273|

        CMPEQ   .L2     B1,1,B1          ; |273|
||      OR      .S1X    B3,A5,A5         ; |273|
||      XOR     .S2     1,B2,B2          ; |273|

        OR      .S1X    B2,A5,A5         ; |273|
||      XOR     .S2     1,B1,B1          ; |273|

        OR      .S1X    B1,A5,A1         ; |273|
  [ A1] B       .S1     L13              ; |273|
        NOP             5
        ; BRANCH OCCURS                  ; |273|
```

which has rolled some of the LDW into parallel instructions and now we have sometimes 4 instructions in parallel

This processing now takes 14.5us when measured with the digital I/Os..

I was doing this only on the first processing loop, when I copied the same code structure into the second processing I found that the compiler didn't give the same assembly.

So I searched for a way to make the compiler optimise both loops.

Eventually I found that changing the loop to

```
        if (count==0x7fffffff)count=-1;
        for(i=0;i<SAMPLES;i++)
        {
                if (inputsb[i]!=(count+1))
                {
                        errors1++;
                }
                count=inputsb[i];
        }
```

```
if (errors1!=0)
{
printf("B error got %x errors in loop %x\n",errors1,loop);
exit(0);
}
```

Gave me a comment in the assembler file of

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                : 369
;*      Loop opening brace source line : 370
;*      Loop closing brace source line : 376
;*      Loop Unroll Multiple             : 4x
;*      Known Minimum Trip Count         : 256
;*      Known Maximum Trip Count         : 256
;*      Known Max Trip Count Factor      : 256
;*      Loop Carried Dependency Bound(^) : 6
;*      Unpartitioned Resource Bound     : 5
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*                                 A-side   B-side
;*      .L units                      3        4
;*      .S units                      0        1
;*      .D units                      4        0
;*      .M units                      0        0
;*      .X cross paths                1        2
;*      .T address paths              2        2
;*      Long read paths               0        0
;*      Long write paths              0        0
;*      Logical  ops (.LS)            2        2     (.L or .S unit)
;*      Addition ops (.LSD)           6        8     (.L or .S or .D unit)
;*      Bound(.L .S .LS)              3        4
;*      Bound(.L .S .D .LS .LSD)      5*       5*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Schedule found with 4 iterations in parallel
;*      done
;*
;*      Loop is interruptible
;*      Collapsed epilog stages     : 3
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 12 bytes
;*      Minimum threshold value     : -mh56
;*
;*      Minimum safe trip count     : 1 (after unrolling)
;*----------------------------------------------------------------------------*
```

This shows that the compiler has recognised the loop and has really optimised it.

In previous files that comment had been

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*      Disqualified loop: bad loop structure
;*----------------------------------------------------------------------------*
```

It seems that the Compiler didn’t like the loop to contain the exit statement, and incrementing an error counter was found easier.

This loop ran in 5us for processing the 1024 samples.

Now I was able to increase the "sample" rate to 16.5Mhz (66Mbytes/sec) with 1 timeslot,

33Mhz (132Mbytes/sec) with 2 timeslots . With 3 timeslots it will run until the sample rate is 49Mhz.

Using 4 timeslots this increases to 56Mhz = 224Mbytes/sec.

The failure at rates faster than this is due to the FIFO read speed on the HERON2 and not the processing time. I can observe that on the logic analyser because there are still gaps between the processing loops.

From the above exercise we can see how even at quite low sample rates the C6000 needs to be running optimised code to keep up. What we are trying to do in terms of processing is not complicated, but we had to search for a good solution so that the compiler could make a good job. You can see how I managed to search for solutions even if the final solution was not so far away from my original code.

C6000 internal memory vs external memory

The C6000 has a limited amount of internal memory, and a larger amount or external memory. Which memory area is used for storage is controlled by the .cdb file used.

A standard HUNT ENGINEERING project generated by the plug in, assigns the memory areas in a general purpose way. You can change them but here I will discuss the "standard" settings.

A declaration inside a function (such as maintask) will be taken from the stack. Each task has it's own stack which is assigned a size and a memory segment using TSK properties. This is normally IDRAM i.e. the variables defined inside the function associated with a task will be in internal memory.

A global declaration made at the top of your source file, outside of a function will be assigned to the .data section. You can choose which memory segment is used for this under MEM and compiler sections in the cdb file. Normally this is also internal data memory.

So if we use the assignment

int datastore[SAMPLES];

either in the task or at the top of the source file then that array will be in internal memory. It is easy to check, you can either print out the value of the pointer "datastore" or after loading the program in Code Composer put your cursor over the source code of the declaration and CCS will pop up a box like datastore=(int *)0x80000054 .

The measurements made so far use these internal memory buffers, but the limited amount of internal memory makes it unsuitable for applications that need to buffer a large amount of data, like we want to.

So we can remove the definition of datastore, and replace it with

```
int *datastore;

datastore=malloc(BUFSIZ*sizeof(int));

if (datastore==NULL)
{
printf("error malloc failed\n");
exit(0);
}
```

This creates the data buffer in the heap section defined by "segment for Malloc()/Free()" in the cdb file under the MEM section. This is normally external memory.

Simply making this change to our program and re-running it shows (using the logic analyser) that the time for processing is now 88.5us.

Looking in the assembler file we see that the loop still has the comment :-

```
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Schedule found with 3 iterations in parallel
;*      done
```

```
;*
;*      Loop is interruptible
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*      Collapsed prolog stages     : 2
;*      Minimum required memory pad : 16 bytes
;*
;*      Minimum safe trip count     : 1 (after unrolling)
```

So the compiler has still recognised the loop and optimised it. The accesses to external memory are what is slowing down the execution, but why by so much?

Well the external memory of a C6000 is burst memory, which means that there is a pipeline of accesses. When a burst is in progress one data item is transferred in each clock, but for the first data item there is a delay of around ten cycles. These cycles are at half of the processor clock too, so the processor is stalled for many clocks waiting for the data access to complete. There are further uncertainties, such as bus turnaround delays, i.e. if the accesses are read,write,read,write it will be less efficient than read,read,write write. For details refer to the TI documentation—either the Peripherals guide or the “CPU and DMA data access” guide (both on the HUNT CD).

Actually the CPU can never cause a burst from memory, so all accesses by the CPU to external memory will be inefficient. Thus it is necessary to use a DMA for an efficient transfer.

HERON-API uses the processor DMAs, and assumes that it can use any of them. If you want to use a DMA in your own program it is necessary to claim one from HERON-API. The function

```
dma=HeronDmaClaim();
if (dma==-1)
{
 printf("error from dma claim\n");
 exit(0);
}
```

properly claims the DMA.

If you look under the “general examples and utilities” for the C6000 there is an example for using a DMA to copy a block of data. I simply cut and pasted the functions from there into my code and changed the processing loop to be :-

```
 dma_start_copy(dma,(unsigned int)inbuffers[nextproc],(unsigned int)copyofdata,SAMPLES);

 dma_wait_end(dma);
 if (count==0x7fffffff)count=-1;
 for(i=0;i<SAMPLES;i++)
 {
             if ((copyofdata[i]-count)!=1)
             {
                         errors1++;
             }
             count=copyofdata[i];
 }
```

So the processing loop is now a DMA copy to internal memory followed by processing the data in internal memory. This uses the temporary buffer copyofdata which is declared in the function so is in internal memory.

The time for this loop when I first tried it was 27.8us.

Investigating what proportion of that time was for the DMA and what was for the processing (using a logic analyser and digital outputs) showed that the DMA took only 8.8us, and that the processing had increased to 19us.

Looking in the assembler file we can see why because it says :-

```
;*      Searching for software pipeline schedule at ...
;*         ii = 7  Did not find schedule
;*         ii = 8  Cannot allocate machine registers
;*                 Regs Live Always   :  8/2  (A/B-side)
;*                 Max Regs Live      : 17/10
;*                 Max Cond Regs Live :  0/3
;*         ii = 9  Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 9  Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 9  Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 10 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 10 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 10 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 11 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 11 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 11 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 12 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 12 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 12 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 13 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 13 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 13 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 14 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 14 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 14 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*         ii = 15 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*        ii = 15 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*        ii = 15 Cannot allocate machine registers
;*              Regs Live Always   :  8/2  (A/B-side)
;*              Max Regs Live      : 17/10
;*              Max Cond Regs Live :  0/3
;*      Disqualified loop: did not find schedule
;*----------------------------------------------------------------------------*
```

I do not know what has changed but clearly the compiler is now confused. I tried a number of things but eventually found that adding the compiler option –mh100 which is "speculate threshold = 100" that the assembler listing showed

```
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Schedule found with 4 iterations in parallel
;*      done
;*
;*      Loop is interruptible
;*      Collapsed epilog stages     : 3
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 12 bytes
;*      Minimum threshold value     : -mh56
;*
;*      Minimum safe trip count     : 1 (after unrolling)
;*----------------------------------------------------------------------------*
```

So it seems that I could use 56, but 100 has done the trick.
Now the combination of DMA and processing becomes 14us, just 15% of the time taken to process data directly in the external memory!
This loop can run with the clock rate of 33Mhz (132Mbytes/sec).

Next in preparation for using Tasks I changed my code from using HeronWaitIO to use the semaphore model for HERON-API. This reduced the speed of operation a little to a sample rate of 31Msps (124Mbytes/sec). This is due to the overhead of using DSP/BIOS semaphores.

Sending data to the PC

The PCI bus has a maximum data rate of 132Mbytes/sec. Just as is the case with the C6000, the PC will use interrupts to signal the end of a transfer so that the PC processor can start the next one. This might actually be the device drivers or might be the user code.
When a PC is running Windows there is no guaranteed interrupt response time. Microsoft documentation indicates that interrupt response times can be the order of 10 to 100ms. Clearly this is not a real time operating system.
Just as in the case of the C6000 while the interrupt is being serviced, the PCI bus will not be transferring any data. This can cause data to be lost in the system unless the data is buffered during that time.
The external memory of the C6000 is ideal for buffering that data, so I changed the DSP code to have one task that received data from the FPGA :-

```
while(1)
{

     /* start to fill buffer a*/
  status = HeronRead(rfifo,inbuffers[nextinput],SAMPLES);
  if (status != HERON_IO_IN_PROGRESS)
               {
               printf("error from read \n");
               exit(0);
               }
  if (start==1)
  {
               /* need to signal the start to the IO4 */
               HeronUmi0_Out(HERON_UMI_LOGIC1);
               start=0;
  }
  else
  {
  SEM_post(&buffer_received);             /* this will increment the sem by 1*/
  }


  if ((nextinput==0) & (start==0))
  {
               wrap=1;      /* have filled the last, now inputting to buffer 0*/
  }

  if (nextinput==NO_OF_BUFFS-1)
  {
      nextinput=0;
  }
  else
  {
         nextinput++;
     }


     if ((nextinput==nextoutput-1) || ((nextinput==(NO_OF_BUFFS-1))&(nextoutput==0)))
     /* we are now filling the buffer that is being sent */
     {

         printf("buffers were overrun \n");
               exit(-1);
    }

  SEM_pend(&read_complete,SYS_FOREVER);
}
}
```

And added a processing task that was :-

```
void proctask(void)
{


int count=-1;
int errors1=0;
int i;


while(1)
{
/* wait to be told there's a buffer ready */

  SEM_pend(&buffer_received,SYS_FOREVER); /* the semaphore will decrement when we execute this*/

               /* so if there are several, the remainder will be remembered*/

/* copy the next buffer into internal memory for processing*/
HeronDigioOut(0x1);
  dma_start_copy(dma,(unsigned int)inbuffers[nextproc],(unsigned int)copyofdata,SAMPLES);

  dma_wait_end(dma);
  HeronDigioOut(0x3);
  if (count==0x7fffffff)count=-1;
  for(i=0;i<SAMPLES;i++)
  {
               if ((copyofdata[i]-count)!=1)
               {
                            errors1++;
               }
               count=copyofdata[i];
  }
  if (errors1!=0)
  {
               printf("B error got %x errors in buf %x\n",errors1,nextproc);
               exit(0);
  }

/* update the buffer we are doing*/

  if (nextproc==NO_OF_BUFFS-1)
  {
  nextproc=0;
  }
  else
  {
  nextproc++;
  }
HeronDigioOut(0);
SEM_post(&buffer_processed);
}
}
```

And finally an output task

```
void writetask(void)
{
int status;
  while(1)
  {
  /* wait to be told there's a buffer that has been processed */

               SEM_pend(&buffer_processed,SYS_FOREVER); /* the semaphore will decrement when we
  execute this*/

                                        /* so if there are several, the remainder will be
  remembered*/

  /* copy the next buffer to the host PC*/

               status = HeronWrite(wfifo,inbuffers[nextoutput],SAMPLES);
               if (status != HERON_IO_IN_PROGRESS)
                            {
                            printf("error from read \n");
                            exit(0);
                            }

  /* update the buffer we are doing*/

               if (nextoutput==NO_OF_BUFFS-1)
               {
               nextoutput=0;
               }
               else
               {
               nextoutput++;
               }

   HeronUmi2_Out(HERON_UMI_LOGIC1);
  /* wait until the PC has accepted the buffer*/
               SEM_pend(&write_complete,SYS_FOREVER);
                              HeronUmi2_Out(HERON_UMI_LOGIC0);
  }
  }
```

Together these three task manage a group of buffers in the external memory of the C6000, so that the capture and processing of data will continue even while the PC is not receiving buffers. Then when the PC is responding again it can catch up with the capture and processing again. Without using this technique the throughput will be governed by the worst possible interrupt response from your PC.
In my case I used a fairly old PC (AMD K6-450) running Win98 and wrote the host program to look very similar to the DSP program ie.

```
while(1)
{
     /* start to fill buffer a*/
  Status = HeRead(hDevice, data1, (SAMPLES)*sizeof(HE_DWORD), ReadIoStatus);


if (start==0)
```

```
 {
               /* process the input data buffer b here if it is not the first time */

 /* do your processing */

 }
 else
 {
 start=0;
 }

 /* wait for buffer a to be filled*/
 HeWaitForIo(hDevice, ReadIoStatus);

 /* start to fill buffer b */
 Status = HeRead(hDevice, data2, (SAMPLES)*sizeof(HE_DWORD), ReadIoStatus);

 /* process the input data buffer a here */

 /* do your processing */


 /* wait for buffer b to be filled*/

 HeWaitForIo(hDevice, ReadIoStatus);

     }
```

I loaded the DSP program using Code Composer, started the Host program and then told Code Composer to run.
The system would now run at only 4Mhz (16Mbytes/sec). This was disappointing to me so I looked at the FIFO signals to see what was happening. I used the HERON module specification to find the correct module pins to look at with my Logic analyser. It showed that there were huge periods where the FIFO was transferring nothing at all, and they were more often than I would have expected for the PC interrupt delays.
I decided that the only other thing that was running was Code Composer which was actually using the same PCI interface too.
So it was time to try and run the same system without using Code Composer. That meant making the DSP project again as a Server/Loader project. Actually I wanted nothing to disturb the transfer at all, so I removed all of the printf calls too, replacing them with either digios or UMI accesses.
Then I followed the Combined Server/Loader and API example and put my host code into the DoAPI function.
This allowed me to run at 9Mhz (36Mbytes/sec) so it clearly was Code Composer that was interfering with my transfers.

I added processing into my Host code to check the count was still correct. When I did this I realised that I must add a synchronisation between the PC program and the C6000 program. To do this I used a FIFO message changing the DSP loop to have

```
if (start==1)
 {
 /* wait for host to be ready*/
 status = HeronReadWord(hrfifo,&dummy);
 /* using the readword function will make this task poll until it receives the word*/


               /* need to signal the start to the IO4 */
               HeronUmi0_Out(HERON_UMI_LOGIC1);
               start=0;
 }
 else
```

and the host code to have

```
if (start==0)
 {
               /* process the input data buffer b here if it is not the first time */

 /* do your processing */
```

```
     if (count==0x7fffffff)count=-1;
               for(i=0;i<SAMPLES;i++)
               {
                         if((data2[i]-count)!=1)
                         {
                                   printf("error got %x at %x\n",data2[i],count);
                                   exit(0);
                         }
                         count=data2[i];
               }


  }
  else
  {
/* signal to the DSP that the host is ready*/
  Status = HeWrite(hDevice, data1, sizeof(HE_DWORD), WriteIoStatus);
//             HeWaitForIo(hDevice, WriteIoStatus);
  start=0;
               }
```

This loop wouldn't run reliably at all, so I searched for the optimisation options in MSVCC and set that to maximum.
Now my loop would run at up to 7Mhz (28Mbytes/sec), showing that even the PC has trouble processing at these data rates. Actually causing the PC to access a disk or do some work makes this system fail at this rate too.
Finally I replaced my checking on the PC with a file write of the data buffer. This would not run at all reliably at any data rate.

So I moved to a more modern PC, which was a P4 2.4Ghz running Windows 2K. Because of the Host API I was able to simply run the same executables despite the different operating system.
Here I measured :-
No processing 15Mhz, i.e. 60Mbytes/sec
Checking of the counter 15Mhz again.
Writing the file 5.5Mhz, i.e. 22Mbytes/sec

I suspect that the disk rate could be increased if we used a circular buffer and a separate task on the PC just like I did on the DSP, but for the purposes of this example I have a complete system and have discovered a number of items that it is necessary to be concerned about.

Conclusion

At each and every stage it was necessary to check and improve every aspect, but we can summarise some important things to remember below :-

1. Even at slow sample rates the C6000 processing speed can become an issue.
2. Optimisation of your C6000 code is very important and although there are no fixed rules this document shows some general techniques for making a difference and trying to determine what that difference is.
3. Adding FIFOs to your FPGA designs can be a good idea to increase the system throughput.
4. When sending data to a host PC the data flow will not be constant but bursty. This means it is impossible to achieve high data rates unless you have a suitable buffer to absorb the data while the PC is not responding.
5. Using Code Composer will affect the PCI bus activity and can reduce your system performance considerably.
6. Other activity on your PC will also affect the PCI bandwidth.
7. The speed of the PC will (obviously) affect the speed of your system if the PC is used as a node in that system.

So this example shows that high system performances can be achieved, but you must take care of many things along the way. Hopefully this document helps you to do that.