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

Programming HERON2 flash memory.

Rev 1.1 P.Warnes 7-5-03

This example shows how the flash memory on a HERON2 module can be programmed. With the example program provided it is possible to store a TI "**.out*" file into flash memory. There are very few limitations on the "**.out*" file you wish to store in flash memory.

You need Code Composer Studio to run this example. The flash memory on any HERON2 module in the system can be programmed (individually) using Code Composer Studio.

Example software

The example software supplied consists of a number of "*.*out*" files. First, there is a program that will perform the actual flash programming. This program ("*flaspro2. out"* for HERON2 modules) is loaded into Code Composer and is then run as you would run any 'C6x application. When run, the program will ask you for two files. The first file is a flash rom boot-loader, which is provided by us, the second file is the application you want to store into flash memory.

In order to load the program it is important that the EMIF of the processor is initialised correctly, and that the cache is off. To enable you to achieve this regardless of what has previously been run we provide the heron2init.gel file

For HERON2 modules with a 'C6203 processor, the rom boot-loader file is *"**r2v1.out**"*. When "*flaspro2.out*" asks you for the first file, you need to provide the filename and path to this file.

The source code of "*flaspro2.out*" is included. It is available in the "*flaspro2*" sub-directory, together with a linker command file ("*flaspro2.cmd*") and a compile/link batch file ("*flaspro2.bat*"). Other files in this directory hold the standard TI COFF file reader routines.

The source code of *"r2v1.out"* is included. It is available in the "*rom2v1"* sub-directory, together with a linker command file ("*rom2v1.cmd*") and a build batch file ("*rmake.bat*").

The flash programming process will overwrite the standard HUNT ENGINEERING boot-loader code that existed in the flash memory before you programmed it.

On HERON2 modules, if you have programmed your own application and wish to return to the standard HUNT ENGINEERING boot-loader, use the *"r2v1.out"* file. Simply load and run "*flashpro2.out*" with Code Composer and when asked by "*flashpro2.out*" what file to store, reply *"r2v1.out"*. When "*flaspro2.out*" asks for the second file, don't specify a filename (make sure the edit box is empty) and then type return.

When you do need to re-program the original boot-loader, please note that in certain cases external memory may not have been properly initialised. In that case, run *"flaspro2.out"* twice. The first time it will initialise external memory (and will apparently crash or complain there's not enough memory), the second time the C initialisation routines can properly set up the heap ("*.sysmem*") in external memory.

How to prepare your application to run from flash memory

There are few limitations on the application you want to store in flash memory. Simply use the "**.out*" file of the application that you developed using Code Composer Studio. You also don't need to worry about initialisation as this is all done in the boot-loader code.

The few limitations that exist have to do with allowing "*r2v1.out*" some space. Your "**.out*" file should not use the area between address 0x400 and address 0x1000. This is where the boot-loader code is located. You should also not use the area between 0x8000FC00 and 0x80010000 for initialised (data) sections. There is no problem using this space for un-initialised (data) sections, such as heap ("*.sysmem*") or stack ("*.stack*"). In Appendix B it is explained in more detail how to verify that your application doesn't use the address regions mentioned.

Since there's only 2Mbytes of flash memory, your program should not be bigger than 2Mbytes. To find out your application's size, look in the "**.map*" file for your application, and sum all initialised data sections and all code and vector sections. (In Appendix A it is explained in more detail how to generate a map file.) Uninitialised data sections such as heap and stack will be set up by C initialisation code and are not stored in flash memory.

How to run the example and program the flash memory

Start up Code Composer. If you have more than 1 processor, then you will see the Parallel Debug Manager window. In this case, open the window of the processor whose flash memory you wish to program.

First use File→ Load GEL and select the heron2init.gel file. Now select the GEL → heron2init → Setup_HERON2 menu item and click your mouse. This will initialise the EMIF and cache ready to load the flash programming utility.

Do a File → Load Program and select *"flaspro2.out"*. Next, do a Debug → Run (F5). You should now see in the "*Stdout*" window: "*Enter boot loader COFF filename*". A window will appear saying "*Standard Input Dialog Box*". Enter the name of the supplied *"r2v1.out"*. Click "*OK*". The program will show that it found 3 sections, "*.vectors*", "*.text*" and "*.size*". The program will then ask you "*Enter application COFF filename:*". Enter the path and name of your application ("*.*out*"), and click "*OK*".

The program will print out several messages that show what sections it found in your application that are candidates to be stored in flash memory. This can be a time-consuming process, if your application is not very small. Please be patient. The file read instructions are all executed via the JTAG connection, which is slow. After reading the file, *"flaspro2.out"* will actually program the flash memory. Progress messages will be shown.

When the programming has finished, you will see "*Finished*" displayed in the "*Stdout*" window. You can now move on to the next processor if you have more processors' flash memory to program. When all processors' flash memory has been programmed, you can start using/testing the system.

Verification and Debug

You can verify that something got written into flash memory by having a look at flash memory using Code Composer Studio. Do a View → Memory, then at *address* type "*0x01400000*". The first 0x2000 bytes are the *"r2v1.out"* program supplied by us. Everything above 0x01402000 is your application program. The application program is stored in consecutive blocks, and you should see a "magic" number (0xdeafd00f) at 0x1402008.

To check that your program is booted, you could do the following. Quit Code Composer Studio. Now reset the JTAG (Start → Programs → HUNT ENGINEERING → API JTAG reset) and the board (Start → Programs → HUNT ENGINEERING → API board reset). Re-start Code Composer Studio. If you have more than 1 processor in your system, open the window of the processor you're working on. Load the symbols of the program you stored in flash memory (File → Load Symbol). Do a Run (Debug → Run), then halt (Debug → Halt). Depending on where the execution is halted you will see C code or assembly. If in assembly, do a number of single steps or step outs to arrive at C code. (Obviously, you will only see C code when you compiled your application with debug information on (-g switch).)

Troubleshooting

Question: I'm trying to load *"flaspro2.out"*, but Code Composer Studio gives me messages that say "Data verification failed at address 0x….", where …. is a number between 0 and 0x10000.

Answer: This is because the cache is switched on. Probably you have not run the GEL file to initialise the EMIF and Cache. If you have, quit Code Composer Studio. First, reset the JTAG (Start → Programs → HUNT ENGINEERING → API JTAG reset), then reset the board (Start → Programs → HUNT ENGINEERING → API board reset). Re-start Code Composer Studio and try again.

Question: I'm trying to load an "*.*out*" file, but Code Composer Studio gives me messages that say "Data verification failed at address 0x…", where … is an address in SBSRAM (0x00400000 & higher) or SDRAM (0x03000000 & higher).

Answer: This is because the external memory is not configured properly (EMIF). Probably you have not run the GEL file to initialise the EMIF and Cache. If you have, quit Code Composer Studio. First, reset the JTAG (Start → Programs → HUNT ENGINEERING → API JTAG reset), then reset the board (Start → Programs → HUNT ENGINEERING → API board reset). Re-start Code Composer Studio and try again.

Question: I've run the *"flaspro2.out"* program, but the flash memory doesn't get written to.

Answer: Most probably you have not removed the flash memory write protect jumper. Please refer to the HERON2 manual, which is on the HUNT ENGINEERING CD in the \manuals directory, on which jumper to change. After you programmed the flash, and it works as intended, you can change the jumper again to write-protect your application.

Question: I'm running *"flaspro2.out"*, but when I do a Go Main or Single Step I don't see C code?

Answer: That's because the source code for *"flaspro2.out"* is in a different directory. If you wish to step through *"flaspro2.out"*, or the boot-loaders, compile/link into their own sub-directory, and when doing the Load Program, load the "*.*out*" from the sub-directory.

Question: Can I store DSP/BIOS programs in flash memory?

Answer: Yes, no problem. You can even use <std.h> routines, such as "*printf*". Routines such as "*printf*" will not block when Code Composer Studio isn't running, but simply proceed. Once you start Code Composer Studio, "*printf*" calls will display in the "*Stdout*" window (but not "*printf*" calls executed before). Similarly, functions such as "*fread*" and "*fwrite*" will not block, and appear to execute, but don't do anything, when Code Composer Studio is not running.

Question: Can I store programs that use the HERON-API in flash memory?

Answer: Yes.

Question: Can I store Server/Loader programs in flash memory?

Answer: No.

Question: Why not?

Answer: Because the Server/Loader uses blocking functions (e.g. "bootloader()", "printf", "fread()") that expect to be able to communicate with the host. They will loop idly until the communication has fully completed.

The boot process

The ‘C6x processor after being reset will copy the lowest 64 Kbytes from flash memory into program memory. Execution will then start at address 0. It is the *“r2v1.out”* code that will start to execute. This code will look in flash memory off address 0x01402000 for your application program.

Your application is stored in flash memory in blocks. Every block starts with the destination address (4 bytes), then follows a size (4 bytes), then a magic number (4 bytes), finally followed by data. At the end of the series of blocks follows the entry point of your application.

The magic number is used to verify that the flash rom was programmed properly. If any magic number is missing, the boot-loader will notice this and then behave as the standard boot-loader. So if anything goes wrong during flash memory programming, then the module is likely to continue to work as before programming the flash memory.

The “*r2v1.out*” code also initialises your HERON2 module, so that external memory is configured before your application is run. The “*r2v1.out*” code will try to read a boot stream from flash memory, but if there’s no such boot stream or if the flash memory data is corrupted, the boot-loader will try to read a boot stream from a FIFO.

After a reset, the cache is not switched on by the boot-loader and addresses 0 to 0x10000 are simply configured as internal program memory. If you wish to use the cache you will have to switch it on in your application. You have to make sure, though, that your program is running from external memory when you do that; also keep in mind that any code remaining in internal program memory will get “lost” when the cache is switched on.

In the case of the standard HUNT ENGINEERING boot loader, after execution has started from address 0, the program will simply loop and wait for data to come in over one of the FIFO’s. Using a simple protocol data is read from the FIFO and stored in memory. After all data has been read and stored, the program will jump to the entry point of the program stored in memory. The entry point will have been initialised using the simple protocol mentioned, and was part of the data stream over the FIFO.

Appendix A How to generate a "**.map*" file

If you are using a batch file to compile and link your application, a "**.map*" file is generated using the "*-m*" option followed by a filename of the map file. For example, a batch file may look like:

```
cl6x a.c -k -g -o2 -mw -mv6201 -z -ar -c a.cmd -l rts6201.lib -o a.out -m a.map
```

The last part, "*-m a.map*", will generate a map file with the name "*a.map*".

If you are using Code Composer Studio to compile and link your application, open Project → Options in the processor window whose project you wish to add a map file to. The "*Build Options*" window will appear, click the "*Linker*" tab. In the field named "*Map filename*" write the name of your map file, for example, "*a.map*". Click "*OK*". If you re-link your application you should find a map file "*a.map*" in your project directory.

An example map file is shown below, it is taken from the "*flaspro1*" directory.

```
******************************************************************
          TMS320C6x COFF Linker Version 3.01
******************************************************************
>> Linked Tue May 30 11:21:50 2000

OUTPUT FILE NAME:   <flaspro1.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 0000e2a0

MEMORY CONFIGURATION

         name      origin    length     used      attributes    fill
         --------  --------  ---------  --------  ----------  --------
         VECS      00000000  000000400  00000400     RWIX
         PMEM      00001000  00000f000  0000d7d4     RWIX
         SBSRAM    00400000  000040000  00000000     RWIX
         SDRAM     03000000  000fffc00  00800000     RWIX
         BMEM      80000000  000010000  000094ab     RWIX

SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.vectors   0    00000000    00000400
                00000000    00000400     flaspro1.obj (.vectors)
.text      0    00001000    0000d6a0
                …
                0000e680    00000020     remove.obj (.text)
.cinit     0    80000000    00000464
                …
                8000045c    00000008     --HOLE-- [fill = 00000000]
.const     0    80000464    00000583
                …
                800009e5    00000002     fputs.obj (.const)
.tables    0    80000000    00000000     UNINITIALIZED
.data      0    80000000    00000000     UNINITIALIZED
                …
                80000000    00000000     cload.obj (.data)
.stack     0    800009e8    00008000     UNINITIALIZED
                …
                800009e8    00000000     rts6201.lib : boot.obj (.stack)
.bss       0    800089e8    000001b8     UNINITIALIZED
                …
                80008b98    00000008     flaspro1.obj (.bss:c)
.sysmem    0    03000000    00800000     UNINITIALIZED
                …
                03000000    00000000     rts6201.lib : sysmem.obj (.sysmem)
.cio       0    80008ba0    00000120     UNINITIALIZED
```

```
                …
                80008ba0    00000120     rts6201.lib : trgmsg.obj (.cio)
.far       0    80008cc0    000007ec     UNINITIALIZED
                …
                800094a4    00000008                 : memory.obj (.far)
.memtab    0    80000000    00000000     UNINITIALIZED
.switch    0    0000e6a0    00000134
                …
                0000e6a0    00000134     cload.obj (.switch)

GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name
address    name
--------   ----
800089e8   $bss
…
0000e6a0   etext
ffffffff   pinit

GLOBAL SYMBOLS: SORTED BY Symbol Address
address    name
--------   ----
1000       .text
…
800094a4   __memory_size
ffffffff   pinit
ffffffff   ___pinit__

[188 symbols]
```

From this map file, we can see that 5 sections are initialised, and therefore they should be stored in flash memory. They are: "*.vectors*", "*.text*", "*.cinit*", "*.const*" and "*.switch*". All other sections are UN-INITIALIZED and don't need to be stored in flash memory. These sections will be initialised at run-time by the C initialisation routines. To find out exactly how much space is needed for this program, realise that the program "*flaspro1.out*" will store initialised sections in blocks of at most 0x4000 bytes. Per block it uses 12 bytes for block information (address, size, magic number). So we get:

Section	size	# blocks	flash size
.vectors	0x0400	1 block	0x040c
.text	0xd6a0	3 blocks	0xd6c4
.cinit	0x0464	1 block	0x0470
.const	0x0583	1 block	0x0590
.switch	0x0134	1 block	0x0140

Which gives a total of 0xe610 bytes, almost 64 KB (57936 bytes). This will fit comfortably in the 502 KB flash memory available (512 KB – 8 KB for the HUNT ENGINEERING boot loader).

Appendix B How to map sections

In order for the flash programming to work properly, you should not use addresses 0x400 to 0x1000 in your linker command file. This is because the HUNT ENGINEERING boot loader uses those addresses and if these get overwritten whilst your program is loaded into memory, the loading process will simply stop at some point, and will appear never to start up. You should also not uses addresses 0x8000fc00 to 0x80010000 for initialised data sections. It's OK, though, to use this address range for uninitialised sections, such as for example the stack ("*.stack*") or the heap ("*.sysmem*").

If you are using a batch file to compile and link your application, please edit the linker command file that you are using so that software sections follow the rules above. For example, take the following linker command file ("**.cmd*"), taken from the "*flaspro1*" directory.

```
-c
-heap  0x800000         /* 8Mbytes of HEAP */
-stack 0x008000         /* 16Kbytes of STACK */

MEMORY
{
    VECS:    o = 00000000h  l = 00000400h /* Reset & interrupt vectors     */
    PMEM:    o = 00001000h  l = 0000F000h /* Internal Program Memory       */
    SBSRAM:  o = 00400000h  l = 00040000h /* 256Kbytes SBSRAM              */
    SDRAM:   o = 03000000h  l = 00FFFC00h /* 16Mbytes - 1Kbytes of SDRAM   */
    BMEM:    o = 80000000h  l = 00010000h /* Internal Data Memory          */
}

SECTIONS
{
    .vectors:   >       VECS
    .text:      >       PMEM

    .cinit:     >       BMEM
    .const:     >       BMEM
    .tables:    >       BMEM
    .data:      >       BMEM
    .stack:     >       BMEM
    .bss:       >       BMEM
    .sysmem:    >       SDRAM
    .cio:       >       BMEM
    .far:       >       BMEM
    .memtab:    >       BMEM
}
```

The MEMORY part in the linker command file ("**.cmd*") defines physical memory. On HERON1 modules, there's 64 KB internal program memory at address 0, 64 KB internal data memory at address 0x80000000, 256 KB SBSRAM at address 0x400000 and 16 Mbytes SDRAM at address 0x3000000.

The SECTIONS part "maps" software sections (on the left) onto physical memory (on the right). The software sections are defined by the C compiler. Certain pragma definitions also generate sections, such as the "*#pragma CODE_SECTION*" and the "*#pragma DATA_SECTION*". The HERON-API uses such definitions to create the "*heronapi_code*" and "*heronapi_data*" software sections. (For more informa-tion on pragma definitions, see the "*spru187e.pdf*" document in the \manuals directory on the HUNT ENGINEERING CD, section 7.6 "*Pragma Directives*".) In addition, assembly code may add software sections. In both cases (pragma and assembly) the software sections added are defined by you.

As you can see, the VECS section in MEMORY is defined to start at address 0, and is 0x400 bytes is size. The PMEM section starts at address 0x1000 and is 0xf000 bytes in size. This leaves

addresses 0x400 till 0x1000 un-used and thus the standard boot loader will not be overwritten.

All data sections are mapped onto BMEM, except "*.sysmem*". We need to look at the map file to verify that indeed no initialised data sections would overwrite addresses 0x8000fc00 or upward. In this case, the initialised data sections are "*.cinit*" and "*.const*". Looking at the map file in Appendix A, we see that "*.cinit*" runs from 0x80000000 till 0x80000464, and "*.const*" runs from 0x80000464 till 0x800009e5. Clearly this poses no problems, since all addresses are lower than 0x8000fc00.

If you are using Code Composer Studio, do the following. In the processor window of the processor whose flash memory you want to program, open the "**.cdb*" file (*DSP/BIOS Config*). In the right-hand window, open the memory manager (*MEM-Memory Section Manager*). Right-click *IPRAM* and select "*Properties*". In the "*IPRAM Properties*" window that has appeared, make sure that the "*base*" field is set to *0x1000* or higher. And make sure that the "*len*" field is set to *0xf000* or lower. Click "*OK*". Now re-link your application (Project → Build).

We need to look at the map file to verify that indeed no initialised data sections would overwrite addresses 0x8000fc00 or upward. In Appendix A (above) it is explained how to generate a map file. Open the map file (File → Open). Ignore all UNINITIALISED sections, but inspect all sections that are mapped onto IDRAM (which starts at 0x80000000). These should not use any address between 0x8000 fc00 and 0x80010000. The "*.bss*" and "*.far*" sections can be disregarded. If you find overlap with the 0x8000fc00-0x8001000 region, map 1 or more initialised sections to a different physical block, or try to make that initialised section smaller. You can map software sections onto a different physical block as follows. Open the "**.cdb*" file (*DSP/BIOS Config*). Right-click "*MEM - Memory Section Manager*". In the menu that now appears, select "*Properties*". In the window that now appears (*MEM – Memory Sec-tion Manager Properties*), change 1 or more fields that now map onto "*IDRAM*" to a different physical block. Re-compile and re-link your application (Project → Build) and inspect your map file again, until you find that no initialised data sections overlap the 0x8000fc00 – 0x80010000 area.