XAPP983 (v1.0) September 24, 2007

Executing and Debugging Software From Flash Memory

Author: Simon George

Summary

This document and the associated reference design provide guidance for assigning and debugging software to or in FLASH memory, specifically for a MicroBlaze™ embedded processor design.

This application note is comprised of five sections:

- Defining the Architecture and Building Microcontroller Platform
- Creating the Flash based Software Project
- Configuration of Debug Platform
- Manipulating ELF image and Programming Flash
- Debugging Software Platform

It is assumed that the user is familiar with the Xilinx Embedded design methodology and associated tool chains.

Included Systems

Included with this application note is reference system XAPP983. The reference system is available for downloading at:

www.xilinx.com/bvdocs/appnotes/xapp983.zip

Hardware and Software Requirements

Software

- FPGA Platform: Xilinx Integrated Software Environment - ISE 9.1i SP3
- Embedded Platform: Xilinx Embedded Development Kit - EDK 9.1i SP2

Hardware

The development board listed below is used in this application note. However, with reliance on a minimal set of external components, the methodology remains valid for most alternative commercial and custom platforms, or both.

Xilinx Spartan™- 3E Starter Kit

© 2007 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at http://www.xilinx.com/legal.htm. PowerPC is a trademark of IBM Inc. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

Embedded Subsystem

The embedded subsystem is comprised of the modules shown in Figure 1.

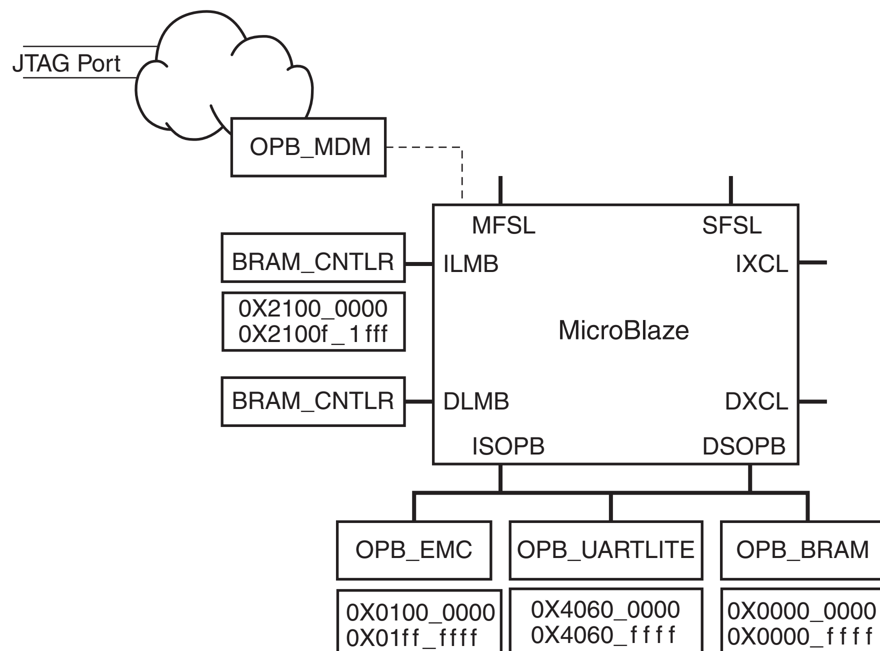

Figure 1: **Embedded Platform Block Diagram (Clk Freq = 50MHz)**

Design Methodology

In this application note, a software project name of `ExecuteFromFlash-XPS` has been chosen. The user can use their own project name. Note that the associated scripts must also be changed.

Defining the Architecture and Building the Microcontroller Platform

Use Platform Studio to create an embedded processing platform that includes, as a minimum, the building blocks illustrated in Figure 1. The use of the Base System Builder wizard is recommended to accelerate the process and to provide automatic handling of clock, reset, and JTAG connection dependencies.

Note: To enable *run time instruction stepping* when debugging from a read only memory space, such as *flash*, hardware breakpoint functionality is required of the target processor. This functionality is available for the Xilinx MicroBlaze processor by selecting On-chip H/W debug module from the Processor configuration window as shown in Figure 2.

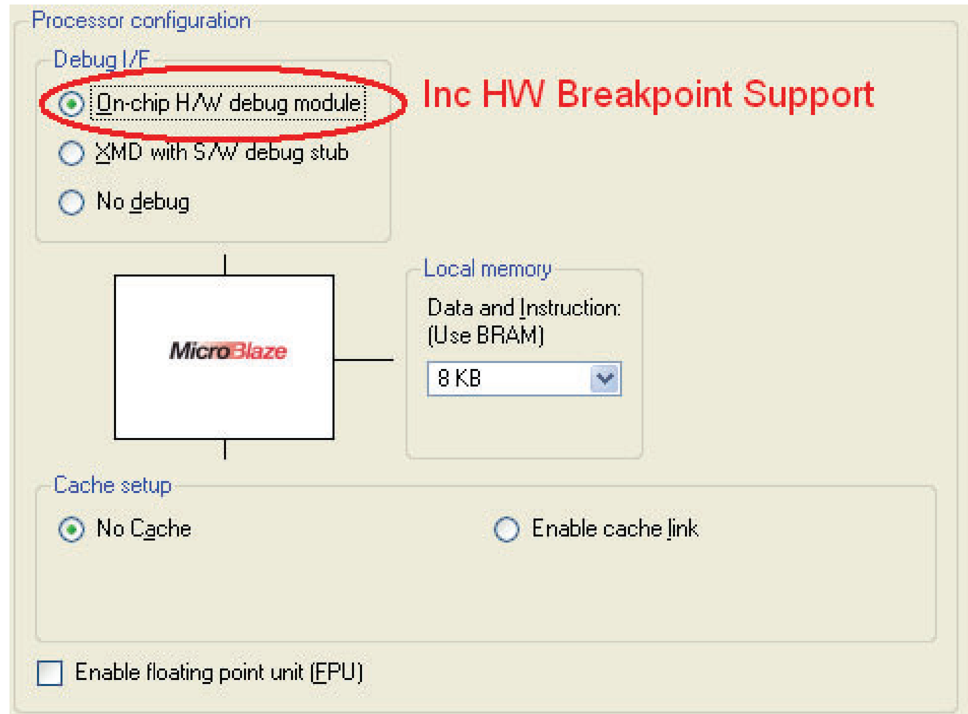

Figure 2: **BSB MIcroBlaze Control**

Following the completion of the subsystem design, it is strongly advised that the number of HW breakpoints built into the MicroBlaze core be increased from the default value of 2 to a value of 4 or more to allow more elaborate break and run sequences once the user is in the software development and debug stage of the project. The generated system should appear similar to that shown in Figure 3.

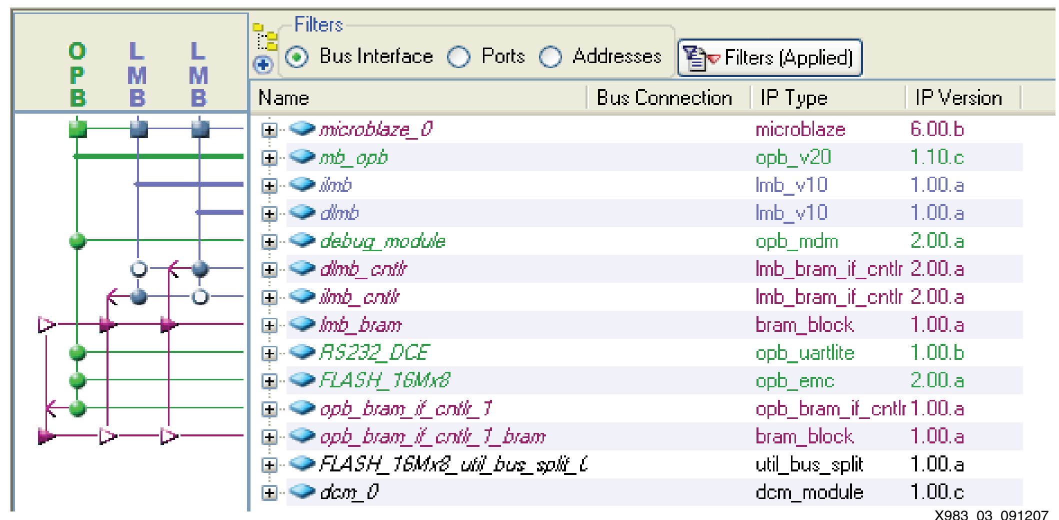

Figure 3: **System Assembly Bus Interface**

Double click on **CPU core** in the Platform Studio System Assembly window to open the MicroBlaze configuration window shown in Figure 4 where the options and settings of the CPU core configuration are displayed. As advised earlier, change the value in the Number of PC Breakpoints field.

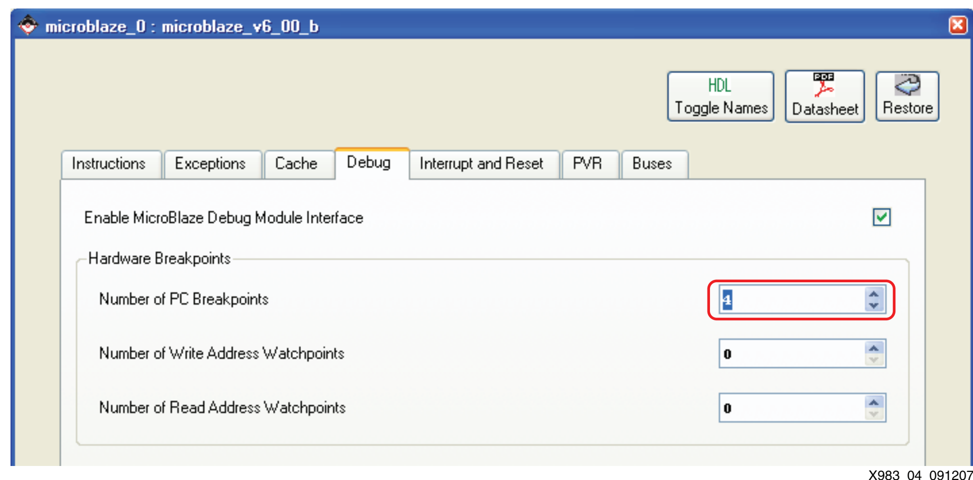

Figure 4: **MicroBlaze Configuration (Platform Studio View)**

When the subsystem architecture is fully defined, proceed to build the platform in preparation for target download.

While downloading, observe the serial console shown in Figure 5 to validate the successful system implementation.

```
-- Entering main() --
Starting MemoryTest for dlmb_cntlr:
  Running 32-bit test...PASSED!
  Running 16-bit test...PASSED!
  Running 8-bit test...PASSED!
-- Exiting main() --
```

X983_05_091207

Figure 5: **Sample Serial Console From BSB-Derived System**

The architecture of the embedded platform has been defined. Now the user can develop the associated software platform.

Creating the Flash-based Software Project (Platform Studio)

Create and migrate the Software project to be FLASH centric by mapping read only sections to FLASH memory space.

1. In XPS, **Software** → **Add Software Application Project** to create a new Software project.

2. Define project name `<ExecuteFromFlash-XPS>` as shown in Figure 6, then click **OK**.

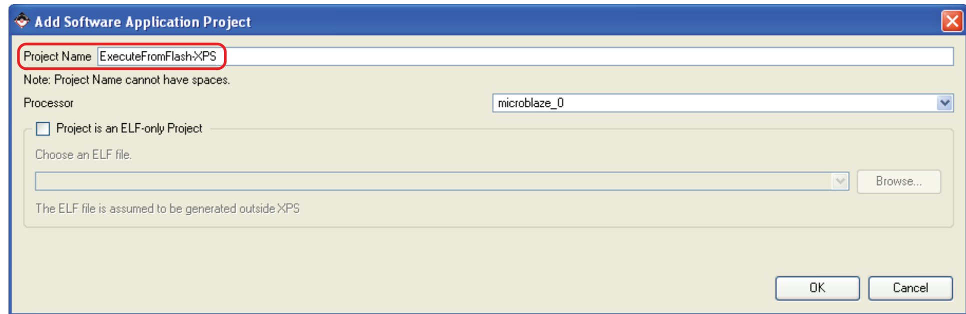

Figure 6: **New Software Project**

3. In XPS select **Software** → **Generate Linker Script** to invoke the Linker script wizard for the newly created SW project.

Using the drop down menus provided in the Generate Linker Script window, move the read only sections, .text, .rodata, .sdata2, and .sbss2 to the FLASH region as shown in Figure 7, then click **OK**. A description of the sections can be seen in Figure 7.

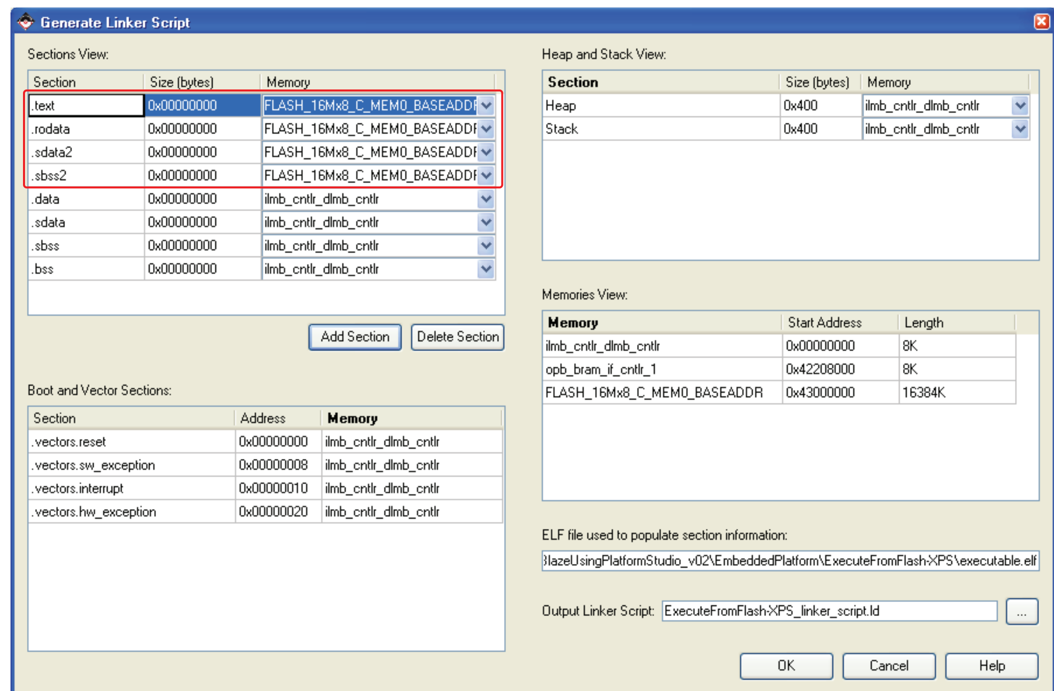

Figure 7: **Generate Linker Script**

4. Add source software code to the software application project. The TestApp_Memory code provided by the BSB Wizard during initial system creation may be used during the initial system creation as has been done in the associated example project.

5. Disable compiler optimization for the software project to allow initial debug and code validation as shown in Figure 8 and Figure 9. Once the initial debug and code validation has been completed, the optimization may be enabled. To do so, right click on Project: `<ExecuteFromFlash-XPS>`, and select **Set Compiler Options**.

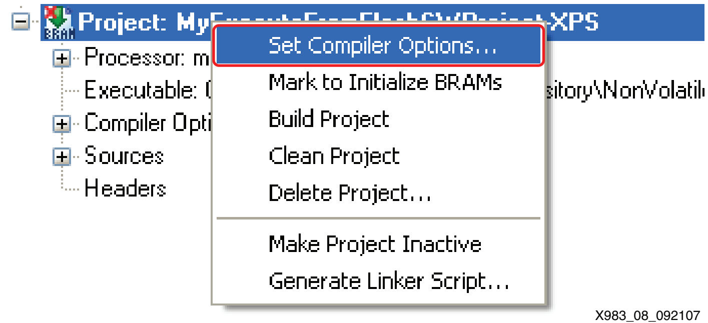

Figure 8: **Accessing Compiler Options**

In the Set Compiler Options window, select the Debug and Optimization tab, then make the selections shown in Figure 9.

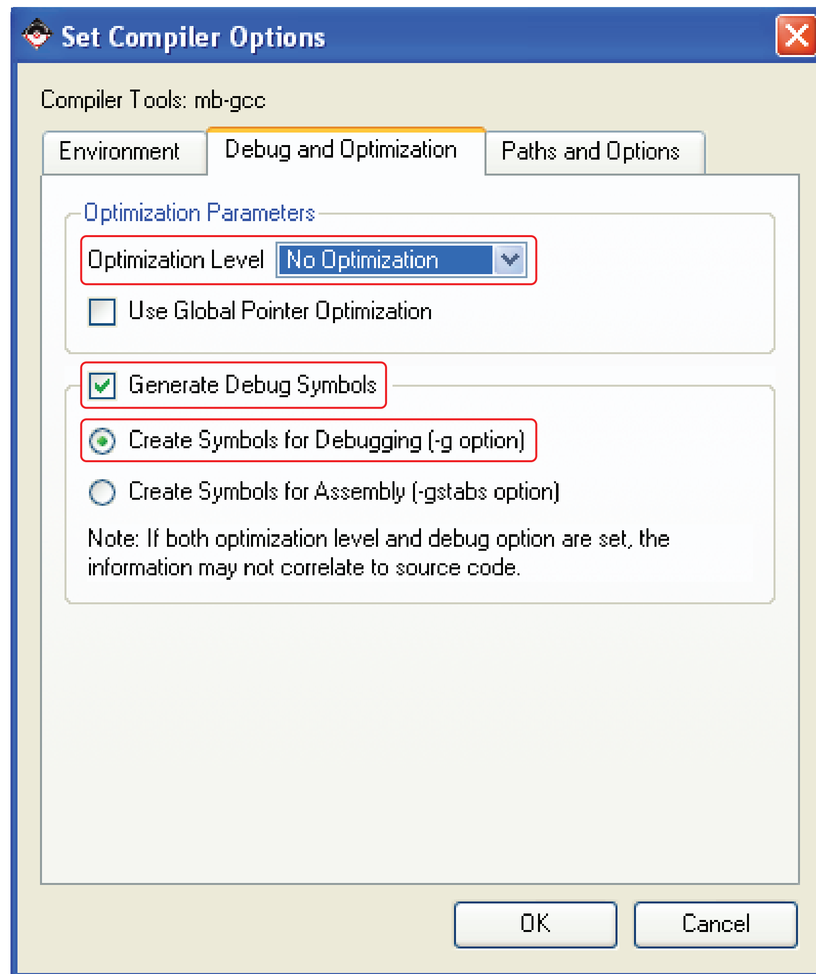

Figure 9: **Setting Compiler Options**

6. In XPS, select **Software** → **Build All User Applications** to compile the software project.

The software project is now compiled, assembled, and linked.

Configuring the Debug Platform

Configure the debug environment to handle non-volatile memory space.

1. Locate the base address and associated size of the non-volatile memories shown in Figure 10.

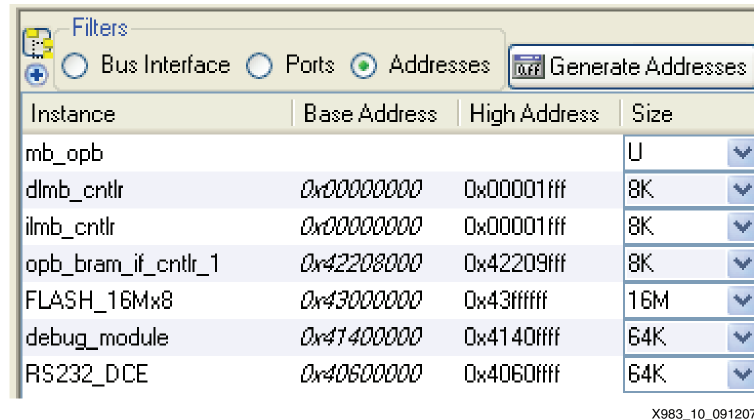

Figure 10: **System Assembly Address**

2. Define the read only memory segment. To instruct the Xilinx MicroProcessor Debug agent to use the hardware breakpoints for PC (Program counter), compare points in the identified FLASH region. In the XPS menu, select **Debug** → **XMD Debug Options**.

   In the XMD Debug Options window, select the Advanced Options tab. Make the selections as shown in Figure 11.

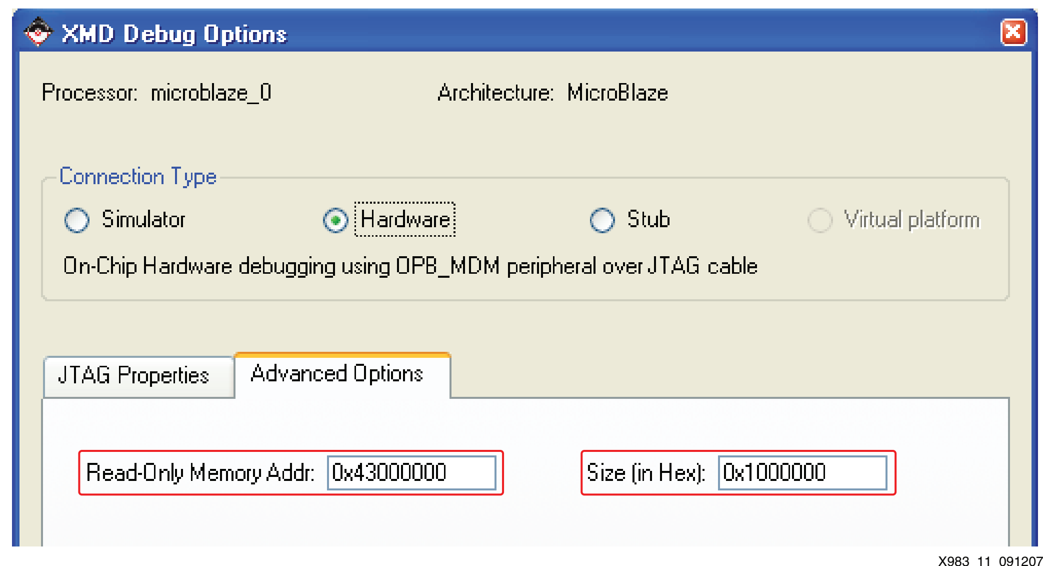

Figure 11: **XMD Debug Options**

Manipulating the ELF Image and Programming FLASH

Manipulate the compiled software image to create two discrete and distinct images — one for download into the volatile memory of the design and one for the persistent FLASH region. Program the persistent FLASH region into CFI compliant parallel Flash memory.

Following the compilation of the `<ExecuteFromFlash-XPS>` SW project, the resultant executable .elf file is a single file representation of the complete software image, which is solely segmented by memory region. Because these memory regions require different programming techniques, the image must be segmented. To segment the memory region, use the following steps:

1. Create elf with *no load* set on FLASH memory mapped regions – namely 'volatile.elf'

   In XPS, select **Project** → **Launch EDK Shell**.

   ```
   $ mb-objcopy \
   --set-section-flags .text=alloc,readonly,code \
   --set-section-flags .init=alloc,readonly,code \
   --set-section-flags .fini=alloc,readonly,code \
   --set-section-flags .rodata=alloc \
   --set-section-flags .sdata2=contents \
   --set-section-flags .sbss2=contents \
   ./ExecuteFromFlash-XPS/executable.elf ./ExecuteFromFlash-
   XPS/volatile.elf
   ```

2. Create binary image containing FLASH mapped sections, allowing subsequent download by Flash Writer – namely 'flash.bin'

   In XPS, select **Project** → **Launch EDK Shell**.

   ```
   $ mb-objcopy -O binary \
   -j .text \
   -j .init \
   -j .fini \
   -j .rodata \
   -j .sdata2 \
   -j .sbss2 \
   ./ExecuteFromFlash-XPS/executable.elf
   ./ExecuteFromFlash-XPS/flash.bin
   ```

3. Invoke the Platform Studio *Flash writer* utility to program the *flash.bin* image into the FLASH memory device(s). Configure the utility as shown in the Program Flash Memory window in Figure 12, then select **OK**.

   In XPS, select **Device Configuration** → **Program Flash Memory**.

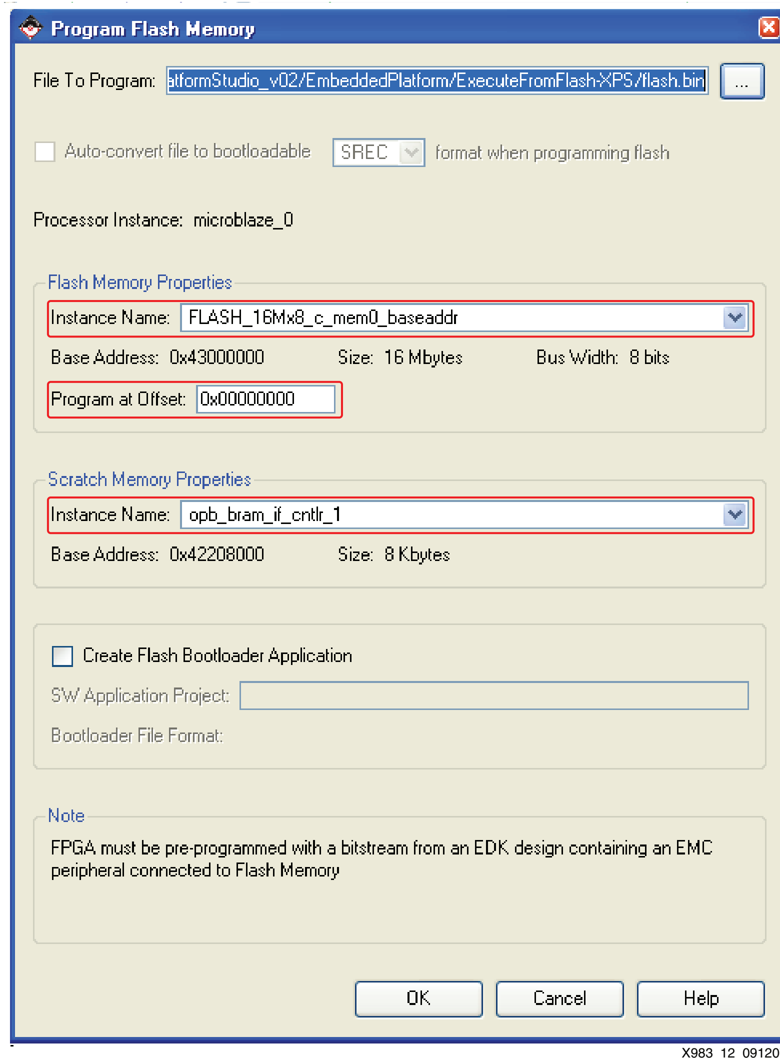

Figure 12: **Flash Writer Utility**

Programming may take several 10s of seconds. Monitor the console window in Platform Studio during this time to observe how the programming is progressing. An example of a successful programming completion is shown in Figure 13.

```
* ----------------------------------------------------------------
* Flash part information:
* ----------------------------------------------------------------
* Device Algorithm              : Intel/Sharp Extended
* Number of flash parts         : 1
* Mode of each flash part       : 8 bits wide
* Size of each flash part       : 16 MB
* Flash sector config           : DEFAULT
* ----------------------------------------------------------------

* Erasing appropriate flash block(s)...
done.

* Programming of the flash part with the image starts...

* Programming completed:  91.56%

* Programming completed: 100.00%
* Flashwriter completed successfully!

Done!
```

X983_13_091207

Figure 13: **Successful Flash Programming Example**

Note: The detailed commands can be copied into a cygwin shell directly to avoid extensive typing

Debugging the Software Platform

With the bitstream built and FLASH memory programmed, the user may now debug the configured target.

1. In XPS, select **Debug** → **Launch XMD…** to establish a debug connection from the host to the MicroBlaze target as shown in Figure 14.

```
JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       01c22093          6         XC3S500E
 2       05046093          8         XCF04S
 3       06e5e093          8         XC2C64A_VQ44_1532

MicroBlaze Processor Configuration :
-------------------------------------
Version............................6.00.b
No of PC Breakpoints...............4
No of Read Addr/Data Watchpoints...0
No of Write Addr/Data Watchpoints..0
Instruction Cache Support..........off
Data Cache Support.................off
Exceptions  Support................off
FPU  Support.......................off
Hard Divider Support...............off
Hard Multiplier Support............on - (Mul32)
Barrel Shifter Support.............off
MSR clr/set Instruction Support....on
Compare Instruction Support........on

Connected to MDM UART Target
Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1235
        section, .vectors.reset: 0x00000000-0x00000007
        section, .vectors.sw_exception: 0x00000008-0x0000000f
        section, .vectors.interrupt: 0x00000010-0x00000017
        section, .vectors.hw_exception: 0x00000020-0x00000027
        section, .text: 0x43000000-0x43000fcb
        section, .init: 0x43000fcc-0x43000ff7
        section, .fini: 0x43000ff8-0x43001017
        section, .rodata: 0x43001018-0x43001137
        section, .data: 0x00000050-0x0000006b
        section, .ctors: 0x0000006c-0x00000073
        section, .dtors: 0x00000074-0x0000007b
        section, .jcr: 0x0000007c-0x0000007f
        section, .bss: 0x00000080-0x0000008b
        section, .heap: 0x0000008c-0x0000048f
        section, .stack: 0x00000490-0x0000088f
Downloaded Program ./ExecuteFromFlash-XPS/volatile.elf
Setting PC with Program Start Address 0x00000000

XMD% _
```

X983_14_091207

Figure 14: **XMD Debug Session Console**

2. Establish a *GDB Insight* software debug session to allow graphical interaction with the target. Select the FLASH centric software project when prompted for a target project.

   From XPS, select **Debug** → **Launch Software Debugger**.

3. Configure the GDB connections to the target as shown in the Target Selection window in Figure 15.

   From the GDB Debugger, select **File** → **Target Settings**.

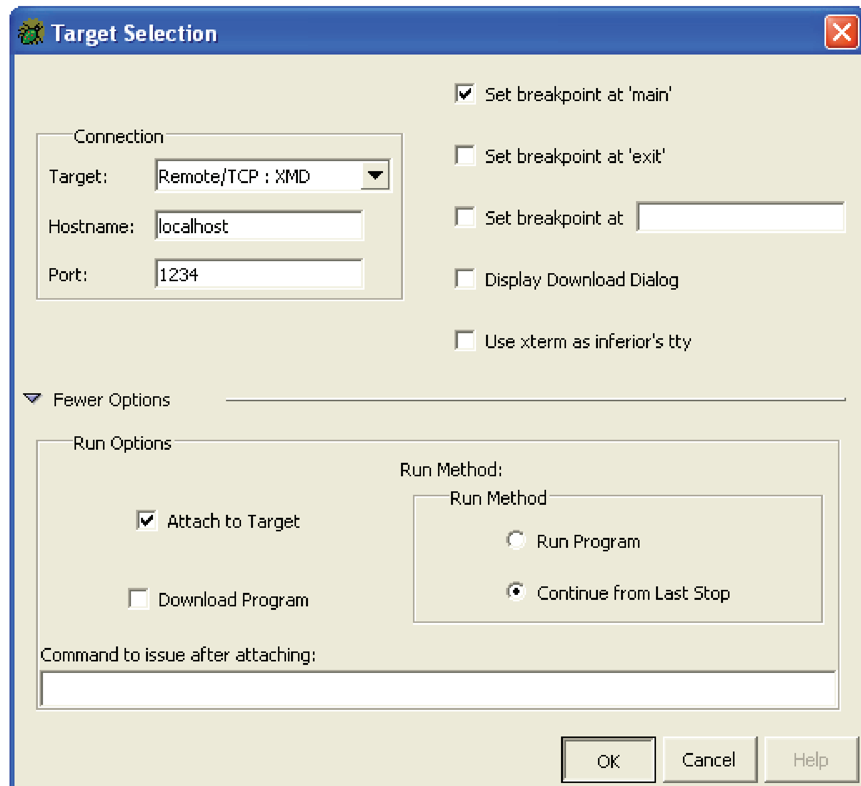

Figure 15: **GDB Target Connection Settings**

WARNING!

Make certain not to download the program again because the debugger is working with the full `executable.elf` image, something which GDB and XMD cannot download successfully because of the read only nature of the FLASH mapped region. The reason for using the full image and not just the `volatile.elf` segment is to retain assembly code visibility during debug, which can be sourced from a file only — not from the programmed machine code of the target.

4. Select **OK** in step 3 to connect GDB to the target and to allow debug to begin. The breakpoints set in the FLASH region will be converted to HW breakpoints, while the breakpoints set in the volatile regions will use a more extensible software trap mechanism. See Figure 16.

```
TestApp_Memory.c - Source Window
File  Run  View  Control  Preferences  Help
Find:
TestApp_Memory.c        main
         43 int main (void) {
- 0x4300011c <main>:                    addik   r1, r1, -36
- 0x43000120 <main+4>:                  swi     r15, r1, 0
- 0x43000124 <main+8>:                  swi     r19, r1, 32
- 0x43000128 <main+12>:                 addk    r19, r1, r0
         44
         45
         46     /*
         47      * Enable and initialize cache
         48      */
         49     #if XPAR_MICROBLAZE_0_USE_ICACHE
         50        microblaze_init_icache_range(0, XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
         51        microblaze_enable_icache();
         52     #endif
         53
         54     #if XPAR_MICROBLAZE_0_USE_DCACHE
         55        microblaze_init_dcache_range(0, XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
         56        microblaze_enable_dcache();
         57     #endif
         58
         59     print("-- Entering main() --\r\n");
  0x4300012c <main+16>:                 imm     17152
- 0x43000130 <main+20>:                 addik   r5, r0, 4120 // 0x43001018 <__rodata_start>
- 0x43000134 <main+24>:                 brlid   r15, 3516 // 0x43000ef0 <print>
- 0x43000138 <main+28>:                 or      r0, r0, r0
         60
```

X983_16_091207

Figure 16: **GDB Connection to Target**

Because the operation from here is identical to that experienced when running from volatile memory, the environment can be used to debug in a conventional manner. However, to restart from program entry, close the GDB and XMD sessions to ensure that all volatile data sections remain *untouched* prior to entry into main.

Additional Considerations

If the user determines that the basic memory tests are not working, consider the following:

The Spartan-3E 500 starter kit is shipped with the FPGA configuration mode set to Master-Serial which is what is needed for initial system demonstration. However, to allow platform programming with a JTAG download, the configuration mode must be changed. Therefore, change J30 to 'JTAG' mode as per the PCB legend.

If the user balks at generating the numerous *elf* variants, consider this:

By creating volatile and non volatile elf sub images, the debug sessions can be iteratively restarted without the need to re-download the bitstream.

```
Address Map for Processor microblaze_0
  (0000000000-0x00001fff) dlmb_cntlr    dlmb
  (0000000000-0x00001fff) ilmb_cntlr    ilmb
  (0x40600000-0x4060ffff) RS232_DCE     mb_opb
  (0x41400000-0x4140ffff) debug_module  mb_opb
  (0x42208000-0x42209fff) opb_bram_if_cntlr_1   mb_opb
  (0x43000000-0x43ffffff) FLASH_16Mx8   mb_opb

Initializing Memory...
Checking ELFs associated with MICROBLAZE instance microblaze_0 for overlap...


Analyzing file ExecuteFromFlash-XPS/executable.elf...
WARNING:MDT - Elf file ExecuteFromFlash-XPS/executable.elf does not reside
   completely within BRAM memory of processor microblaze_0.
WARNING:MDT - The sections of  ELF residing outside BRAMs must be initialized
   separately using a debugger, a bootloader, or an ACE fileINFO:MDT - BRAM lmb_bram will be initialized with ELF of processor microblaze_0
Running Data2Mem with the following command:
data2mem -bm "implementation/system_bd" -bt "implementation/system.bit"  -bd
"ExecuteFromFlash-XPS/executable.elf" tag microblaze_0  -o b
implementation/download.bit

Memory Initialization completed successfully.


Done!
```

X983_17_091207

Figure 17: **ELF Memory Region Overlap Warning**

Creating an xmd.ini File

An `xmd.ini` file resident in the root directory of Platform Studio allows command line options to be passed to a new XMD session, thereby allowing the automatic delivery of customization commands that are required for a specific project, and saving the user from having to type the command line options each time. To create the `xmd.ini` file, use the following steps:

1. Cut and paste between “Start” and “End” lines in a new text file named `xmd.ini`.
2. Save file in root directory of active Platform Studio project.

   Note: If using the XPS integrated editor, make certain to change the ‘Save as type’ entry to *.*, to ensure that an `xmd.ini.c` file will not be created.

---*Start of file* ---

```
rst
dow ./ExecuteFromFlash-XPS/volatile.elf
```

---*End of file* --

Creating an ELFImageManipulationAndDownloadToFlash.tcl File

Cut and paste between Start and End lines in a new text file named `ELFImageManipulationAndDownloadToFlash.tcl`

3. Save file in the root directory of active Platform Studio project
4. Optional step - Map the script to custom button of XPS for maximum ease of use.

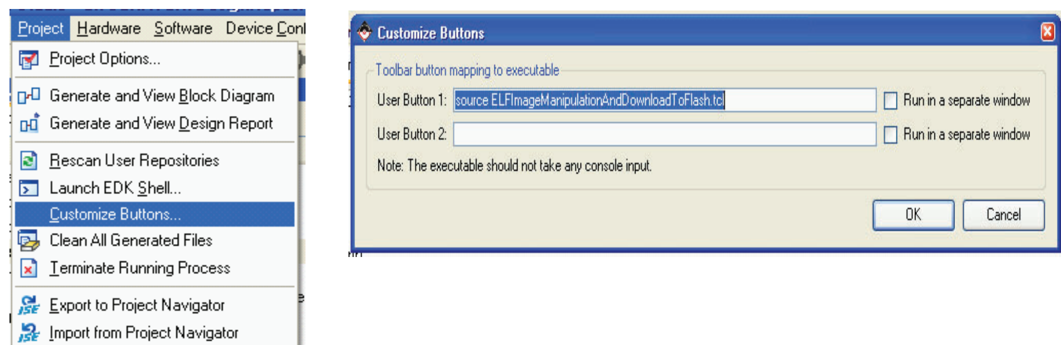

Figure 18: **Creating a tcl File**

--*Start of file*--

```
# Create elf with no load set intermediate download
mb-objcopy \
--set-section-flags .text=alloc,readonly,code \
--set-section-flags .init=alloc,readonly,code \
--set-section-flags .fini=alloc,readonly,code \
--set-section-flags .rodata=alloc \
--set-section-flags .sdata2=contents \
--set-section-flags .sbss2=contents \
./ExecuteFromFlash-XPS/executable.elf ./ExecuteFromFlash-
XPS/volatile.elf
echo "Stage 1... Done!"

# Create binary image for FLASH mapped sections for download by
Flash Writer
mb-objcopy -O binary \
-j .text \
-j .init \
-j .fini \
-j .rodata \
-j .sdata2 \
-j .sbss2 \
./ExecuteFromFlash-XPS/executable.elf ./ExecuteFromFlash-
XPS/flash.bin
echo "Stage 2... Done!"

#xmd -tcl flashwriter.tcl
#echo "Stage 3... Done!"
```

--*End of file*--

Because the Flashwriter.tcl script is generated by the Flash Writer utility GUI, the loop must be manually undertaken the first time around. Once that has occurred, the user can uncomment the Stage3 entries in the script.

Explanation of ELF File Sections

An executable file is created by concatenating input sections from the object files (.o files) being linked together. The compiler, by default, creates code across standard and well defined sections. Each section is given a well-known name based on its associated meaning and purpose. The various standard sections of the object file are displayed in Figure 19. In addition to these sections, users can also create custom sections and assign them to selected memories.

Section	Description
.text	Text Section
.rodata	Read-Only Data Section
.sdata2	Small Read-Only Data Section
.data	Read-Write Data Section
.sdata	Small Read-Write Data Section
.sbss	Small Uninitialized-Data Section
.bss	Uninitialized-Data Section

X983_19_091207

Figure 19: **Section Layout of an Object or Executable File**

In addition to the sections described, custom sections can be created and assigned to memories of the user's choice.

.text

This section of the object file contains executable program instructions. This section has the `x` (executable), `r` (read-only) and `i` (initialized) flags. This means that this section can be assigned to an initialized read-only memory (ROM) that is addressable from the processor instruction bus.

.rodata

This section contains read-only data. This section has the `r` (read-only) and the i (initialized) flags. Like the `.text` section, this section can also be assigned to an initialized, read-only memory that is addressable from the processor data bus.

.sdata2

This section is similar to the .rodata section. It contains small read-only data of size less than 8 bytes. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all the contents of this section are accessed using a single instruction. The size of the data going into this section can be changed with the -G option to the compiler. This section has the `r` (read-only) and the `i` (initialized) flags

.data

This section contains read-write data and has the `w` (read-write) and the `i` (initialized) flags. It must be mapped to initialized random access memory (RAM). It cannot be mapped to a ROM.

.sdata

This section contains small read-write data of a size less than a specified size; the default is 8 bytes. The size of the data going into this section can be changed with the -G option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all contents of the section can be accessed using a single instruction. This section has the `w` (read-write) and the `i` (initialized) flags and must be mapped to initialized RAM.

.sbss

This section contains small un-initialized data of a size less than a specified size; the default is 8 bytes. The size of the data going into this section can be changed with the -G option. This section has the `w` (read-write) flag and must be mapped to RAM.

.bss

This section contains un-initialized data. The program stack and the heap are also allocated to this section. This section has the `w` (read-write) flag and must be mapped to RAM.

.heap

This section contains uninitialized data that is used as the global program heap. Dynamic memory allocation routines allocate memory from this section. This section must be mapped to RAM.

.stack

This section contains uninitialized data that is used as the program stack. This section must be mapped to RAM. This section is typically laid out right after the `.heap` section. In some versions of the linker, the `.stack`, and `.heap` sections might appear merged together into a section named `.bss_stack`.

.init

This section contains language initialization code and has the same flags as `.text`. It must be mapped to initialized ROM.

.fini

This section contains language cleanup code and has the same flags as `.text`. It must be mapped to initialized ROM.

.ctors

This section contains a list of functions that must be invoked at program startup and the same flags as `.data`, and it must be mapped to initialized RAM.

.dtors

This section contains a list of functions that must be invoked at program end, the same flags as `.data`, and it must be mapped to initialized RAM.

.got2/.got

This section contains pointers to program data, the same flags as `.data`, and it must be mapped to initialized RAM.

.eh_frame

This section contains frame unwind information for exception handling. It contains the same flags as `.rodata` and can be mapped to initialized ROM

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
9/24/07	1.0	Initial Xilinx release.