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
Embedded Subsystem

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

![Embedded Platform Block Diagram](image)

Notes:
1. The UART peripheral is not mandatory. It has been included to ease platform testing through the observation of serial console activity on the host, a consequence of running embedded code on the target.
2. The inclusion of LMB BRAM and OPB BRAM are representative of the need for read/writable memory space. The use of two spaces allow the automatically generated TestApp_Memory software project to be validated prior to system changes.

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.

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.
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.

![MicroBlaze Configuration (Platform Studio View)](image)

**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.

![Sample Serial Console From BSB-Derived System](image)

**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.

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.

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](image)

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](image)
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**.

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 10: System Assembly Address](image)

![Figure 11: XMD Debug Options](image)
**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.**
   
   ```shell
   $ mb-objcopy \n   --set-section-flags .text=alloc,readonly,code \n   --set-section-flags .init=alloc,readonly,code \n   --set-section-flags .fini=alloc,readonly,code \n   --set-section-flags .rodata=alloc \n   --set-section-flags .sdata2=contents \n   --set-section-flags .sbss2=contents \n   ./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.**
   
   ```shell
   $ mb-objcopy -O binary \n   -j .text \n   -j .init \n   -j .fini \n   -j .rodata \n   -j .sdata2 \n   -j .sbss2 \n   ./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](X983_12_091207)
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.

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.
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**.

---

**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.

---

*Figure 15: GDB Target Connection Settings*
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.

![Figure 16: GDB Connection to Target](image)

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.

![Figure 17: ELF Memory Region Overlap Warning](image)
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 ‘*.c’, 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.

![Creating a tcl File](image)

**Figure 18: Creating a tcl 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!"

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.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>Text Section</td>
</tr>
<tr>
<td>.rodata</td>
<td>Read-Only Data Section</td>
</tr>
<tr>
<td>.sdata2</td>
<td>Small Read-Only Data Section</td>
</tr>
<tr>
<td>.data</td>
<td>Read-Write Data Section</td>
</tr>
<tr>
<td>.sdata</td>
<td>Small Read-Write Data Section</td>
</tr>
<tr>
<td>.sbss</td>
<td>Small Uninitialized-Data Section</td>
</tr>
<tr>
<td>.bss</td>
<td>Uninitialized-Data Section</td>
</tr>
</tbody>
</table>

*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 \texttt{x} (executable), \texttt{r} (read-only) and \texttt{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 \texttt{r} (read-only) and the \texttt{i} (initialized) flags. Like the \texttt{.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 \texttt{.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 \texttt{r} (read-only) and the \texttt{i} (initialized) flags

.data
This section contains read-write data and has the \texttt{w} (read-write) and the \texttt{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 \texttt{w} (read-write) and the \texttt{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 \texttt{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 \texttt{w} (read-write) flag and must be mapped to RAM.

.heap
This section contains unitialized 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 \texttt{.heap} section. In some versions of the linker, the \texttt{.stack}, and \texttt{.heap} sections might appear merged together into a section named \texttt{bss_stack}.

.init
This section contains language initialization code and has the same flags as \texttt{.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.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/24/07</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
</tbody>
</table>