SmartFusion cSoC: Loading and Booting from External Memories

Table of Contents

Introduction

The SmartFusion® customizable system-on-chip (cSoC) device contains a hard embedded microcontroller subsystem (MSS), programmable analog circuitry, and FPGA fabric consisting of logic tiles, static random access memory (SRAM), and phase-locked loops (PLLs). The MSS consists of a 100 MHz ARM® Cortex™-M3 processor, advanced high-performance bus (AHB) matrix, system registers, Ethernet MAC, DMA engine, real-time counter (RTC), embedded nonvolatile memory (eNVM), embedded SRAM (eSRAM), fabric interface controller (FIC), the Philips Inter-Integrated Circuit (I^2C), and serial peripheral interface (SPI) peripherals.

The MSS has two identical SPI peripherals. These peripherals provide serial interface compliance with the Motorola SPI, Texas Instruments synchronous serial, and National Semiconductor MICROWIRE™ formats. The SPI peripherals in the SmartFusion cSoC can operate as either a Master or a Slave. When operating in the Master mode, the SPI peripherals generate a serial clock and data to the slave device that needs to be accessed. The SPI peripherals can generate a serial clock from 390 KHz to 50 MHz by dividing the MSS clock, which can be controlled by the software. The peripheral DMA (PDMA) in the MSS can be used for continuous DMA streaming for large SPI transfers and thus helps to free up the ARM Cortex-M3.

The external memory controller (EMC) provides glueless interface to external memory devices that can be addressed by the ARM Cortex-M3 or user logic in the FPGA fabric. The EMC is divided into two regions; namely, Region 0 and Region 1. The Region 0 and Region 1 can be independently configured for three memory types supported by EMC (NOR flash, asynchronous RAM, and synchronous RAM). The MSS configuration tool provides a graphical user interface (GUI) to configure the memory type, port size, and latency.

Refer to the *SmartFusion Microcontroller Subsystem User's Guide* for more details on SPI peripherals and the EMC.

This application note describes how to load and boot the application code from the external memories (SPI flash memory and NOR flash memory) on the SmartFusion Development Kit Board. This document also describes the detailed design of the NOR flash driver and SPI flash driver.

A basic understanding of the SmartFusion design flow is assumed. Refer to the *Using UART with a SmartFusion cSoC - Libero SoC and SoftConsole Flow Tutorial* to understand the SmartFusion design flow.

Design Example Overview

This design example demonstrates loading and booting the application code from the external memory devices on the SmartFusion Development Kit Board. It uses one SPI peripheral, one universal asynchronous receiver/transmitter (UART) peripheral, the EMC, and three general purpose input/output (GPIOs) in the MSS. The SPI peripheral, *SPI_1*, is connected to the Atmel SPI flash memory, AT25DF641-MWH-T, while the EMC is connected to the Cypress SRAM, CY7C1061DV33-10ZSXI, at Region 0 and Numonyx NOR flash, JS28F640J3D-75, at Region 1 in the SmartFusion Development Kit Board. The GPIOs are configured as outputs and connected to test the LEDs in the SmartFusion Development Kit Board. These LEDs blink continuously with certain delay while the UART prints the memory location of the global variable on the HyperTerminal. Figure 1 shows the system level block diagram.

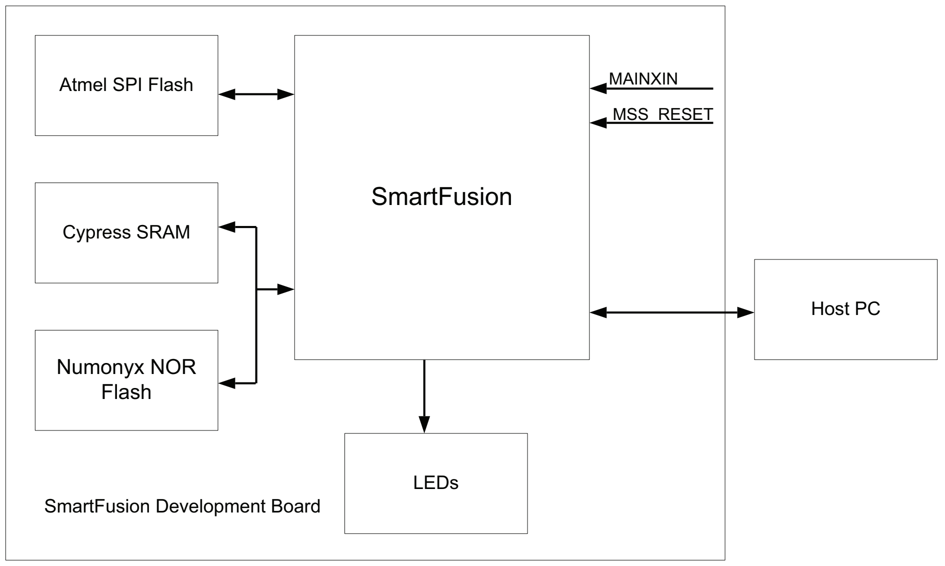


***Figure 1* • System Level Block Diagram**

Design Example Description

There are mainly two methods to load and boot the application code from the external memories in the SmartFusion cSoC as mentioned below.

- Loading the application code into the SPI flash memory and booting from the external SRAM
- Loading and booting the application code from the external NOR flash memory

This design example consists of mainly three steps as follows:

1. Creating the executable image for the application code that can be run from the external memories
2. Loading the executable image from the Host PC to the external flash memories
3. Booting the image from the external memories

The following section gives a detailed description of each step for booting from the external SRAM and the NOR flash memory.

Creating an Executable Image for External Memories

Creating an Executable Image for SPI Flash Memory

The SmartFusion cSoC does not support booting the application code directly from the SPI flash since it is not directly memory mapped to the SmartFusion system memory map. You have to create an executable image which can be executed from the external SRAM and that needs to be stored into the SPI flash memory. You need to use the "production-execute-in-place-externalram.ld" linker description file that is included in the design files to build the image. This linker description file creates an image with instructions, data, and BSS sections in the external SRAM. For further details, refer to "Appendix A - Design Files" on page 14.

In this design example the executable image starts from the start location of the external SRAM, that is, 0x70000000.

Creating an Executable Image for NOR Flash Memory

The SmartFusion cSoC supports loading and booting the application code directly from the NOR flash memory. You need to use the "production-execute-in-place-emcflash.ld" linker description file that is included in the design files to build the executable image. This linker description file creates an image with data and BSS sections in the external SRAM and instructions in the external NOR flash. For further details, refer to "Appendix B - Linker Description Files" on page 15.

In this design example, the executable image starts from the start location of the external NOR flash, that is, 0x74000000.

Loading the Executable Image from Host PC to External Flash Memory

Loading the Executable Image into SPI Flash Memory

This section describes accessing the SPI flash memory and running the flash loader. Refer to the *Accessing Serial Flash Memory Using SPI Interface application note* for more details on the SPI flash driver APIs used in this example design to access the SPI flash memory. The flash loader loads the executable image from the development PC into the SPI flash memory. It uses the UART as the communication channel between the PC and the SmartFusion Development Kit Board to load the image. Refer to the "Flash Loader" section for more details on the usage of flash loader.

Loading the Executable Image into NOR Flash Memory

This section describes accessing the NOR flash memory and running the flash loader. Refer to "Appendix C - NOR Flash Driver Application Programming Interfaces (APIs)" on page 18 for more details on the NOR flash driver APIs used in this example design to access the NOR flash memory. The flash loader loads the executable image into the NOR flash memory. Refer to the "Flash Loader" section for more details on the usage of flash loader.

Flash Loader

The flash loader loads the executable image into the SPI flash memory and NOR flash memory. It consists of two parts: the host loader and the target loader. The host loader is responsible for sending the executable image from the PC to the UART port on the SmartFusion cSoC. The target loader is responsible for reading the executable image from the PC UART and programming the corresponding flash memory.

The flash loader uses the UART as a communication channel between the PC and the SmartFusion cSoC to load the image. So the UART on host and target needs to be properly synchronized. For this, you need to run the target loader first and make the target listen to the UART data from the host. Figure 2 illustrates the data flow of the flash loader.

The host loader (refer to "Appendix A - Design Files" on page 14) takes the following command line parameters:

1. Type of the flash: SPI or EMC flash
2. Address where the image has to be loaded into the flash
3. The name of the image to be loaded
4. Comport number: The COM port number where the SFE USB to RS232 controller is connected. (**My Computer > Manage > Device Manager > Ports (COM & LPT)**)

A sample usage of the flash loader on host for EMC flash (that is, NOR flash in this design example) is described below:

At the command prompt enter:

```
> f2flashloader.exe    emc    address    filename        comportnumber
```

Address range is from 0x70000000 to 0x73FFFFFF; if EMC region 0 is mapped to flash

Address range is from 0x74000000 to 0x77FFFFFF; if EMC region 1 is mapped to flash

For example, if the COM port number is 3 and the address where you are writing is from 0x74000000, then the command is:

```
> f2flashloader.exe emc 0x74000000 filename.bin 3
```

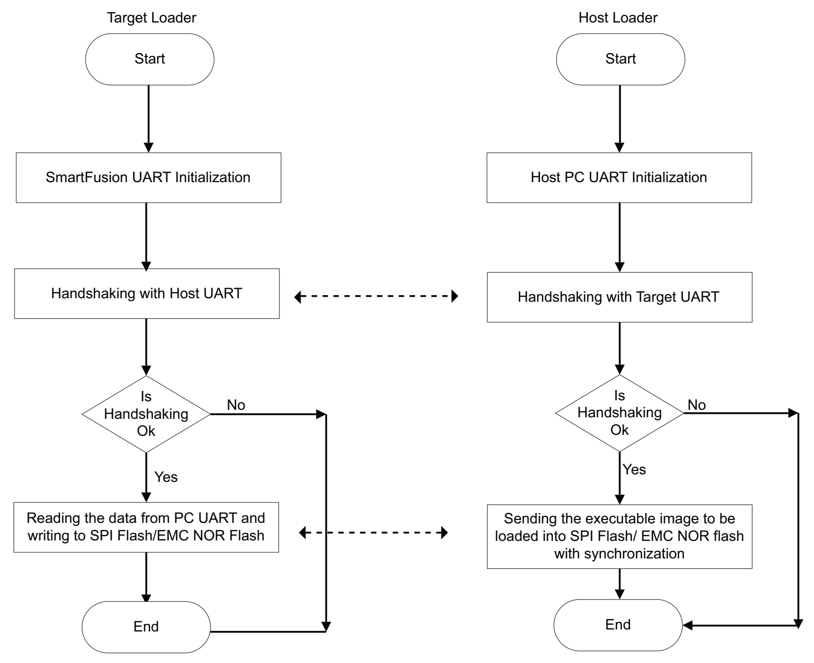

Target Loader
Start
SmartFusion UART Initialization
Handshaking with Host UART
Is Handshaking Ok
No
Yes
Reading the data from PC UART and writing to SPI Flash/EMC NOR Flash
End
Host Loader
Start
Host PC UART Initialization
Handshaking with Target UART
Is Handshaking Ok
No
Yes
Sending the executable image to be loaded into SPI Flash/ EMC NOR flash with synchronization
End

***Figure 2* • Flash Loader Flow Chart**

The following functions are used in the design example to load the executable image into the SPI flash memory and the NOR flash memory.

static void spi_flash_loader (void)

This function reads the executable image from the PC UART and programs the SPI flash memory. This function is a part of the main.c file that is available at:

../ext_mem_load_boot/main.c.

static void emc_flash_loader (void)

This function reads the executable image from the PC UART and programs the NOR flash memory. This function is a part of the main.c file that is available at:

../ext_mem_load_boot/main.c.

Booting the Image from External Memories

Booting from External SRAM

This section describes copying the image from the SPI flash to the external SRAM, copying the vector table from the external SRAM to the MSS eSRAM, branching to the user boot code, and booting from the external SRAM. Figure 3 illustrates the boot process.

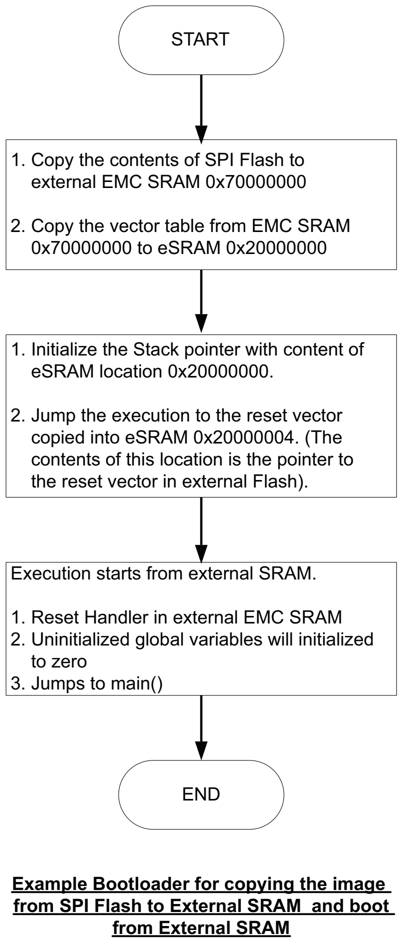

START
1. Copy the contents of SPI Flash to external EMC SRAM 0x70000000
2. Copy the vector table from EMC SRAM 0x70000000 to eSRAM 0x20000000
1. Initialize the Stack pointer with content of eSRAM location 0x20000000.
2. Jump the execution to the reset vector copied into eSRAM 0x20000004. (The contents of this location is the pointer to the reset vector in external Flash).
Execution starts from external SRAM.
1. Reset Handler in external EMC SRAM
2. Uninitialized global variables will initialized to zero
3. Jumps to main()
END
Example Bootloader for copying the image from SPI Flash to External SRAM and boot from External SRAM

***Figure 3* • Flow Chart for Booting From The External SRAM**

The design example uses the following function to carry out the boot process.

void spi_flash_to_esram(uint32_t srcAddr, uint32_t size)

This API copies the contents of the SPI flash (in this example, the executable file built with ext_RamImage SoftConsole project) to the external SRAM and executes as explained in the above flow chart. This function is a part of the main.c file that is available at:

../ext_mem_load_boot/main.c.

Booting from NOR Flash Memory

This section describes copying the vector table from the NOR flash to the MSS eSRAM, branching to the user boot code, and booting from the NOR flash. Figure 4 illustrates the boot process.

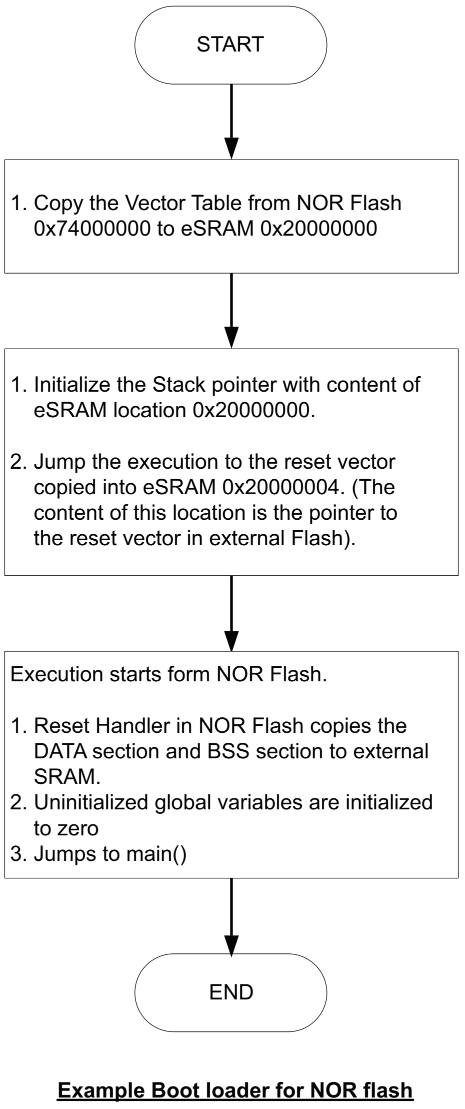

START
1. Copy the Vector Table from NOR Flash 0x74000000 to eSRAM 0x20000000
1. Initialize the Stack pointer with content of eSRAM location 0x20000000.
2. Jump the execution to the reset vector copied into eSRAM 0x20000004. (The content of this location is the pointer to the reset vector in external Flash).
Execution starts form NOR Flash.
1. Reset Handler in NOR Flash copies the DATA section and BSS section to external SRAM.
2. Uninitialized global variables are initialized to zero
3. Jumps to main()
END
Example Boot loader for NOR flash

***Figure 4* • Flow Chart for Booting From the NOR Flash Memory**

The design example uses the following function to carry out the boot process.

static void emc_flash_boot(void):

This API executes as explained in the above flow chart. This function is a part of the main.c file that is available at:

../ext_mem_load_boot/main.c.

Microcontroller Subsystem (MSS) Configuration

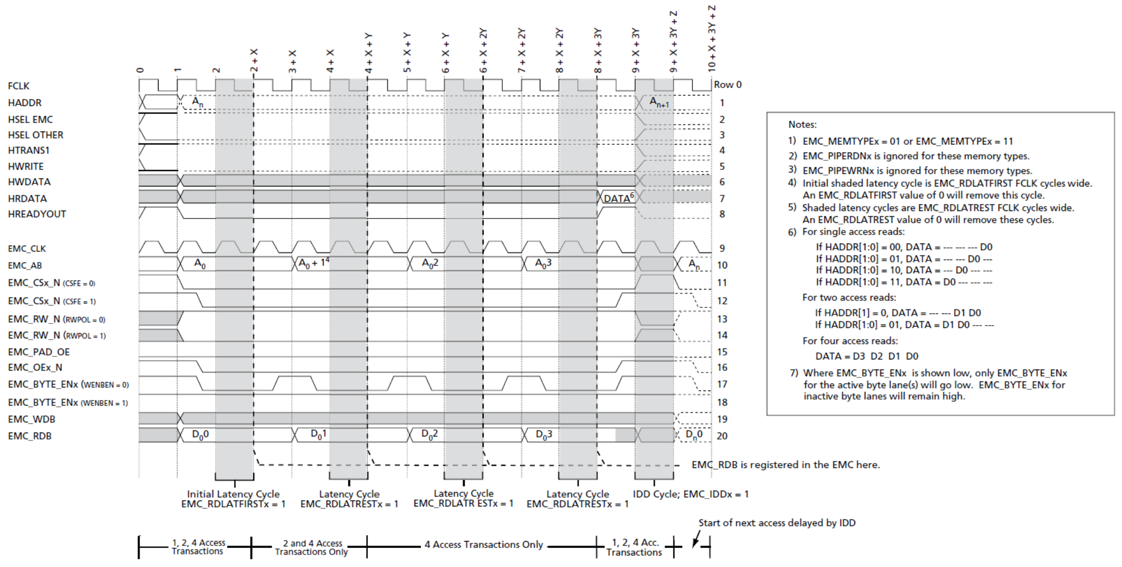
The MSS is configured with one SPI peripheral (SPI_1), one UART peripheral (UART_0), EMC, and three GPIOs. The clock conditioning circuit (CCC) in the MSS generates an 80 MHz clock and acts as a clock source for the peripherals. The GPIOs are configured as outputs and connected to test the LEDs in the SmartFusion Development Kit Board. The test LEDs D1, D2, and D3 are connected to the FPGA I/O B19, B20, and C19 respectively.

The EMC is divided into two regions: Region 0 and Region1. The MSS configuration tool provides a GUI to configure the memory type, port size, and latency for each region. The EMC read/write latency values can be configured as per the user requirements. The minimum latency values at different MSS clock frequencies for the SmartFusion Development Kit Board are given in Table 1.

***Table 1* • EMC Read/Write Latency Values**

Parameters	MSS Clock - 100 MHz		MSS Clock - 80 MHz	
	Asynchronous RAM	NOR flash	Asynchronous RAM	NOR flash
Read Latency for First Access (HCLK cycles)	1	5	1	4
Read Latency for Remaining Accesses (HCLK cycles)	1	1	1	1
Write Latency (HCLK cycles)	0	0	0	0

Figure 5 shows the timing diagram of EMC for Asynchronous Read. For more detail on read/write latencies, refer to the *SmartFusion Microcontroller Subsystem User's Guide*.


FCLK
HADDR
HSEL EMC
HSEL OTHER
HTRANS1
HWRITE
HWDATA
HRDATA
HREADYOUT
EMC_CLK
EMC_AB
EMC_CSx_N (CSFE = 0)
EMC_CSx_N (CSFE = 1)
EMC_RW_N (RWPOL = 0)
EMC_RW_N (RWPOL = 1)
EMC_PAD_OE
EMC_OEx_N
EMC_BYTE_ENx (WENBEN = 0)
EMC_BYTE_ENx (WENBEN = 1)
EMC_WDB
EMC_RDB
Row 0
DATA6
EMC_RDB is registered in the EMC here.
Initial Latency Cycle EMC_RDLATFIRSTx = 1
Latency Cycle EMC_RDLATRESTx = 1
IDD Cycle; EMC_IDDx = 1
Start of next access delayed by IDD
1, 2, 4 Access Transactions
2 and 4 Access Transactions Only
4 Access Transactions Only
1, 2, 4 Acc. Transactions

Notes:

1) EMC_MEMTYPEx = 01 or EMC_MEMTYPEx = 11
2) EMC_PIPERDNx is ignored for these memory types.
3) EMC_PIPEWRNx is ignored for these memory types.
4) Initial shaded latency cycle is EMC_RDLATFIRST FCLK cycles wide. An EMC_RDLATFIRST value of 0 will remove this cycle.
5) Shaded latency cycles are EMC_RDLATREST FCLK cycles wide. An EMC_RDLATREST value of 0 will remove these cycles.
6) For single access reads:

   If HADDR[1:0] = 00, DATA = --- --- --- D0
   If HADDR[1:0] = 01, DATA = --- --- D0 ---
   If HADDR[1:0] = 10, DATA = --- D0 --- ---
   If HADDR[1:0] = 11, DATA = D0 --- --- ---

   For two access reads:

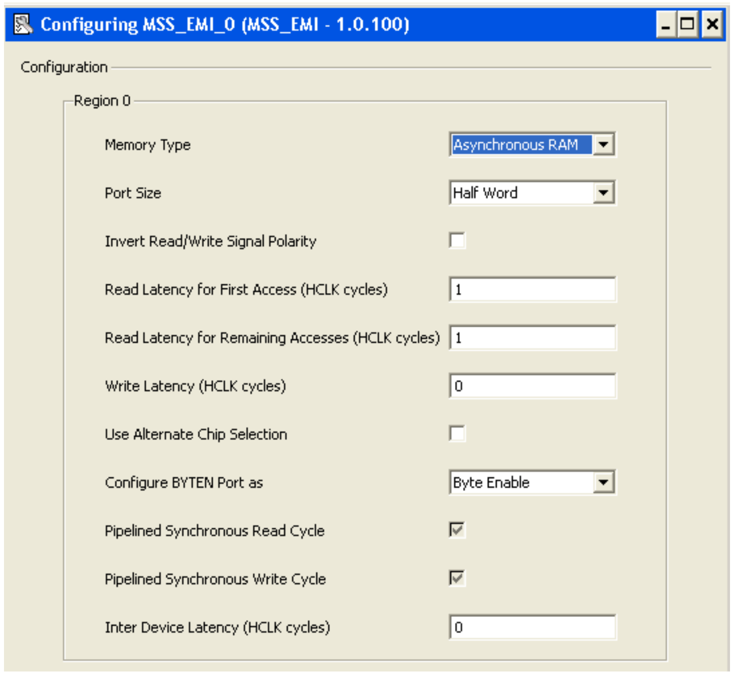
   If HADDR[1] = 0, DATA = --- --- D1 D0
   If HADDR[1:0] = 01, DATA = D1 D0 --- ---

   For four access reads:

   DATA = D3 D2 D1 D0
7) Where EMC_BYTE_ENx is shown low, only EMC_BYTE_ENx for the active byte lane(s) will go low. EMC_BYTE_ENx for inactive byte lanes will remain high.

***Figure 5* • Timing Diagram of EMC for Asynchronous Read**

Figure 6 shows the EMC configuration for Region 0 that is configured as Asynchronous RAM and port size as half word.



Configuring MSS_EMI_0 (MSS_EMI - 1.0.100)

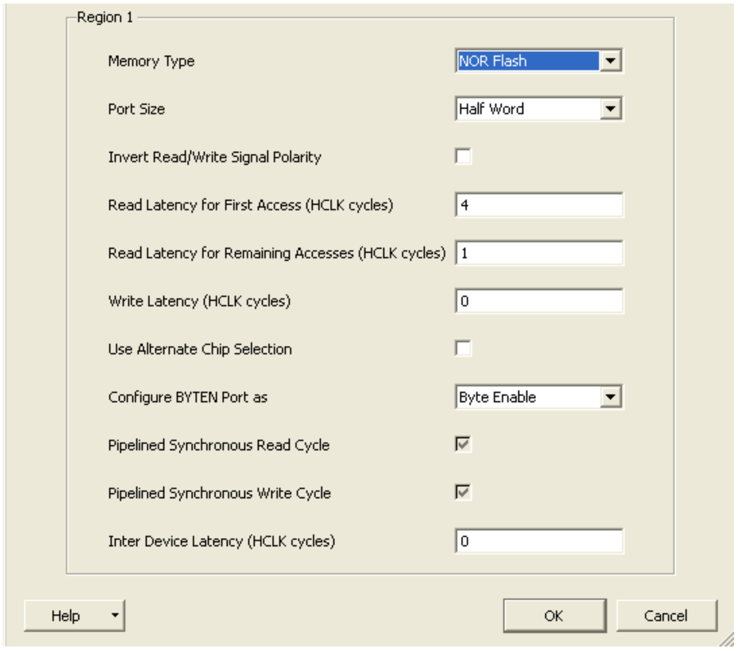
Configuration

Region 0

Memory Type	Asynchronous RAM
Port Size	Half Word
Invert Read/Write Signal Polarity	☐
Read Latency for First Access (HCLK cycles)	1
Read Latency for Remaining Accesses (HCLK cycles)	1
Write Latency (HCLK cycles)	0
Use Alternate Chip Selection	☐
Configure BYTEN Port as	Byte Enable
Pipelined Synchronous Read Cycle	✓
Pipelined Synchronous Write Cycle	✓
Inter Device Latency (HCLK cycles)	0

***Figure 6* • EMC Configuration for Region 0**

Similarly Figure 7 shows the EMC configuration for Region 1 that is configured as NOR flash and port size as half word. Refer to the *EMC Configuration User's Guide* for more details on EMC configuration.



Region 1

Memory Type	NOR Flash
Port Size	Half Word
Invert Read/Write Signal Polarity	☐
Read Latency for First Access (HCLK cycles)	4
Read Latency for Remaining Accesses (HCLK cycles)	1
Write Latency (HCLK cycles)	0
Use Alternate Chip Selection	☐
Configure BYTEN Port as	Byte Enable
Pipelined Synchronous Read Cycle	✓
Pipelined Synchronous Write Cycle	✓
Inter Device Latency (HCLK cycles)	0

***Figure 7* • EMC Configuration for Region 1**

Board Specific Settings

Table 2 gives the jumper settings needed to access the SRAM and NOR flash.

Table 2 • **Jumper Settings to Interface EMC with SRAM and NOR Flash**

Jumper	Pin	Pin
JP17	2	3
JP19	2	3
JP24	1	2
JP16	2	3

Refer to the *SmartFusion Development Kit User's Guide* for more details.

Running the Design

Program the SmartFusion Development Kit Board with the downloaded PDB file (refer to "Appendix A - Design Files" on page 14) using the FlashPro and then power cycle the board.

This design example on SoftConsole work space consists of the following three projects:

- emcFlashImage_MSS_CM3_0_app
- ext_RamImage
- ext_mem_load_boot

emcFlashImage_MSS_CM3_0_app

This project contains the software application to blink the LEDs continuously with certain delay while the UART prints the memory location of global variable on HyperTerminal. This project is built with the linker description file "production-execute-in-place-emc-flash.ld" to generate a bin file.

ext_RamImage

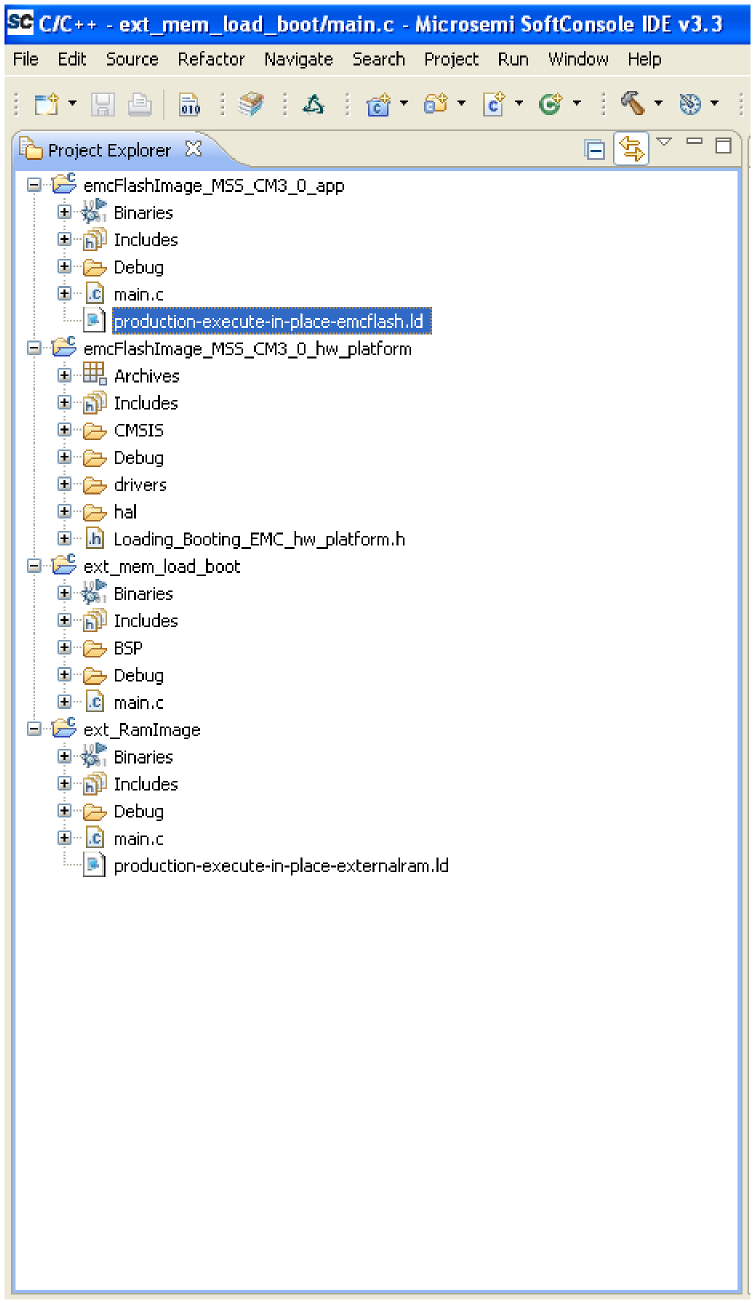
This project contains the software application to blink the LEDs continuously with certain delay while the UART prints the memory location of global variable on HyperTerminal. This project is built with the linker description file "production-execute-in-place-externalram.ld" to generate a bin file.

ext_mem_load_boot

This project is the target loader and bootloader for NOR flash and SPI flash. This project is used to load and execute (boot) the image from external NOR flash and SPI flash with the bin file generated by the *emcFlashImage_MSS_CM3_0_app* project and the *ext_RamImage* project. You need to run the debugger in the SoftConsole with the linker description file "debug-in-actel-smartfusion-esram.ld".

The linker description files "production-execute-in-place-emc-flash.ld" and "production-execute-in-place-externalram.ld" are specific to this design example and are provided in the design files.

Figure 8 shows the project explorer windows with three projects.


C/C++ - ext_mem_load_boot/main.c - Microsemi SoftConsole IDE v3.3
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer

```
emcFlashImage_MSS_CM3_0_app
    Binaries
    Includes
    Debug
    main.c
    production-execute-in-place-emcflash.ld
emcFlashImage_MSS_CM3_0_hw_platform
    Archives
    Includes
    CMSIS
    Debug
    drivers
    hal
    Loading_Booting_EMC_hw_platform.h
ext_mem_load_boot
    Binaries
    Includes
    BSP
    Debug
    main.c
ext_RamImage
    Binaries
    Includes
    Debug
    main.c
    production-execute-in-place-externalram.ld
```

***Figure 8* • Project Explorer View**

Loading and Booting from External NOR Flash Memory

Use the following steps to run the application in the SoftConsole:

1. Click on the project "emcFlashImage_MSS_CM3_0_app".
2. Build the project using the build option already set in the project. The *emcFlashImage.bin* file is generated under the Debug folder.

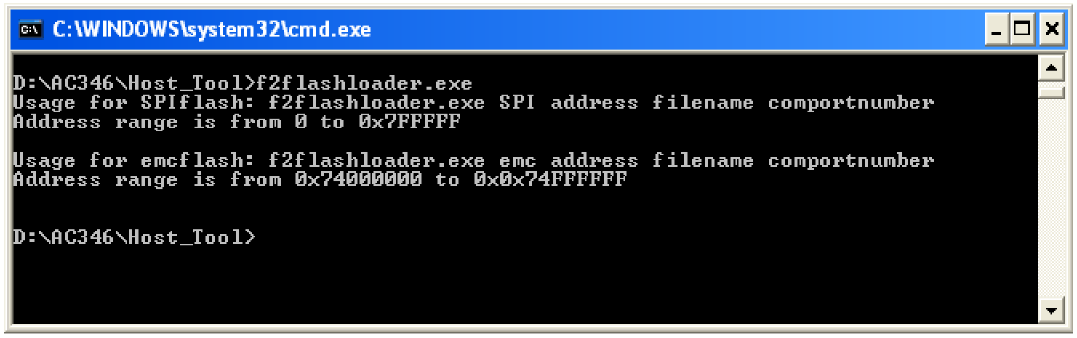
Note: If the *emcFlashImage.bin* file is not generated under the Debug folder, run the *Bin_File_Generator.bat* file given in the *emcFlashImage_MSS_CM3_0_app* project folder to generate the *emcFlashImage.bin* file.
You need to add the SoftConsole installation path, for example, C:\Microsemi\Libero_v10.0\SoftConsole\Sourcery-G++\bin, to the 'Environment Variables' before invoking the batch file.

Now the executable bin file for loading and booting from the external NOR flash is generated.

The following steps demonstrate the loading and execution of the above image from EMC NOR flash:

1. Click on the "ext_mem_load_boot" project and then build the project using the build options provided in the project.
2. Launch the debugger and run the project ext_mem_load_boot.
3. This step loads the executable bin file (emcFlashImage.bin) from the host PC to NOR Flash. Run the host loader tool in the command prompt. Before running the host loader tool, make sure that the COM port is not used by any other application like HyperTerminal or PuTTY.

   Open a command prompt window in the development PC and change to the directory where the host tool is located (refer to "Appendix A - Design Files" on page 14) and then type *f2flashloader.exe*. It prints the help on how to load the bin file generated by the "emcFlashImage_MSS_CM3_0_app" project and "ext_RamImage" project. Figure 9 shows the host loader (Flash loader) help.



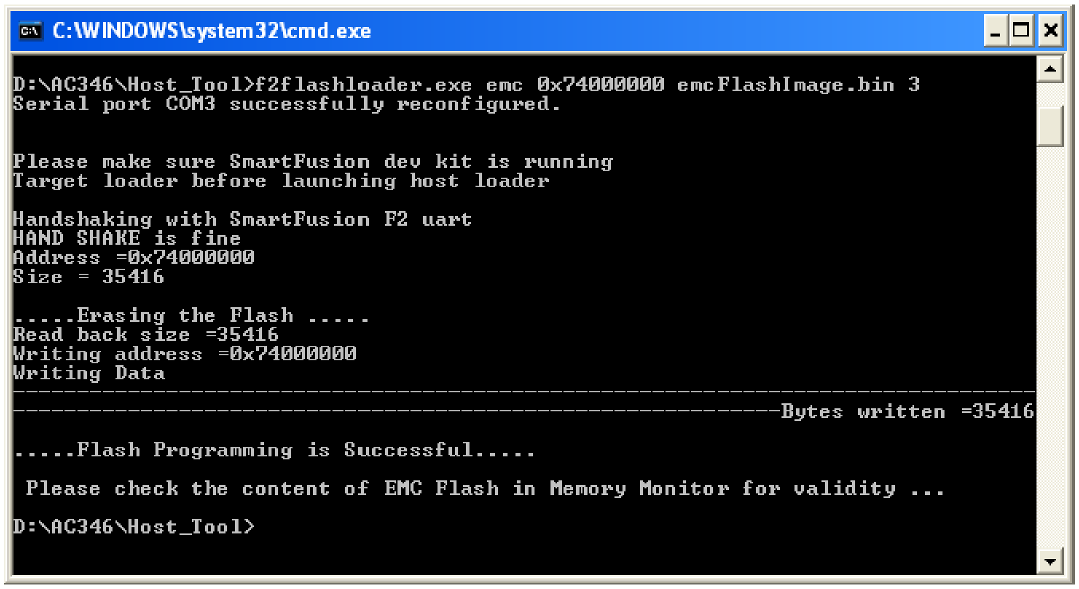
```
D:\AC346\Host_Tool>f2flashloader.exe
Usage for SPIflash: f2flashloader.exe SPI address filename comportnumber
Address range is from 0 to 0x7FFFFF

Usage for emcflash: f2flashloader.exe emc address filename comportnumber
Address range is from 0x74000000 to 0x0x74FFFFFF


D:\AC346\Host_Tool>
```

***Figure 9* • Flash Loader**

4. If the COM port number is 3 and the address range where you are writing is from 0x74000000, then the command is:

```
> f2flashloader.exe    emc    0x74000000    emcFlashImage.bin    3
```

The *emcFlashImage.bin* file is written into the external NOR flash. Figure 10 shows the NOR flash programming.

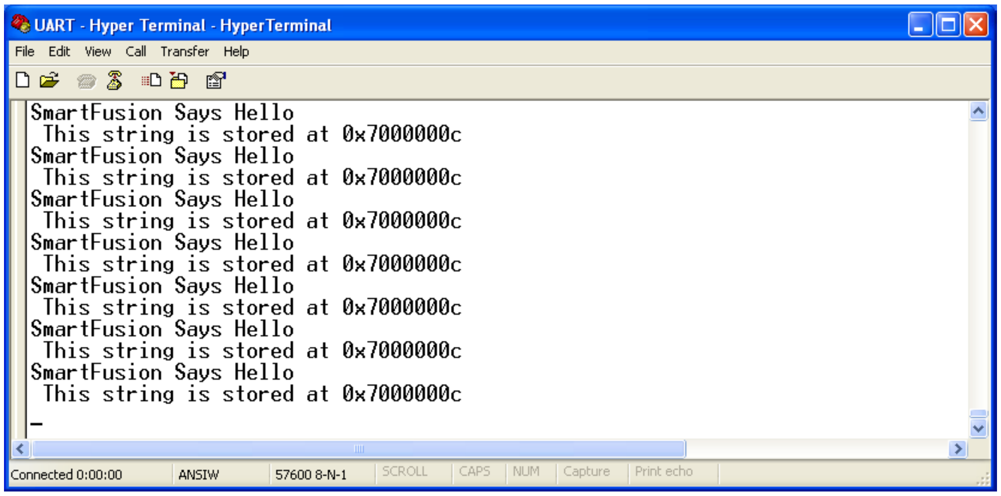


```
D:\AC346\Host_Tool>f2flashloader.exe emc 0x74000000 emcFlashImage.bin 3
Serial port COM3 successfully reconfigured.

Please make sure SmartFusion dev kit is running
Target loader before launching host loader

Handshaking with SmartFusion F2 uart
HAND SHAKE is fine
Address =0x74000000
Size = 35416

.....Erasing the Flash .....
Read back size =35416
Writing address =0x74000000
Writing Data
--------------------------------------------------------------------------------
------------------------------------------------------------Bytes written =35416

.....Flash Programming is Successful.....

 Please check the content of EMC Flash in Memory Monitor for validity ...

D:\AC346\Host_Tool>
```

Figure 10 **• NOR Flash Programming**

Once the *emcFlashImage.bin* file is written into the external NOR flash memory, the test LEDs D1, D2, and D3 in the SmartFusion Development Kit Board start blinking.

5. Start HyperTerminal or PuTTY with the baud rate set to 57600, 8 bits data, 1 stop bit, no parity, and no flow control. The application starts printing the memory location of global variable on the HyperTerminal. Figure 11 shows the screen shot of HyperTerminal with the memory location of global variable.


UART - Hyper Terminal - HyperTerminal
File Edit View Call Transfer Help
Connected 0:00:00
ANSIW
57600 8-N-1
SCROLL
CAPS
NUM
Capture
Print echo

```
SmartFusion Says Hello
 This string is stored at 0x7000000c
SmartFusion Says Hello
 This string is stored at 0x7000000c
SmartFusion Says Hello
 This string is stored at 0x7000000c
SmartFusion Says Hello
 This string is stored at 0x7000000c
SmartFusion Says Hello
 This string is stored at 0x7000000c
SmartFusion Says Hello
 This string is stored at 0x7000000c
SmartFusion Says Hello
 This string is stored at 0x7000000c
_
```

Figure 11 **• Screen-shot of HyperTerminal with Memory Location of Global Variable**

Loading the Application into the SPI Flash Memory and Booting from External SRAM

Use the following steps to run the application in the SoftConsole:

1. Click on the project "ext_RamImage".
2. Build the project using the build option already set in the project. The *externalRAMImage.bin* file is generated under the Debug folder.

Note: If the *externalRAMImage.bin* file is not generated under the Debug folder, run the *Bin_File_Generator.bat* file given in the *ext_RamImage* project folder to generate the *externalRAMImage.bin* file. You need to add the SoftConsole installation path, for example, C:\Microsemi\Libero_v10.0\SoftConsole\Sourcery-G++\bin, to the 'Environment Variables' before invoking the batch file.

Now the executable bin file for loading into the SPI flash memory and booting from the external SRAM is generated. The following steps demonstrate the loading into the SPI flash memory and booting from the external SRAM:

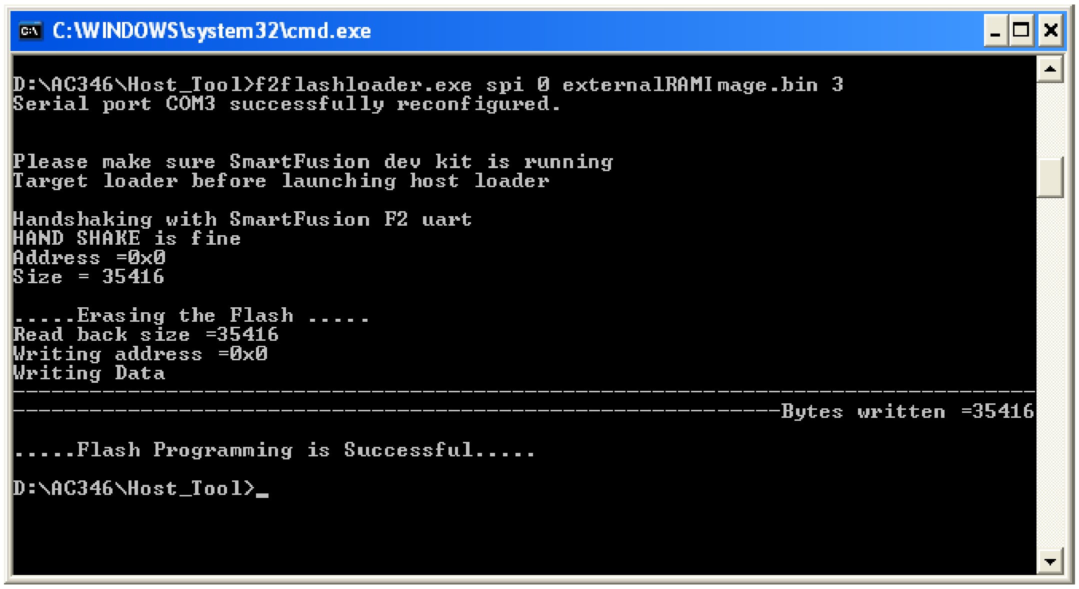
1. Click on the "ext_mem_load_boot" project and then build the project using the build options provided in the project.
2. Launch the debugger and run the project *ext_mem_load_boot*.
3. Load the executable bin file (externalRAMImage.bin) from the host PC to the SPI Flash. Run the host loader tool in the command prompt. Before running the host loader tool ensure that the COM port is not being used by any other application like HyperTerminal or PuTTY.

   Open the command prompt window in the development PC and change to the directory where the host tool is located (refer to "Appendix A - Design Files" on page 14) and then type f2flashloader.exe. It prints the help on how to load the bin file generated by the "emcFlashImage_MSS_CM3_0_app" project and "ext_RamImage" project.

   Enter the following command to load the Image created from the "ext_RamImage" project to the SPI flash, at start location:

   ```
   > f2flashloader.exe   spi   0   externalRAMImage.bin   3
   ```

   The *externalRAMImage.bin* file is written into the SPI flash. Figure 12 shows the SPI flash programming.

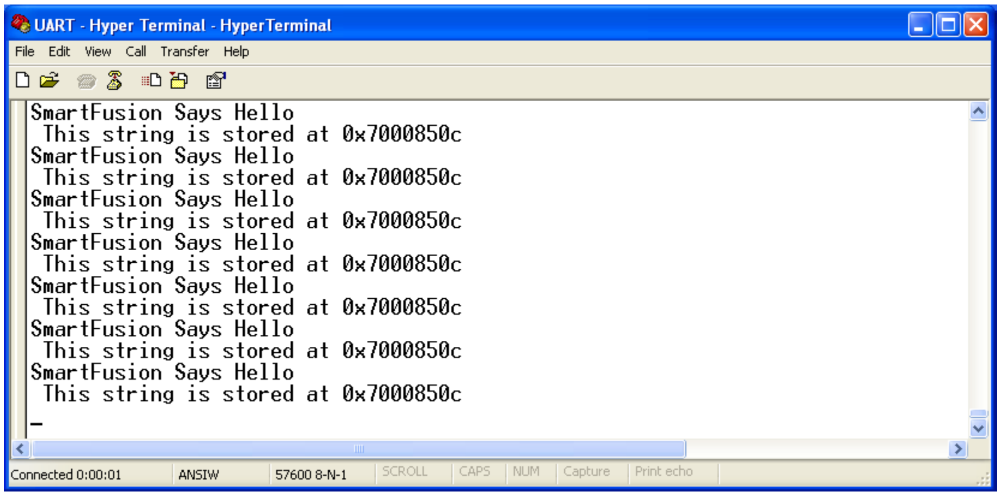


```
D:\AC346\Host_Tool>f2flashloader.exe spi 0 externalRAMImage.bin 3
Serial port COM3 successfully reconfigured.


Please make sure SmartFusion dev kit is running
Target loader before launching host loader

Handshaking with SmartFusion F2 uart
HAND SHAKE is fine
Address =0x0
Size = 35416

.....Erasing the Flash .....
Read back size =35416
Writing address =0x0
Writing Data
--------------------------------------------------------------------------------
-------------------------------------------------------------Bytes written =35416

.....Flash Programming is Successful.....

D:\AC346\Host_Tool>_
```

Figure 12 **• SPI Flash Programming**

Once the *externalRAMImage.bin* file is written into the SPI flash memory, the test LEDs D1, D2, and D3 in the SmartFusion Development Kit Board start blinking.

4. Start HyperTerminal with the baud rate set to 57600, 8 bits data, 1 stop bit, no parity, and no flow control. The application then starts printing the memory location of the global variable on HyperTerminal. Figure 13 shows the screen shot of HyperTerminal with the memory location of the global variable.


UART - Hyper Terminal - HyperTerminal
File Edit View Call Transfer Help

```
SmartFusion Says Hello
 This string is stored at 0x7000850c
SmartFusion Says Hello
 This string is stored at 0x7000850c
SmartFusion Says Hello
 This string is stored at 0x7000850c
SmartFusion Says Hello
 This string is stored at 0x7000850c
SmartFusion Says Hello
 This string is stored at 0x7000850c
SmartFusion Says Hello
 This string is stored at 0x7000850c
SmartFusion Says Hello
 This string is stored at 0x7000850c
_
```

Connected 0:00:01 ANSIW 57600 8-N-1 SCROLL CAPS NUM Capture Print echo

***Figure 13* • Screen-shot of HyperTerminal with Memory Location of Global Variable**

Release Mode

The release mode programming file (STAPL) is also provided. Refer to the Readme.txt file included in the programming zip file for more information.

Refer to the *Building Executable Image in the Release Mode and Loading into eNVM Tutorial* for more information on building an application in the release mode.

Conclusion

The design example demonstrates the loading and booting the application code from the external memories (SPI Flash and EMC NOR Flash) on the SmartFusion Development Kit Board. This document also describes the usage of the NOR flash driver APIs, SPI flash driver APIs, Flash Loader, and Linker description files for creating the images for external memories.

Appendix A - Design Files

You can download the design files from the Microsemi SoC Products Group website:

www.microsemi.com/soc/download/rsc/?f=A2F_AC346_DF.

The design zip file consists of Libero System-on-Chip (cSoC) projects and programming file (*.stp) for A2F500 and A2F200. Refer to the Readme.txt file included in the design file for directory structure and description.

You can download the programming files (*.stp) in release mode from the Microsemi SoC Products Group website: www.microsemi.com/soc/download/rsc/?f=A2F_AC346_PF.

The programming zip file consists of STAPL programming files (*.stp) for A2F500, A2F200, and a Readme.txt file.

Appendix B - Linker Description Files

Linker Description Files to Execute from External SRAM

Memory region

```
MEMORY
{
  /* SmartFusion external EMC RAM */
  ram (rwx) : ORIGIN = 0x70000000, LENGTH = 2M
}
```

Vector table and Init code

```
.reset :
  {
    *(.isr_vector)
    *sys_boot.o(.text)
    . = ALIGN(0x4);
  } >ram
```

Data

```
.data :
  {
    __data_load = LOADADDR(.data);
    _sidata = LOADADDR (.data);
    __data_start = .;
    _sdata = .;
    KEEP(*(.jcr))
    *(.got.plt) *(.got)
    *(.shdata)
    *(.data .data.* .gnu.linkonce.d.*)
    . = ALIGN (4);
   _edata = .;
  } >ram
```

BSS

```
.bss :
  {
    __bss_start__ = . ;
    _sbss = .;
    *(.shbss)
    *(.bss .bss.* .gnu.linkonce.b.*)
    *(COMMON)
    . = ALIGN (8);
    __bss_end__ = .;
    _end = .;
    __end = _end;
    _ebss = .;
    PROVIDE(end = .);
  } >ram
```

Instructions

```
.text :
  { CREATE_OBJECT_SYMBOLS
    __text_load = LOADADDR(.text);
    __text_start = .;
    *(.text .text.* .gnu.linkonce.t.*)
    *(.plt)
    *(.gnu.warning)
    *(.glue_7t) *(.glue_7) *(.vfp11_veneer)
    . = ALIGN(0x4);
    /* These are for running static constructors and destructors under ELF.  */
    KEEP (*crtbegin.o(.ctors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*crtend.o(.ctors))
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*crtend.o(.dtors))
    *(.rodata .rodata.* .gnu.linkonce.r.*)
    *(.ARM.extab* .gnu.linkonce.armextab.*)
    *(.gcc_except_table)
    *(.eh_frame_hdr)
    *(.eh_frame)
    KEEP (*(.init))
    KEEP (*(.fini))
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array))
    PROVIDE_HIDDEN (__preinit_array_end = .);
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP (*(SORT(.init_array.*)))
    KEEP (*(.init_array))
    PROVIDE_HIDDEN (__init_array_end = .);
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*(.fini_array))
    KEEP (*(SORT(.fini_array.*)))
    PROVIDE_HIDDEN (__fini_array_end = .);
  } >ram
  /* .ARM.exidx is sorted, so has to go in its own output section.  */
   __exidx_start = .;
  .ARM.exidx :
  {
    *(.ARM.exidx* .gnu.linkonce.armexidx.*)
  } >ram
  __exidx_end = .;
  _etext = .;
```

Linker Descriptions File for Execute-In-Place from External NOR Flash

Memory region

```
MEMORY
{ /* SmartFusion external EMC Flash */
  rom (rx) : ORIGIN = 0x74000000, LENGTH = 16M
/* SmartFusion external emc RAM */
  ram (rwx) : ORIGIN = 0x70000000, LENGTH = 2M
}
```

Vector table and Init code

```
.reset :
  {
    *(.isr_vector)
    *sys_boot.o(.text)
    . = ALIGN(0x4);
  } >rom
```

Data

```
.data :
  {
    __data_load = LOADADDR(.data);
    _sidata = LOADADDR (.data);
    __data_start = .;
    _sdata = .;
    KEEP(*(.jcr))
    *(.got.plt) *(.got)
    *(.shdata)
    *(.data .data.* .gnu.linkonce.d.*)
    . = ALIGN (4);
   _edata = .;
  } >ram AT>rom
```

BSS

```
.bss :
  {
    __bss_start__ = . ;
    _sbss = .;
    *(.shbss)
    *(.bss .bss.* .gnu.linkonce.b.*)
    *(COMMON)
    . = ALIGN (8);
    __bss_end__ = .;
    _end = .;
    __end = _end;
    _ebss = .;
    PROVIDE(end = .);
  } >ram AT>rom
```

Instructions

```
.text :
  {
    CREATE_OBJECT_SYMBOLS
    __text_load = LOADADDR(.text);
    __text_start = .;

    *(.text .text.* .gnu.linkonce.t.*)
    *(.plt)
    *(.gnu.warning)
    *(.glue_7t) *(.glue_7) *(.vfp11_veneer)

    . = ALIGN(0x4);
    /* These are for running static constructors and destructors under ELF.  */
    KEEP (*crtbegin.o(.ctors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*crtend.o(.ctors))
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*crtend.o(.dtors))

    *(.rodata .rodata.* .gnu.linkonce.r.*)

    *(.ARM.extab* .gnu.linkonce.armextab.*)
    *(.gcc_except_table)
    *(.eh_frame_hdr)
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array))
    PROVIDE_HIDDEN (__preinit_array_end = .);
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP (*(SORT(.init_array.*)))
    KEEP (*(.init_array))
    PROVIDE_HIDDEN (__init_array_end = .);
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*(.fini_array))
    KEEP (*(SORT(.fini_array.*)))
    PROVIDE_HIDDEN (__fini_array_end = .);
  } >rom
  /* .ARM.exidx is sorted, so has to go in its own output section.  */
   __exidx_start = .;
  .ARM.exidx :
  {
    *(.ARM.exidx* .gnu.linkonce.armexidx.*)
  } >rom
  __exidx_end = .;
  _etext = .;
```

Appendix C - NOR Flash Driver Application Programming Interfaces (APIs)

This section describes the software driver APIs used in this design to carry out the transactions with the NOR Flash. These drivers are included in the design files with this design example.

Function Description of Data Structures

Enum : emc_flash_status_t

This enum represents the status of the different APIs like *emc_flash_write*, *emc_flash_read*, *emc_flash_chip_erase*, and *emc_flash_block_erase*.

```
typedef enum {
    NOR_SUCCESS = 0,
    NOR_BLOCK_LOCK_ERROR = 0x2,
    NOR_PROGRAM_SUSPEND = 0x4,
    NOR_GENERAL_ERROR = 0x08,
    NOR_PROGRAM_ERROR = 0x10,
    NOR_ERASE_ERROR   = 0x20,
    NOR_CMD_SEQ_ERROR = 0x30,
    NOR_ERASE_SUSPEND = 0x40,
    NOR_INVALID_ARGUMENTS,
    NOR_INVALID_ADDRESS,
    NOR_UNSUCCESS};
```

Function Description of API

emc_Init()

This function initializes the EMC controller with proper wait states for read and write access.

For example:

```
emc_init();
```

emc_flash_chip_erase (uint32_t start_addr)

This function erases the content of the external NOR flash from the address provided to this function till the last address of the NOR flash. For example:

```
emc_flash_chip_erase(0x74000000);
```

*emc_flash_block_erase (uint16_t *blockAddr)*

This function erases the content of a particular block of the external NOR flash based on the address provided to this function. For example:

```
emc_flash_block_erase (0x74000000)
```

*emc_flash_read(uint32_t start_addr, uint8_t * p_data, size_t nb_bytes)*

This function reads the content of the SmartFusion EMC external NOR Flash. The data is read from the memory location specified by the first parameter. This address is the absolute address in the processor's memory space at which the external flash is located.

- @param start_addr: This is the address at which data is read. This address is the absolute address in the processor's memory space at which the external flash is located.
- @param p_data: This is a pointer to the buffer for holding the read data.
- @param nb_bytes: This is the number of bytes to be read from the external flash.
- @return: The return value indicates if the read was successful. The possible values are:
  – NOR_SUCCESS: Describes that the NOR Flash operation is correct and complete
  – NOR_INVALID_ADDRESS: Describes that the function has received invalid address
  – NOR_UNSUCCESS: Describes that the NOR Flash operation is incomplete

For example:

```
status = emc_flash_read(0x74000000 + ii, output_buffer, length);
```

*emc_flash_write (uint32_t start_addr,const uint8_t * p_data, size_t nb_bytes)*

This function writes the content of the buffer passed as parameter to the external NOR Flash. The data is written from the memory location specified by the first parameter. This address is the absolute address in the processor's memory space at which the External Flash is located.

- @param start_addr: This is the address at which the data is written. This address is the absolute address in the processor memory space at which the External Flash is located.
- @param p_data: This is a pointer to the buffer holding the data to be written into the External Flash.
- @param nb_bytes: This is the number of bytes to be written into the External Flash.
- @return: The return value indicates if the write was successful. The possible values are:
  – NOR_SUCCESS: Describes the NOR Flash operation is correct and complete
  – NOR_INVALID_ADDRESS:Describes that function has received invalid address
  – NOR_UNSUCCESS: Describes the NOR Flash operation is incomplete

For example:

```
status = emc_flash_write(0x74000000 + ii, input_buffer, length);
```

List of Changes

The following table lists critical changes that were made in each revision of the document.

Revision*	Changes	Page
Revision 3 (February 2012)	Removed ".zip" extension in the link (SAR 36763).	14
Revision 2 (January 2012)	Changed Figure 8 (SAR 36034).	10
	Changed Figure 9 (SAR 36034).	11
	Changed Figure 10 (SAR 36034).	12
	Changed Figure 12 (SAR 36034).	13
	Added the section "Release Mode" (SAR 36034).	14
	Modified the section "Appendix A - Design Files" (SAR 36034).	14
Revision 1 (August 2010)	Modified the section "Running the Design" (SAR 27472).	9
	Modified the section "Loading and Booting from External NOR Flash Memory" (SAR 27472)	11
	Modified the section "Appendix A - Design Files" (SAR 27472)	14
	Removed Table 3.	

Note: **The revision number is located in the part number after the hyphen. The part number is displayed at the bottom of the last page of the document. The digits following the slash indicate the month and year of publication.*

51900214-3/02.12