

SmartFusion cSoC: Basic Bootloader and Field Upgrade eNVM Through IAP Interface

Table of Contents

Introduction	1
Introduction to Field Upgrade Using IAP	2
Design Example Overview	4
Design Description	5
IAP Programming: eNVM Update in Release Mode Using the IAP Through UART	7
Running the Design	13
Conclusion	19
Appendix A	19
Appendix B	19
List of Changes	26

Introduction

The SmartFusion[®] customizable system-on-chip (cSoC) FPGAs devices integrate FPGA technology with hardened ARM[®] Cortex[™]-M3 processor based microcontroller subsystem (MSS) and programmable high-performance analog blocks built on a low power flash semiconductor process. The MSS consists of hardened blocks, such as a 100 MHz ARM Cortex-M3 processor, peripheral DMA (PDMA), embedded nonvolatile memory (eNVM), embedded SRAM (eSRAM), embedded FlashROM (eFROM), external memory controller (EMC), Watchdog Timer, Phillips Inter-Integrated circuit (I²C), SPI, 10/100 Ethernet controller, real-time counter (RTC), general purpose input output (GPIO) block, fabric interface controller (FIC), in-application programming (IAP), and system registers. The programmable analog block contains the analog compute engine (ACE) and analog front-end (AFE) consisting of ADCs, DACs, active bipolar prescalers (ABPS), comparators, current monitors, and temperature monitor circuitry.

The IAP block is the portion of the MSS that interfaces with the Cortex-M3 processor through the APB bus. The IAP block provides the hardware capability to program the flash components of the SmartFusion cSoC device within the programmed end users application. The flash components are: an FPGA array, the MSS eNVM, and the FlashROM.

Re-programming the eNVM blocks (field upgrading the firmware/software) using the Cortex-M3 processor is achieved by executing the eNVM programming algorithm from eSRAM. Since individual pages (132 bytes) of the eNVM can be write-protected, the programming algorithm software can be protected from inadvertent erasure. When reprogramming the eNVM, both MSS I/Os and FPGA I/Os are available as interfaces for sourcing the new eNVM image.

The executable image for Cortex-M3 processor can be partitioned across the following memories of the SmartFusion cSoC device:

1. Internal Memories: eNVM, eSRAM, and fabric RAM
2. External memories: Connected through EMC interface (SRAM, PSRAM, SSRAM, and NOR flash)

Based on the application's requirement and various combinations of the above memories, the executable application can be created. Linker script is used by the linker to create the required image. This application introduces the usage of the linker script generator tool for the SmartFusion cSoC Cortex-M3 processor executable.

This application note demonstrates the following using the SmartFusion cSoC:

1. A basic bootloader for the SmartFusion cSoC device has multiple images in the eNVM with different memory maps. It boots the required image as per the need. The bootloader should be designed and implemented as a black box. For this you need to parse the complete *.elf files that are loaded or to be loaded in eNVM/EMC flash, which requires more memory and booting time. The bootloader proposed in this application note is not a full-fledged bootloader, but is used as a reference example to boot multiple images.
2. Making a complete release mode image to run from eSRAM. This image is stored in the eNVM and the bootloader relocates this image if it is selected for execution. This is an example of using the eSRAM as a complete executable memory and booting using the bootloader.
3. Performing the IAP programming to update the partial or complete eNVM in release mode. This requirement is accomplished using the above two steps.

Introduction to Field Upgrade Using IAP

The IAP interface in the SmartFusion cSoC device is used by the Cortex-M3 processor to perform the upgrade or programming of eNVM and/or FPGA fabric. Figure 1 shows a typical SmartFusion cSoC device configuration customized for performing the IAP.

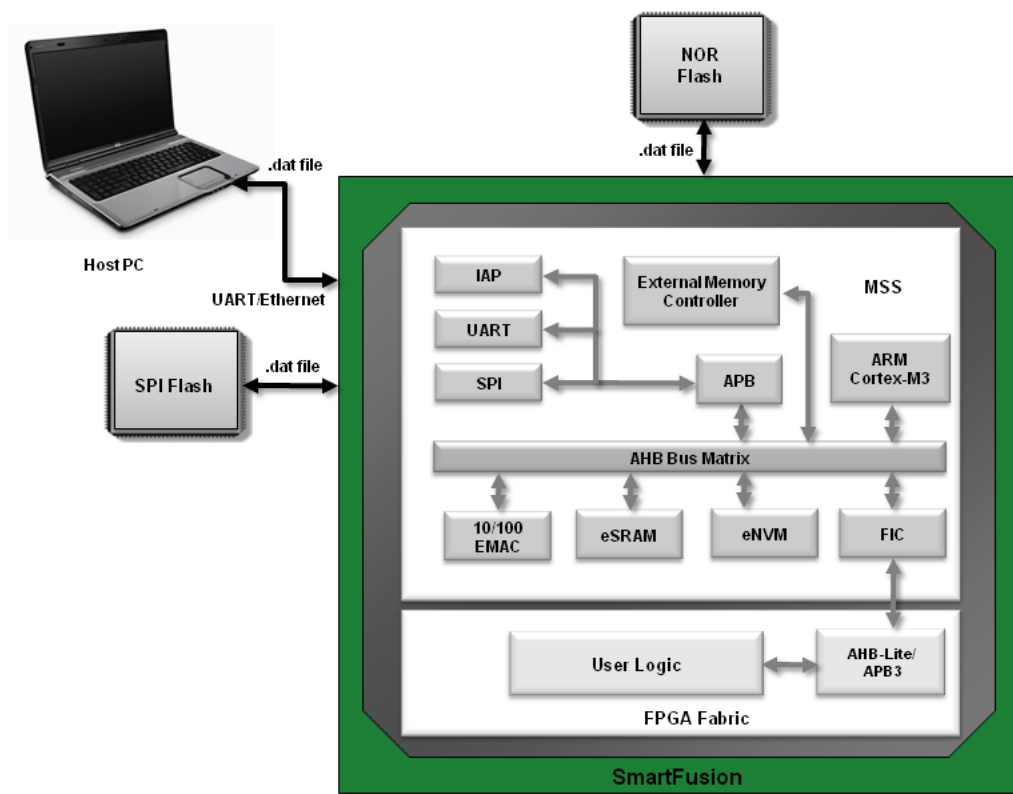


Figure 1 • Block Diagram for the IAP Programming for eNVM and/or FPGA Fabric

The IAP interface is a part of the digital subsystem which is an APB peripheral to the Cortex-M3 processor. This peripheral block is a combination of both hardware and firmware that provides low level interface to program the flash component of the SmartFusion cSoC device through the control and status registry.

This block performs the following:

- Mimic the JTAG signals through the existing test access ports
- Manage the data stream communication from the source to the IAP block

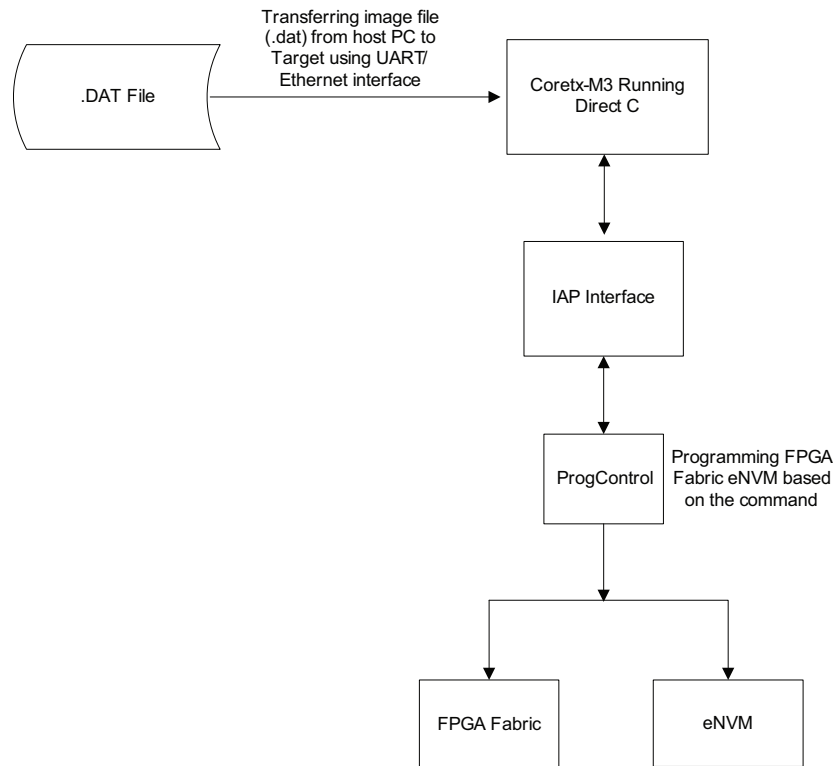


Figure 2 • IAP Data Flow Block Diagram

In release mode, while updating the eNVM content, the programming code (IAP Programmer) should not reside on the eNVM as it cannot be reprogrammed and accessed simultaneously.

For this reason we need to design a multi-boot system with the eNVM partitioned into multiple regions as follows:

- Bootloader/eNVM Region 1: It provides the option to select the IAP Programmer (eNVM Region 2) or application code (eNVM Region 3) and boot the selected executable image.
- IAP Programmer/eNVM Region 2: It contains the IAP executable image that is built to run completely from the eSRAM in release mode. The bootloader copies this partition to eSRAM if you have selected IAP programming.
- Application/eNVM Region 3: The IAP Programmer running from eSRAM programs this region using the *.dat file provided by you. This program is launched by either the IAP Programmer after programming the *.dat file into this region successfully, or by the bootloader if this has been programmed earlier based on your selection. [Figure 3](#) illustrates the partitioning of the eNVM for mutually exclusive executable images.

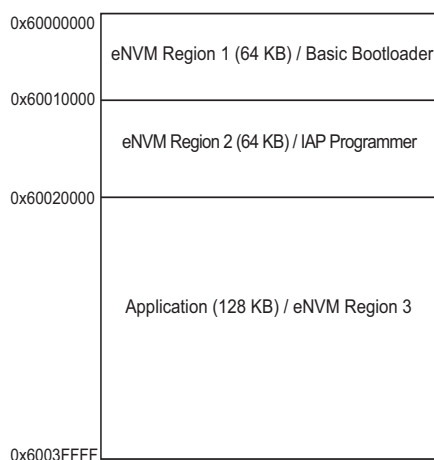


Figure 3 • Example of Partitioning the eNVM for Mutually Exclusive Executable Images

The memory regions shown in [Figure 3](#) are used in this application note as a example memory map. There is no specific rule for the memory maps for the applications. The memory maps explained in this application note are just for reference. As per the applications memory, these requirements can be changed and the corresponding linker scripts can be generated.

The simple approach for understanding these memory requirements for these applications, you can use the partitions explained in this application note as it is and generate the linker scripts. While compiling the projects, the compiler notifies you if the memory regions are not sufficient, if so, adjust the partitions accordingly. There are still possibilities that the compiler generates code that may have runtime errors like heap and stack collisions etc. In such cases the data regions and stack sizes have to be adjusted as per the applications run time memory requirements.

Design Example Overview

The design example consists of the following mutually exclusive executable images that reside on eNVM.

1. Bootloader
2. IAP Programmer
3. Example application

Both eNVM and eSRAM are partitioned into multiple regions to run this multi-boot system design. Each design uses different linker scripts with different memory maps. A linker script generator tool is used to generate different linker script files with different memory maps as listed below:

1. Bootloader
 - 64 KB of eNVM for interrupt vector table, instructions, and constant data sections
 - 4 KB of eSRAM for stack, heap, and data sections
2. IAP Programmer completely running from eSRAM
 - 60 KB of eSRAM for all the sections of an executable image
3. Simple application image
 - 128 KB of eNVM for interrupt vector table, instructions, and constant data sections
 - 64 KB of eSRAM for stack, heap, and data sections

The reason for creating the second image that will run exclusively from the eSRAM is to run an executable image independent of eNVM in a release mode or end product. This is required to use an IAP interface to update partial or full eNVM in release mode. You cannot update the eNVM while the IAP Programmer is running from eNVM.

Design Description

Bootloader

The bootloader performs the function of relocating or loading (copying the different sections of image to their executable location from loaded location) and making other executable images to run or debug. The bootloader needs the inputs related to the starting address of the executable image residing in the system memory map.

As explained in the [Figure 3 on page 4](#), we will be having two images in eNVM apart from the bootloader from which you can provide the input about the application that has to be executed by the bootloader. The Bootloader is also provided with the timeout feature in which if you do not select any input for booting within the timeout period, the bootloader boots the default sample application image from Region 3. In the design files provided for this application note, the boot time parameter is provided in the `boot_config.h` file. Timeout can vary by changing this parameter as per the applications requirements.

When the image is selected, the bootloader initializes the Cortex-M3 processor stack pointer and PC to the new image's reset handler. Once the image or sections of the image have been loaded to appropriate memories, the bootloader performs the steps explained in [Figure 4 on page 6](#) to branch to the newly loaded image execution.

In this basic bootloader implementation, the executable image runs from the eSRAM and the bootloader copies the entire image of 60 KB size from eNVM Region 2 to the bottom of the eSRAM.

The executable images residing in eNVM Region 1 (bootloader) and eNVM Region 3 (sample application) are created to be executed in place from the eNVM and hence no copying is required.

The reset handler of each image takes care of the executable image section replacements in all cases. If all the images are created to be executed in place of eNVM, then there is no requirement for copying the image from one location to other. In this particular example, we need to create an IAP Programmer image to be executed completely from eSRAM and hence in the bootloader implementation, copying logic to copy from eNVM Region 2 to eSRAM is implemented.

```
*-----
* Branch to an application startup code: Theory of operation:
* Step 1:
*     Load the address of the vector table into R0.
* Step 2:
*     Load the application's initial stack pointer value into R1.
* Step 3:
*     Load the address of the Cortex M3 vector table Offset register into R2.
* Step 4:
*     Load the application's initial stack pointer to CortexM3 vector table
*     Offset registerSCB_VTOR.
* Step 5:
*     Load the applications's reset handler address into the Link Register.
*     The location at address R0 + 0x04 is the reset vector,
*     R0 being the base address of the vector table.
* Step 6:
*     Load the application's initial stack pointer to
*     the Main Stack Pointer register.
* Step 7:
*     Return from the function by branching to the Link Register. This causes
*     to branch to the application's startup code/Reset Handler.
*/
    .align 4
__user_code_esram:
    ldr r0,=VECTOR_TABLE_ADDR    /* Step 1. */
    ldr r1,[r0,#0x0]             /* Step 2. */
    ldr r2,=SCB_VTOR_ADDRESS     /* Step 3. */
    str r0, [r2]                 /* Step 4. */
    ldr lr,[r0,#0x4]             /* Step 5. */
    msp msp,r1                   /* Step 6. */
    bx lr                        /* Step 7. */
    .end
```

Figure 4 • Steps of the Bootloader to Jump to the New Image

Following is the flow chart of the example bootloader Implemented for this application note:

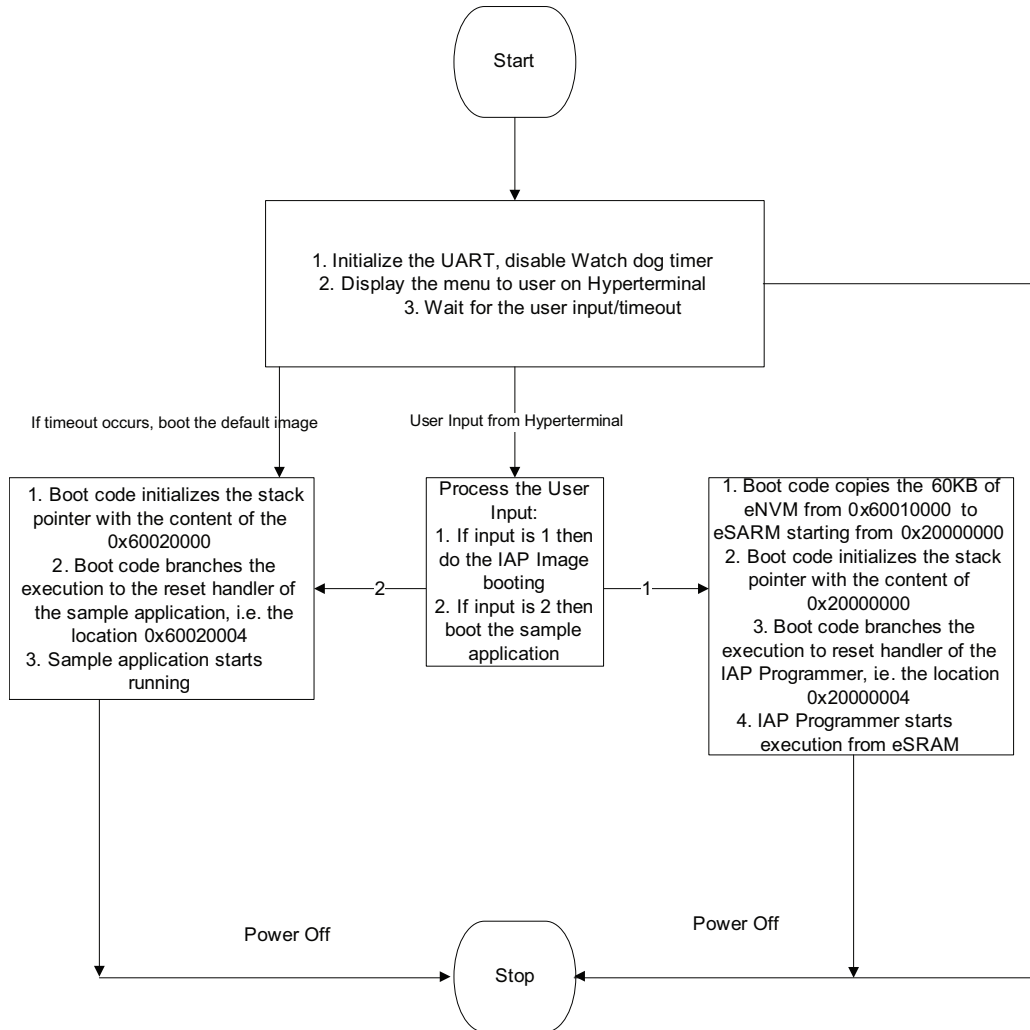


Figure 5 • Flow Chart of the Basic Bootloader Implemented

IAP Programming: eNVM Update in Release Mode Using the IAP Through UART

In release mode, while programming eNVM using the IAP interface, the sections of the programming code (IAP Programmer) should not reside on eNVM. This is because accessing eNVM simultaneously using the Cortex-M3 processor and IAP (JTAG) signals is not possible. Hence, we need to run the complete programming code from eSRAM or external memories in such a way that no part of the code or data of IAP programming is accessed by the Cortex-M3 processor from eNVM.

This application note explains the complete procedure on how this requirement can be met by designing and implementing using the multiple executable images. These executable images include the bootloader and IAP Programmer. The linker scripts for these executable images are generated by using the linker script generator tool for various memory map regions.

This application uses the host loader to get the programming file or *.dat file content from the host PC using the UART as the host interface between Host PC and the SmartFusion cSoC device. [Figure 6 on page 8](#) displays the flow chart of the complete IAP programming for the eNVM by getting the *.dat file or programming file from the host PC.

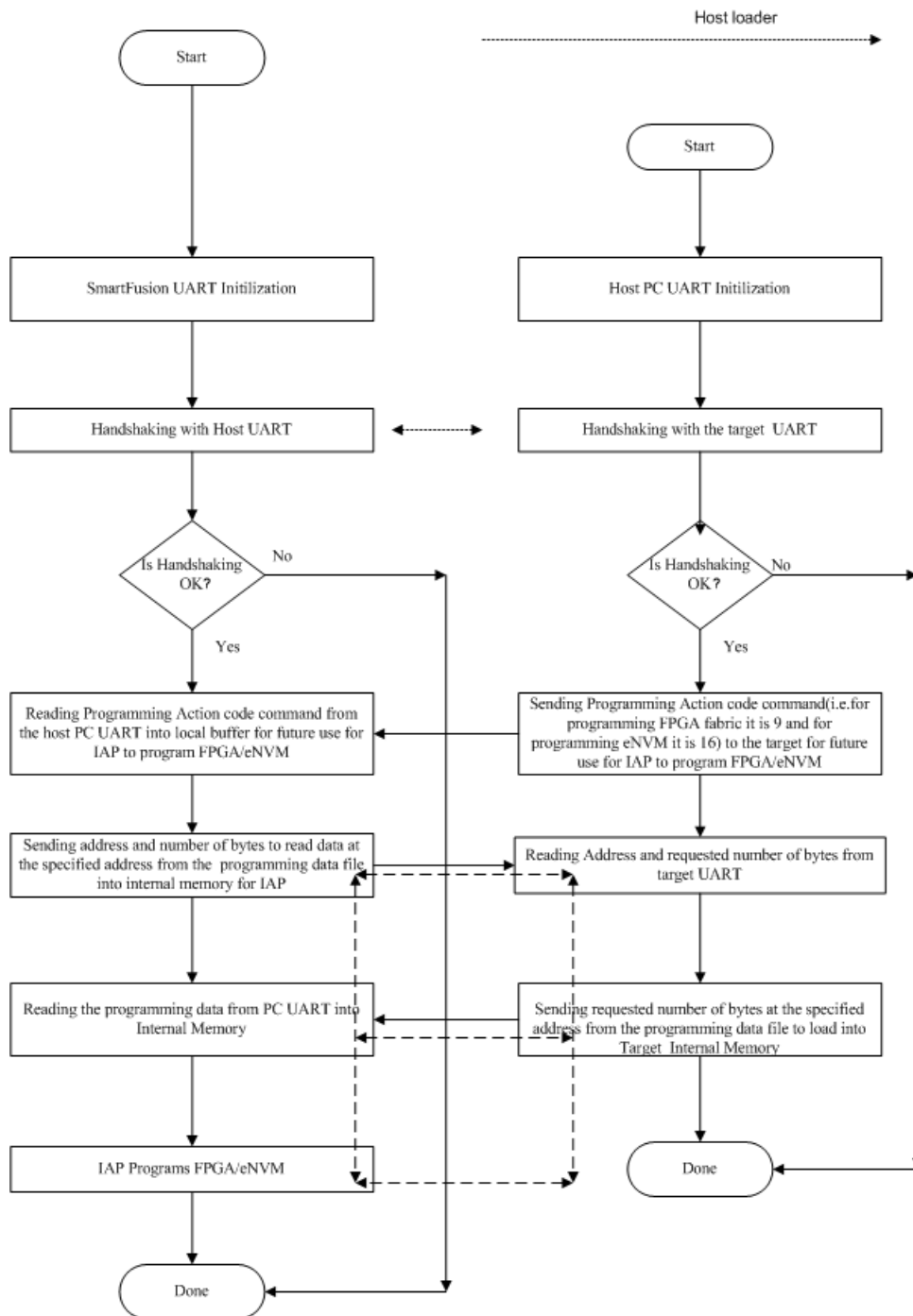


Figure 6 • Flow Chart of IAP Programming Through UART

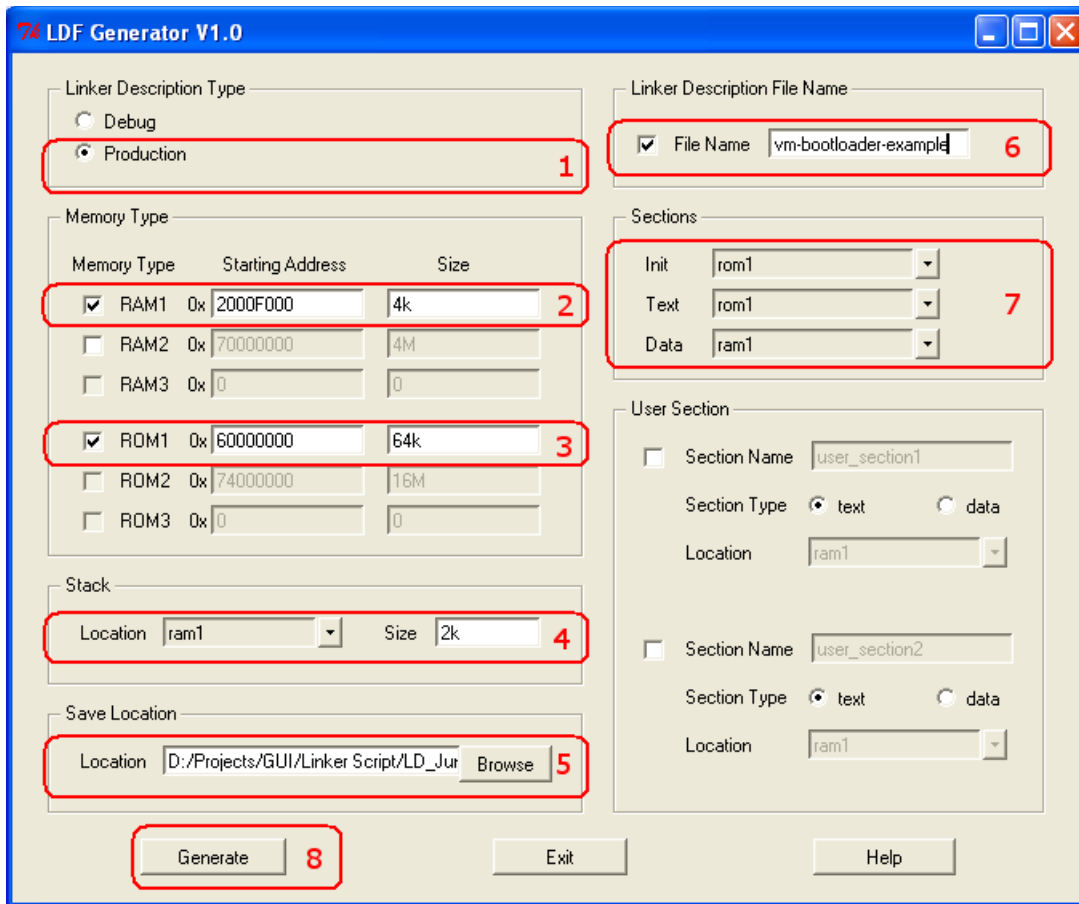
There are various possible methods for getting the *.dat file to the IAP Programmer, such as internet or Ethernet, on board memories (SPI Flash) etc. All these methods of IAP programming are explained in the [SmartFusion cSoC: Programming FPGA Fabric and eNVM Using In-Application Programming Interface](#) application note.

Using the Linker Script Generator Tool

The comprehensive user guide of the linker script generator is covered in "Appendix B" on page 19 of this document. For creating the linker scripts for this design example, use the following settings in the linker script generator as provided below:

1. The executable image of the basic bootloader uses the following memory space:
 - 64 KB of eNVM for interrupt vector table, instructions, and constant data sections
 - 4 KB of eSRAM for stack, heap, and data sections

Figure 7 shows the LDF Generator with options selected for the bootloader example.



The screenshot shows the LDF Generator V1.0 interface with the following settings highlighted by red boxes and numbers:

- 1**: Linker Description Type set to **Production**.
- 2**: Memory Type table with **RAM1** selected, Starting Address **0x2000F000**, and Size **4k**.
- 3**: Memory Type table with **ROM1** selected, Starting Address **0x60000000**, and Size **64k**.
- 4**: Stack Location set to **ram1** and Size set to **2k**.
- 5**: Save Location set to **D:/Projects/GUI/Linker Script/LD_Jur** with a **Browse** button.
- 6**: Linker Description File Name set to **vm-bootloader-example** with a **File Name** checkbox checked.
- 7**: Sections table with **Init**, **Text**, and **Data** all set to **rom1**.
- 8**: **Generate** button.

Figure 7 • LDF Generator with Options Selected for the Bootloader Example

The above selection generates the linker script with the memory map shown in Figure 8 on page 10 for the bootloader image.

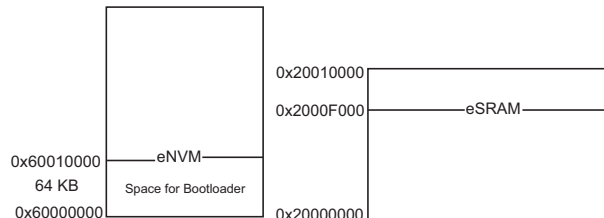


Figure 8 • Memory Regions Used for the Bootloader Example

- The IAP Programmer completely running from eSRAM uses 60 KB of eSRAM for all the sections of an executable image. Figure 9 shows the LDF Generator with settings for a production mode image that completely runs from eSRAM.

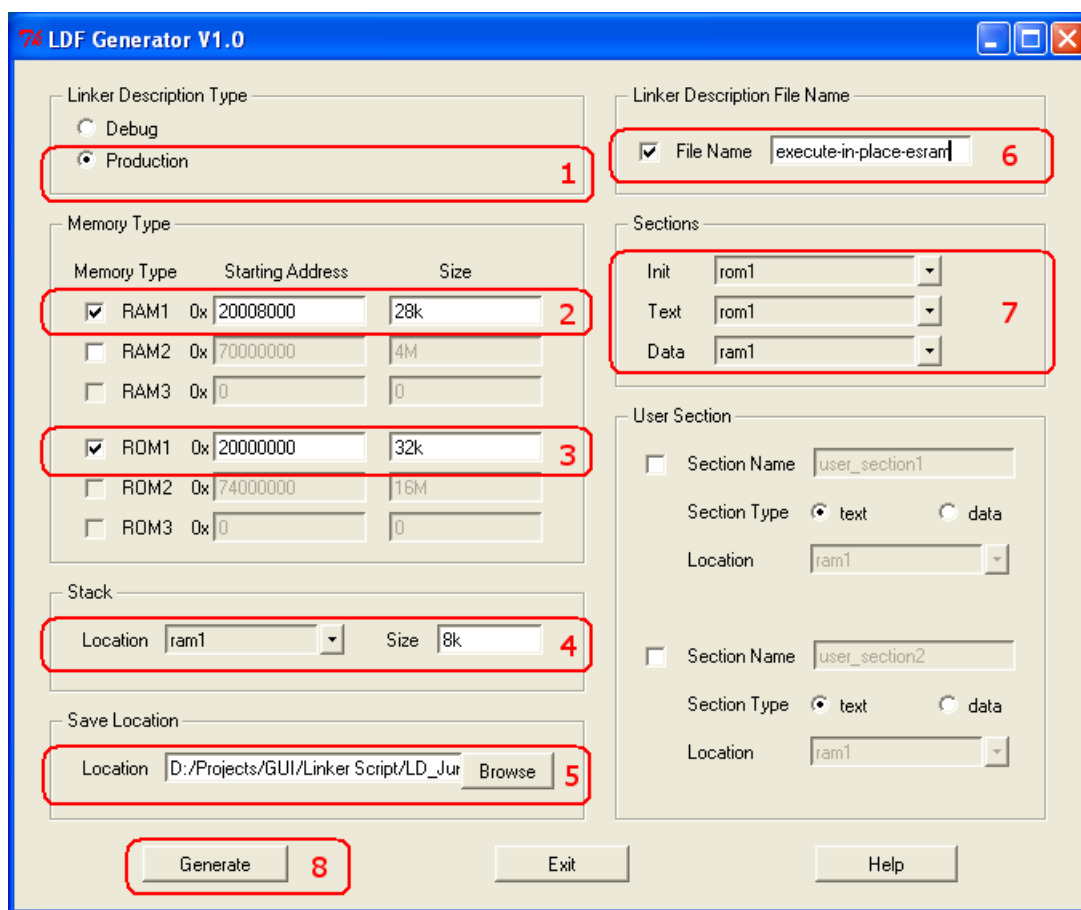


Figure 9 • LDF Generator With Options Selected for Production Mode Image Only Running From eSRAM

3. The settings in the [Figure 9 on page 10](#) generate the linker script with the memory map shown in [Figure 10](#) for a production mode image. In this application note, this linker script is used to build the IAP Programmer algorithm implemented in software with eSRAM memory map.

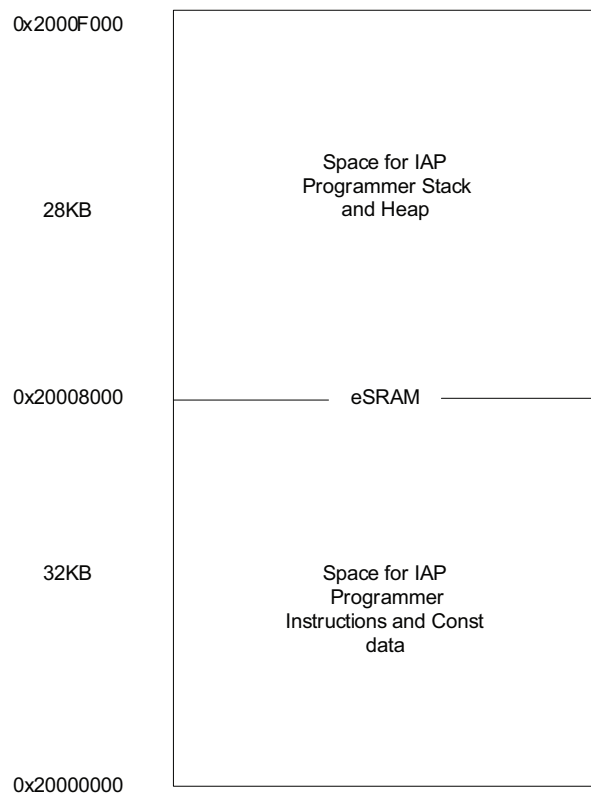
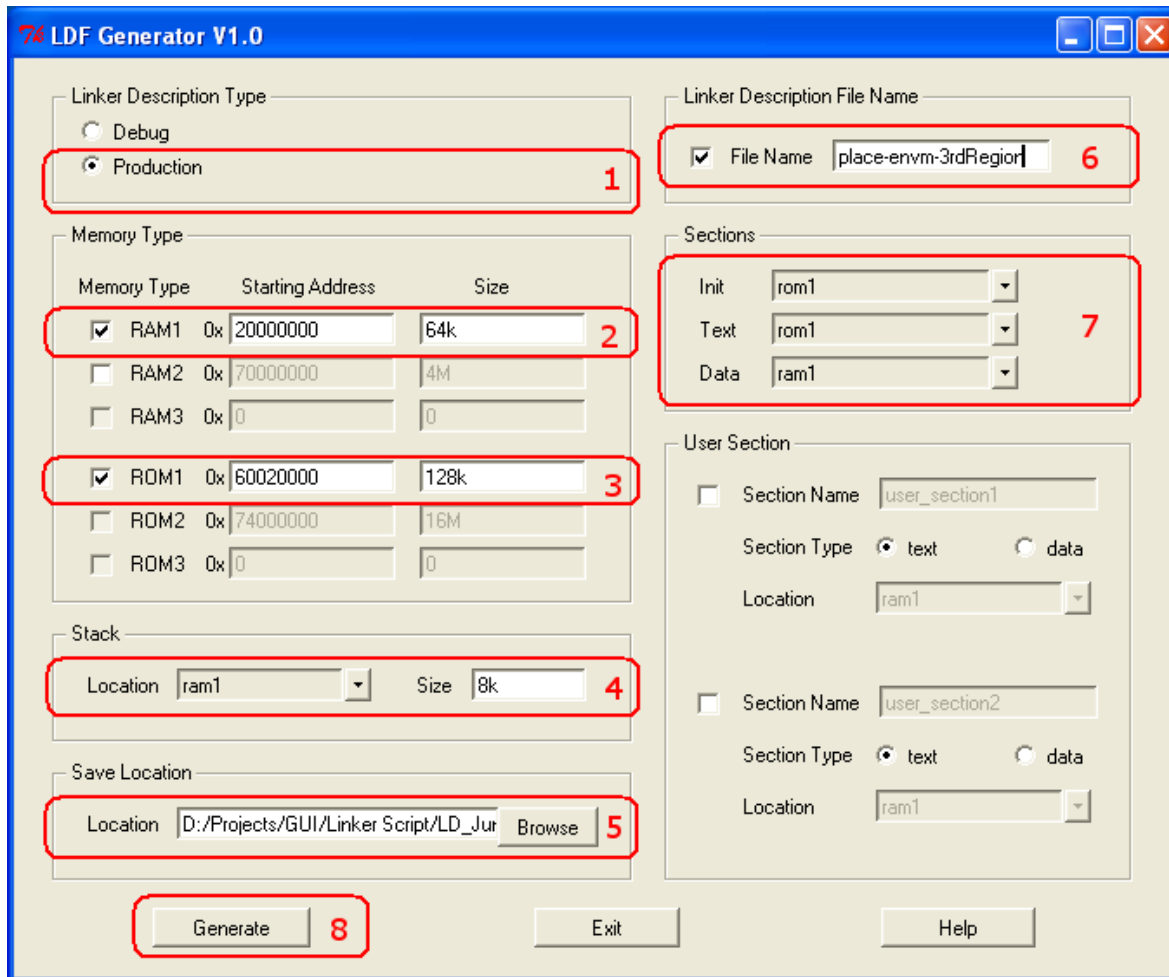


Figure 10 • Memory Regions Used for the IAP Programmer

4. For a simple sample application image the following memory space is used:
 - 128 KB of eNVM for interrupt vector table, instructions, and constant data sections
 - 64 KB of eSRAM for stack, heap, and data sections

Figure 11 shows the LDF Generator with options selected for the sample application.



The screenshot shows the LDF Generator V1.0 window. The following options are selected and numbered:

- 1**: Linker Description Type: Production
- 2**: Memory Type: RAM1 (64k)
- 3**: Memory Type: ROM1 (128k)
- 4**: Stack: Location: ram1, Size: 8k
- 5**: Save Location: Location: D:/Projects/GUI/Linker Script/LD_Jur
- 6**: Linker Description File Name: File Name: place-envm-3rdRegion
- 7**: Sections: Init: rom1, Text: rom1, Data: ram1
- 8**: Generate button

Figure 11 • LDF Generator with Options Selected for Sample Application

The above selection generates the linker script with the memory map shown in Figure 12 on page 13 for the sample application. This sample application is implemented to display the text on both HyperTerminal and OLED.

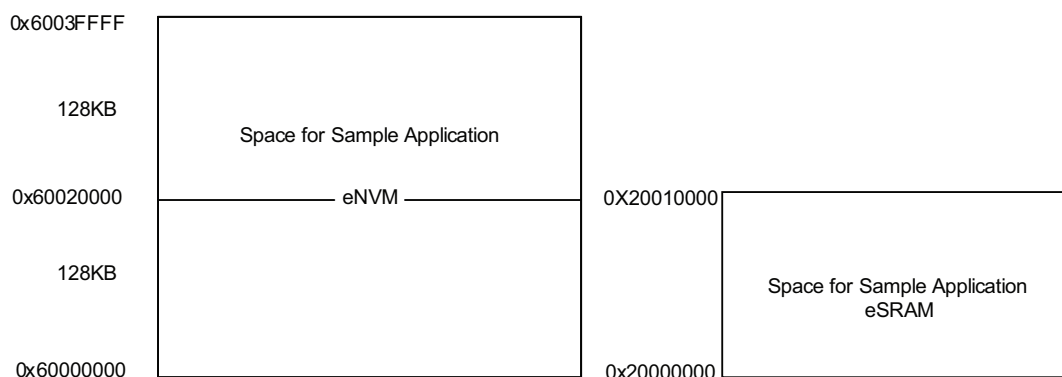


Figure 12 • Memory Regions Used for the Sample Application Example

Board Settings

The design example works on the SmartFusion Development Kit Board with default board settings. Refer to the following user's guide for default board settings:

- [SmartFusion Development Kit User's Guide](#)

Running the Design

Refer to "Appendix A" on page 19 for complete details and links to the design files.

Figure 13 shows the directory structure of the design.

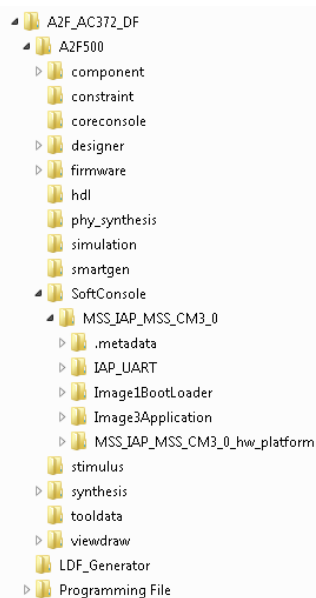


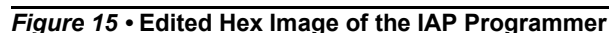
Figure 13 • Directory Structure of the Design Example

- ### Step 1: Procedure to create the eSRAM image in release mode.

1. Launch the SoftConsole project, double-click **Write Application Code** under **Develop Firmware** in the Libero[®] System-on-Chip (SoC) design flow window.
2. Clean and build the application using the Id file “production-execute-in-place-esram.ld”. This is already configured as the default Id file for this project.
3. Once the image is build, open the folder:
VA2F_AC372_DFA2F500\SoftConsole\MSS_IAP_MSS_CM3_0\IAP_UART\UART_HostPC\IAP_UART_From_HostPC_ws\latest_IAP_Prj\Debug and open the *.hex (here IAP_SC.hex). It is displayed as shown in [Figure 14](#). Remove the first line (line 1) of this image by opening it in any text editor in such a way that the second line becomes the first line and then save the file.



- The edited *.hex file is displayed as shown in the [Figure 15](#).



Step 2: Program the provided STP file on the SmartFusion Development Kit Board.

You may need to re-import the *.hex file into eNVM data client in the MSS and save it if there are any changes in the path of the *.hex files or any updates are made to the *.hex files and generate the new STAPL file.

VA2F_AC372_DF\A2F500\designer\impl1\IAP_TOP_fp\IAP_TOP.stp

Step 3: Connect the mini USB cable (USB-RS232) between the USB connector on the SmartFusion Development Kit Board and a USB port of your computer.

Start a HyperTerminal session with a 57600 baud rate, 8 data bits, 1 stop bit, no parity, and no flow control. If your computer does not have the HyperTerminal program, use any free serial terminal emulation program, such as PuTTY or Tera Term. Refer to the [Configuring Serial Terminal Emulation Programs](#) tutorial for configuring HyperTerminal, Tera Term, or PuTTY.

Step 4: Power cycle the board and follow the help provided in the PuTTY.

Figure 16 shows the Menu of the demo design on the PuTTY.

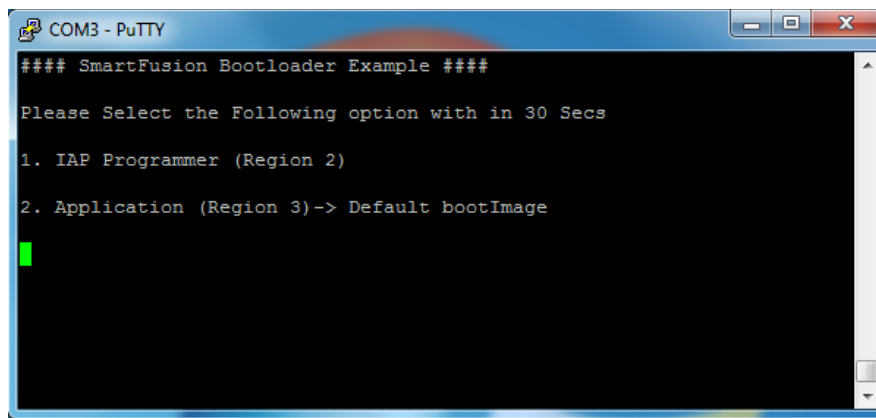


Figure 16 • Menu of the Demo Design on the PuTTY

If input is not provided by the user within 30 seconds then the application from Region 3 will be executed automatically. Figure 17 shows the automatic booting of application image after timeout.

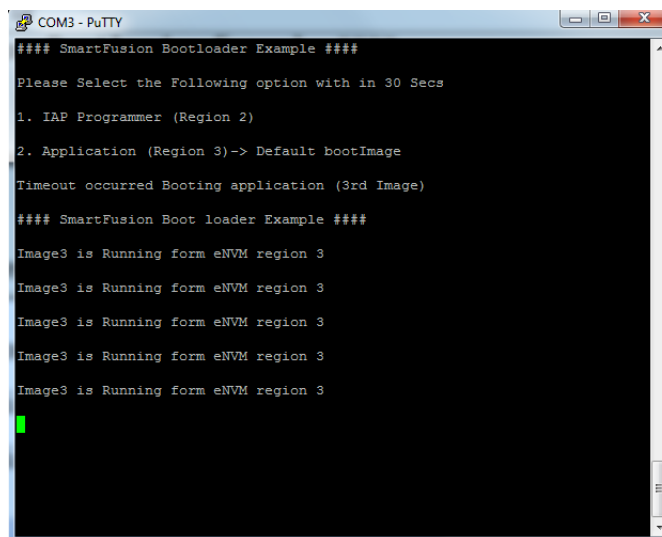


Figure 17 • Booting the Application if User Input is Not Provided

Step 5: If option 1 is selected, the IAP Programmer will be launched by the bootloader.

The IAP Programmer loads or programs the application image into Region 3 or eNVM locations with which the *.dat file is created. So, before selecting this option, create the *.dat file if you wish to change the eNVM programming *.dat file. If you want to create a new *.dat file for FPGA fabric and eNVM, follow the steps given below:

1. Create a Libero SoC project
2. Create *eNVM data client* in MSS
3. Create desired logic in FPGA, connect to the MSS if required
4. Go through the Synthesis, and designer flow
5. Open the FlashPro and export the programming file as a DirectC file (*.dat), using **File > Export > Export Single Programming File**.

The *.dat file used for this application demo is located at:

\\A2F_AC372_DFA2F500\SoftConsole\MSS_IAP_MSS_CM3_0\IAP_UART\UART_HostPC\IAP_UART_From_HostPC_ws\host_tool (file three_eNVM_Images_withOLED.dat at this location is the prebuilt image for this design).

To create or update the application image with the same memory map of this application note, use the *.ld file provided in the application project located at:

\\A2F_AC372_DFA2F500\SoftConsole\MSS_IAP_MSS_CM3_0\Image3Application\simple_app_prj\production-execute-in-place-envm-3rdRegion.ld

Once the dat file is created or updated for eNVM regions, follow the instructions on PuTTY (Figure 18).

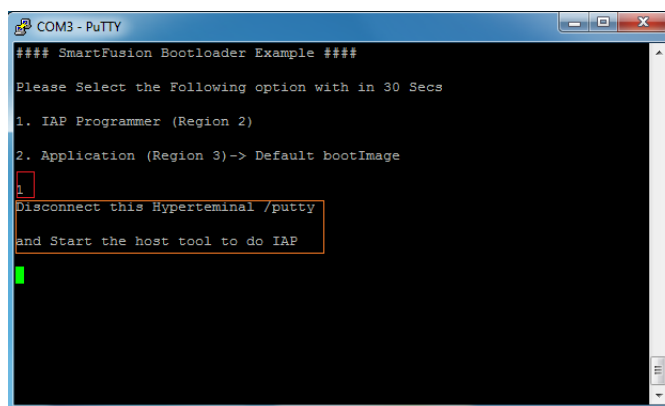
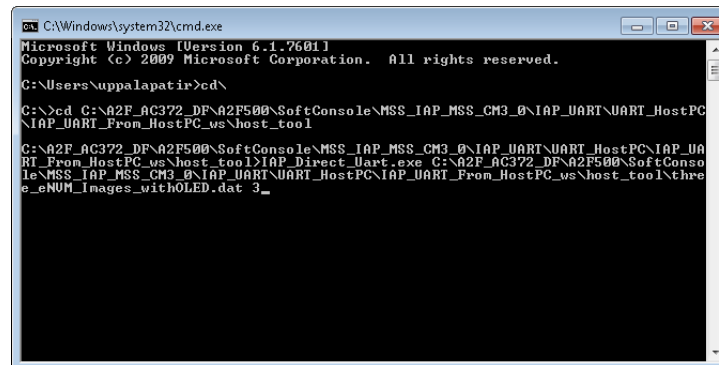


Figure 18 • Menu After the IAP Programmer Selection

Step 6: Open the host loader

As explained in step 5, disconnect the serial terminal and open the host loader located at \\A2F_AC372_DFA2F500\SoftConsole\MSS_IAP_MSS_CM3_0\IAP_UART\UART_HostPC\IAP_UART_From_HostPC_ws\host_tool\IAP_Direct_Uart.exe with the parameters as shown in the Figure 19. The first parameter is the programming input file and the last parameter to the *.exe file is the COM port

number. The COM port number may vary for each system. Check the COM port number to which the USB2UART port is connected. Press **ENTER** to execute this command.



```

C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\suppalapatir>cd\

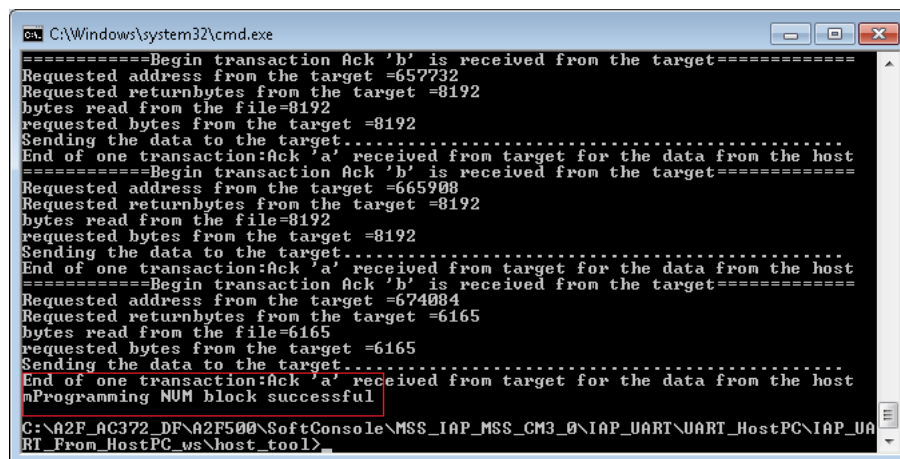
C:\>cd C:\A2F_AC372_DF\A2F500\SoftConsole\MSS_IAP_MSS_CM3_0\IAP_UART\UART_HostPC\IAP_UART_From_HostPC_ws\host_tool

C:\A2F_AC372_DF\A2F500\SoftConsole\MSS_IAP_MSS_CM3_0\IAP_UART\UART_HostPC\IAP_UART_From_HostPC_ws\host_tool>IAP_Direct_Uart.exe C:\A2F_AC372_DF\A2F500\SoftConsole\MSS_IAP_MSS_CM3_0\IAP_UART\UART_HostPC\IAP_UART_From_HostPC_ws\host_tool\Thre_e_eNUM_Images_withOLED.dat 3_
  
```

Figure 19 • Host Tool Usage For Sending the .dat File From Host to SmartFusion

Step 7: There will be continuous debug statements on the command window of the host IAP Programmer.

Here, the output of the provided design automatically boots the device and user can change the design as per the application requirements. At the end of the IAP programming you can view the information appearing on the command prompt as shown in [Figure 20](#).



```

C:\Windows\system32\cmd.exe
=====Begin transaction Ack 'b' is received from the target=====
Requested address from the target =657732
Requested returnbytes from the target =8192
bytes read from the file=8192
requested bytes from the target =8192
Sending the data to the target.....
End of one transaction:Ack 'a' received from target for the data from the host
=====Begin transaction Ack 'b' is received from the target=====
Requested address from the target =665908
Requested returnbytes from the target =8192
bytes read from the file=8192
requested bytes from the target =8192
Sending the data to the target.....
End of one transaction:Ack 'a' received from target for the data from the host
=====Begin transaction Ack 'b' is received from the target=====
Requested address from the target =674084
Requested returnbytes from the target =6165
bytes read from the file=6165
requested bytes from the target =6165
Sending the data to the target.....
End of one transaction:Ack 'a' received from target for the data from the host
mProgramming NUM block successful
C:\A2F_AC372_DF\A2F500\SoftConsole\MSS_IAP_MSS_CM3_0\IAP_UART\UART_HostPC\IAP_UART_From_HostPC_ws\host_tool>
  
```

Figure 20 • Menu of the Host Tool After Completion of the Programming

The above step performs the IAP programming and boots the third eNVM region image. To check this, connect PuTTY once again. The output information is displayed as shown in [Figure 21](#).

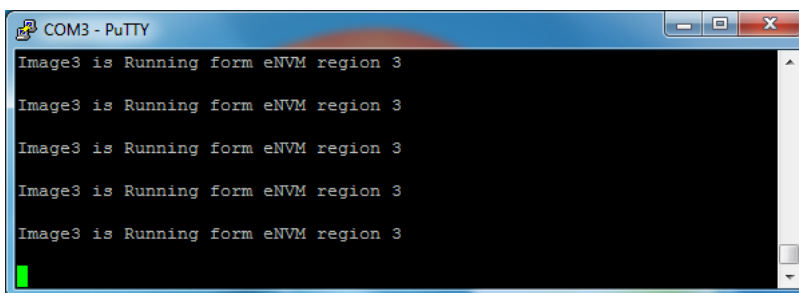


Figure 21 • Message of the eNVM Image 3 Application on PuTTY

Note: When you regenerate the MSS component from Libero SoC project, ensure that the following lines are commented out in `softconsole` project workspace .

MSS_IAP_MSS_CM3_0_hw_platform *library module*:

File name : `DirectC/dpcom.c`

Line number 125: `// #error "Please add code here to get the programming data. Please refer to the Required Source Code Modifications section of the DirectC user's guide."`

File name : `DirectC/dpuser.c`

Line number 15 : `// #include "main.h"`

Line number 48 : `// #error "Please modify this function to time delays. Please refer to the Required Source Code Modifications section of the DirectC user's guide."`

Release Mode

The release mode programming file (STAPL) is also provided. Refer to the `Readme.txt` file included in the programming file for more information. Refer to [SmartFusion cSoC: Building Executable Image in Release Mode and Loading into eNVM](#) tutorial for more information on building an application in release mode.

Conclusion

This application note explains the design and implementation of the following solutions for the SmartFusion cSoC:

1. A basic bootloader to create multiple executable images in a system and boot the input requested image.
2. Creating a release mode image to run completely from eSRAM.
3. eNVM programming in release mode - programming the partial or full eNVM from IAP Programmer running from the eSRAM.

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_AC372_DF.

The design file consists of Libero SoC projects and SoftConsole software projects. Refer to the ReadMe.txt file for directory structure, description, and software versions.

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_AC372_PF.

The programming file consists of STAPL programming file (*.stp) for A2F500-DEV-KIT and a Readme.txt file.

Appendix B

Linker Description File (LDF) Generator Utility

This section explains the use of LDF Generator utility for creating LDFs also called Linker Scripts for SmartFusion cSoC systems. This version of LDF Generator supports only SoftConsole. This document assumes that you already have a basic understanding of the role of Linker and LDF. Refer to [Using ld, the GNU linker](#) document for more details on Linker and LDF.

A basic understanding of the SmartFusion design flow is assumed. Refer to [Using UART with SmartFusion cSoC - Libero SoC and SoftConsole Flow Tutorial](#) to understand the SmartFusion design flow.

LDF Generator Overview

The LDF Generator utility is a powerful graphical user interface (GUI) tool that provides you an easy and convenient way to create LDFs. It reduces the complexity of writing linker scripts that involves complex tasks such as memory map, code, data placement, stack or heap usage etc.

Figure 22 shows a screenshot of the LDF Generator utility.

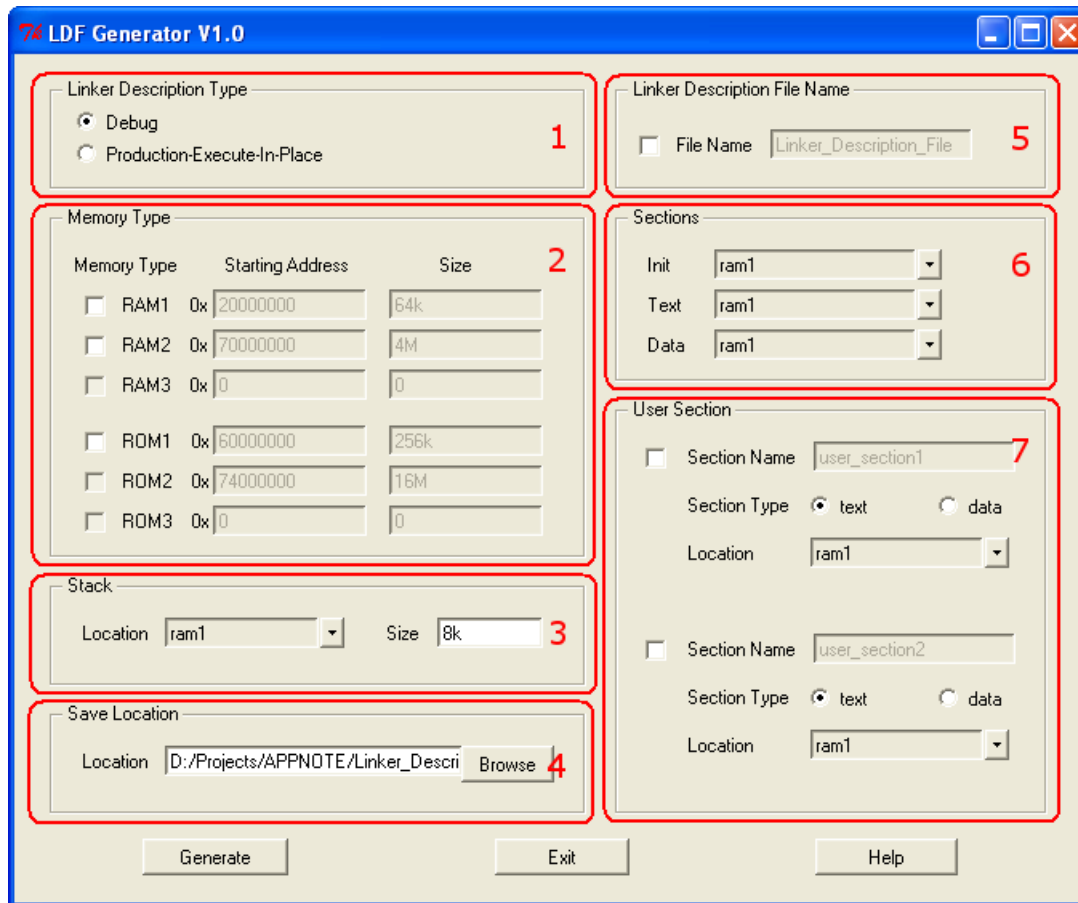


Figure 22 • Linker Description File Generator

The LDF Generator mainly consists of seven segments. You need to enter or select the following parameters:

1. Linker Description Type
2. Memory Type
3. Stack
4. Save Location
5. Linker Description File Name
6. Sections
7. User Sections

Linker Description Type

The LDF type can be selected as either **Debug** or **Production**. There are two radio buttons under the **Linker Description Type** as shown in Figure 23 for selection the LDF type. The default value is **Debug**.

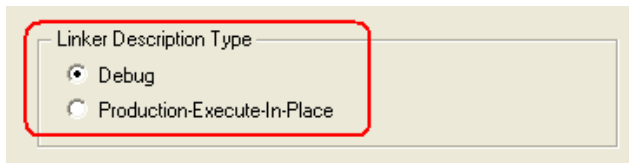


Figure 23 • Linker Description File Type Selection

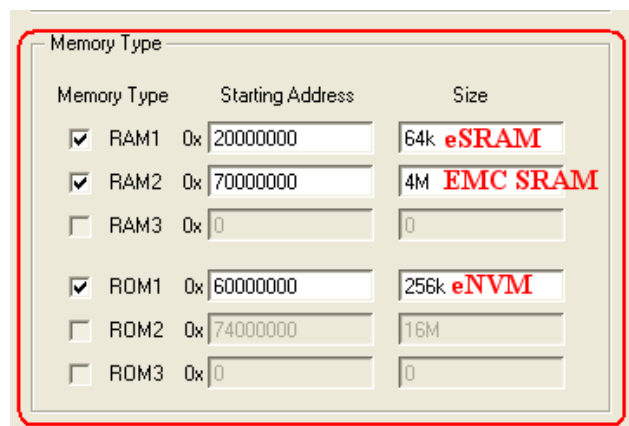
In the **Debug** mode, initialized global and static variables, and uninitialized global and static variables are stored in Read-Write memories (eSRAM and EMC SRAM). The **Debug** mode supports debugging from eSRAM/EMC SRAM and eNVM. If 'ROM1' is selected as memory location for sections or user section, the LDF Generator creates LDF for debugging from eNVM. If 'ROM1' is not selected for sections or user sections, the LDF Generator creates LDF for debugging from eSRAM/EMC SRAM.

In the **Production** mode, initialized global and static variables, and uninitialized global and static variables are stored in Read only Memories (eNVM) and copied back into Read-Write memories at startup.

Memory Type

The **Memory Type** section provides options to select two Read-Write memories and one Read-Only memory. Based on the **Sections** (Init, Text, and Data) storage type (Read-Write/Read-only), the **Memory Type** needs to be selected and the **Starting Address** and **Size** need to be provided.

The **Starting Address** should be in 8-digit Hexadecimal format and the **Size** can be in Kilobytes/Megabytes (for example 64k or 2M). Figure 24 on page 21 illustrates an example Memory Type selection.



Memory Type	Starting Address	Size
<input checked="" type="checkbox"/> RAM1	0x 20000000	64k eSRAM
<input checked="" type="checkbox"/> RAM2	0x 70000000	4M EMC SRAM
<input type="checkbox"/> RAM3	0x 0	0
<input checked="" type="checkbox"/> ROM1	0x 60000000	256k eNVM
<input type="checkbox"/> ROM2	0x 74000000	16M
<input type="checkbox"/> ROM3	0x 0	0

Figure 24 • Memory Type Selection

Stack

The **stack** frame provides you the option to select the stack Location and Size. The default stack location is 'ram1'. The stack location and data section location should be same (that is both stack location and data section in 'ram1' or in 'ram2'). The default stack size is 8k. You can change the stack size based on the design requirements. [Figure 25](#) illustrates an example stack memory location selection.

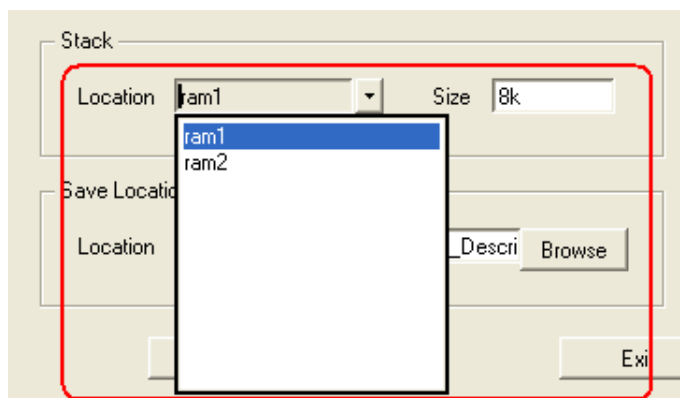


Figure 25 • Stack Location and Size

Save Location

The LDF Generator provides you the option to browse to the directory where the generated LDF needs to be saved. The default location is the current working directory. [Figure 26](#) shows the save location for the generated LDF.



Figure 26 • Save Location

Linker Description File Name

The default name is either Debug or Production based on Linker Description File Type selection. You can type the desired **File Name** in the text field under **Linker Description File Name** as shown in [Figure 27](#).

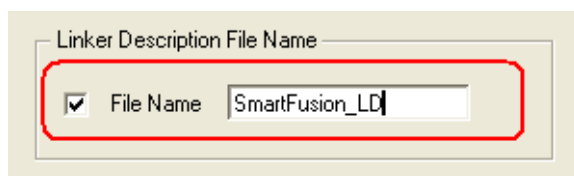


Figure 27 • Changing the Linker Description File Name

Sections

The LDF Generator provides the options to select the following sections and its memory types:

1. Init (Cortex-M3 processor vector table)
2. Text
3. Data

Init, Text, and Data are default sections in the LDF and its memory type needs to be selected from the drop-down menu based on design requirements.

Init

The vector table can be stored in the Read-Write memory or Read-only memory. The LDF Generator provides you the option to select two Read-Write memories and one Read-only memory from the drop-down menu. [Figure 28](#) shows the Init section memory selection.

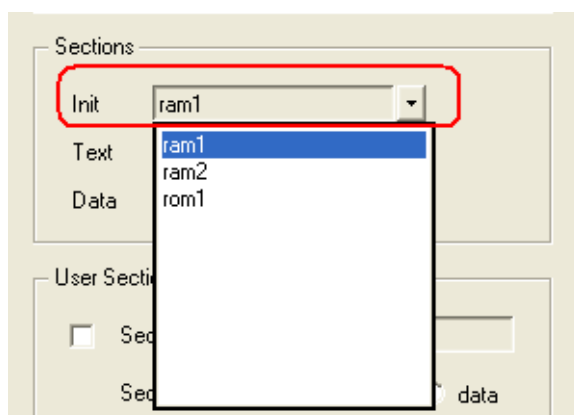


Figure 28 • Init Section Memory Selection

Text

The **Text** section represents the instruction in the program that can be stored in the Read-Write memory or Read-Only memory. The LDF Generator provides you the option to select two Read-Write memories and one Read-only memory from the drop-down menu. [Figure 29](#) shows the Text section memory selection.

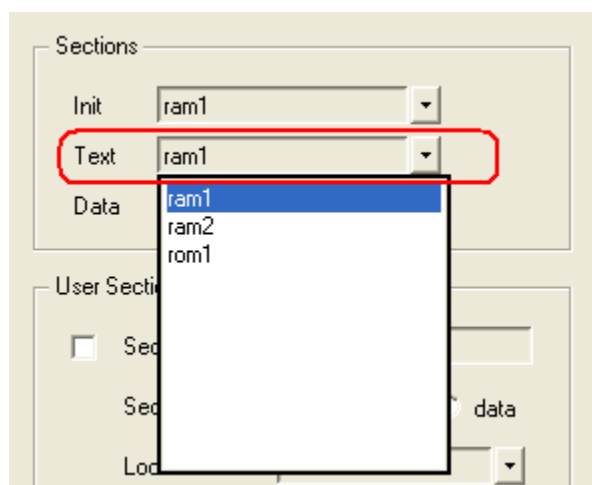


Figure 29 • Text Section Memory Selection

Data

The **Data** section consists of initialized global and static variables that can be stored in the Read-Write memory only. The LDF Generator provides you the option to select two Read-Write memories from the drop-down menu. [Figure 30](#) shows the Data section memory selection.

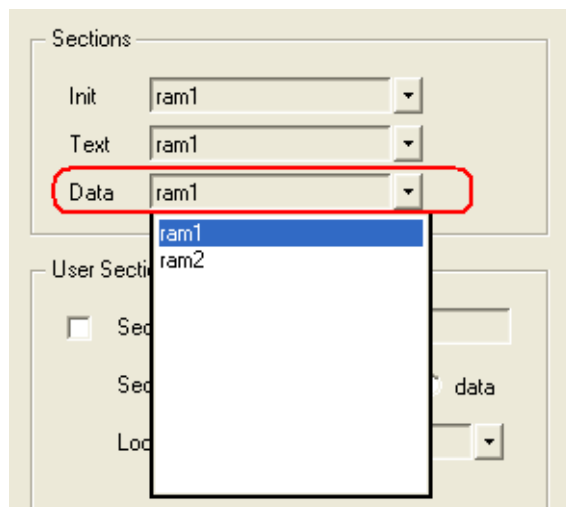


Figure 30 • Data Section Memory Selection

User Section

The LDF Generator provides you the option to select two user defined sections. The section type of each user defined section can be either Text section or Data section. [Figure 31](#) shows the **User Section** selection. You can select either or both check buttons to enable user sections.

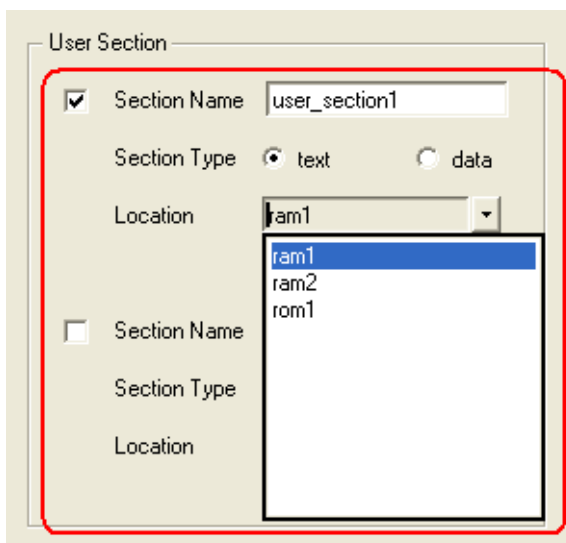


Figure 31 • User Section Selection

Once you select any one of the user sections, you need to append the source file with the following attribute to the above functions and/or data declarations that need to be in the user sections.

```
__attribute__((section ("name_of_user_section")));
```

For example,

```
void hyperterminal_task(void *para) __attribute__((section(".user_section1")));
void hyperterminal_task(void *para)
{
...
...
...
}
```

Once you select any one of the user sections, the LDF Generator pops up a warning message as shown in Figure 32 before generating the linker scripts.

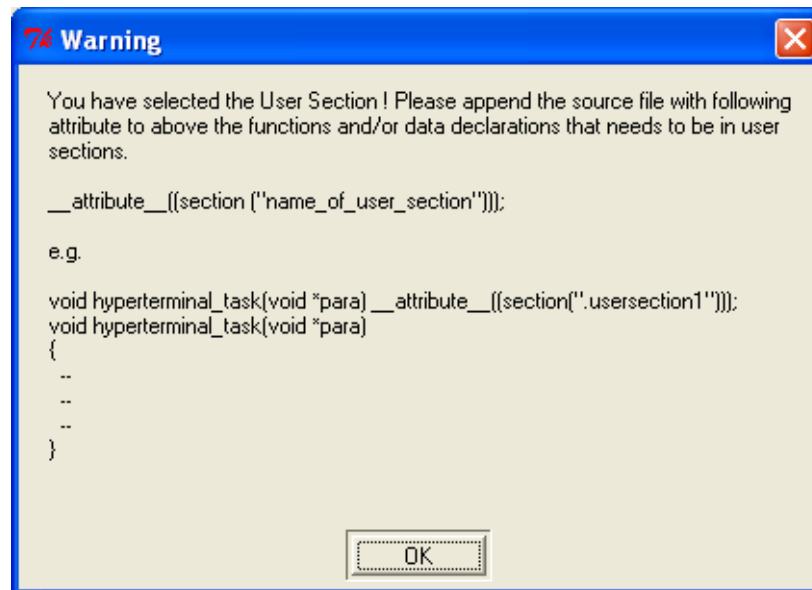


Figure 32 • Attribute Example for User Sections

List of Changes

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

Revision*	Changes	Page
Revision 4 (January 2013)	Added "Board Settings" section under "IAP Programming: eNVM Update in Release Mode Using the IAP Through UART" section (SAR 43469).	13
Revision 3 (April 2012)	Replaced Figure 13 (SAR 38335)	13
	Modified Step 1 under "Running the Design" section (SAR 38335)	13
	Modified Step 5 under "Running the Design" section (SAR 38335)	13
	Replaced Figure 19 (SAR 38335)	17
	Replaced Figure 20 (SAR 38335)	17
Revision 2 (February 2012)	Removed ".zip" extension in the Design files link (SAR 36763).	19
	Modified Step 7 listed under "Running the Design" section (SAR 36683).	13
Revision 1 (January 2012)	Modified text in Step 1, Step 2, Step 5 and Step 6 listed under "Running the Design" section (SAR 35871).	13 to 17
	Added a section called "Release Mode " (SAR 35871).	18
	Modified "Appendix A" section (SAR 35871).	19

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.



Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo CA 92656 USA
Within the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996

Microsemi Corporation (NASDAQ: MSCC) offers a comprehensive portfolio of semiconductor solutions for: aerospace, defense and security; enterprise and communications; and industrial and alternative energy markets. Products include high-performance, high-reliability analog and RF devices, mixed signal and RF integrated circuits, customizable SoCs, FPGAs, and complete subsystems. Microsemi is headquartered in Aliso Viejo, Calif. Learn more at www.microsemi.com.

© 2013 Microsemi Corporation. All rights reserved. Microsemi and the Microsemi logo are trademarks of Microsemi Corporation. All other trademarks and service marks are the property of their respective owners.