

## **Table of Contents**

Purpose
Introduction
Resources
The Cortex-M3 Processor Code Space
Linker Script
Syntax of Commands Used in this Application Note
MEMORY Command
SECTIONS Command
'.': The Location Counter
EXCLUDE_FILE Command
Declaring Function Pointers to Avoid Veneer Generation
Design Description
Hardware Implementation
Software Implementation
Running the Entire Code in Fabric LSRAM [Implementation1]
Subroutine in eSRAM and Main Code in Fabric LSRAM [Implementation2]
Stack in eSRAM and Splitting the Code Between eNVM and the Fabric LSRAM [Implementation3] . 17
Running the Implementations
Speeding Up Code Execution by Copying into Internal SRAM at Boot-time

## **Purpose**

This application note describes how to distribute the application code into different memories such as embedded static random access memory (eSRAM), embedded nonvolatile memory (eNVM), SRAM in fabric etc., and execute the code.

## Introduction

Linker scripts are files (file extension .ld) that contain commands to direct the linker tool, ld, to generate executable files that have data and code sections in the desired memory addresses. Linker scripts can also produce run-time addresses (called virtual memory addresses) that are different from load memory addresses (that is, address where the program image is loaded). This makes it possible to store the program image(s) in one or more non-volatile memories at boot time but run these same images from faster, volatile memories at run-time. The application developer has to write code to relocate (copy) this image to the correct run-time address. This application note covers a number of ways the code and data can be partitioned across various memories and describes the linker script commands involved in the process.

SmartFusion<sup>®</sup>2 System-on-Chip (SoC) field programmable gate array (FPGA) devices integrate an ARM<sup>®</sup> Cortex<sup>™</sup>-M3 processor, up to 512 KB of eNVM, 64 KB of eSRAM, and memory interfaces for DDR/SDR SDRAM for program code with a field programmable fabric for user register transfer level (RTL) implementation.



## **Resources**

This application note is accompanied by three implementation examples targeted to the SmartFusion2. Resources required to run these examples are detailed in Table 1.

The software example implementations accompanying this application note can be used with any Microsemi<sup>®</sup> SoC product that uses ARM embedded processor and the Softconsole tool chain (that is, GNU tools) with minor modifications.

#### Table 1 • Resource Details

Resource Details	Description
Hardware Resources	
SmartFusion2 development kit. Refer the SmartFusion2 Development Kit User Guide for more information	Rev D or later
Host PC or Laptop	<ul> <li>Windows XP SP2 Operating System - 32-bit/64-bit</li> <li>Windows 7 Operating System - 32-bit/64-bit</li> </ul>
Software Resources	
Libero $^{\ensuremath{\mathbb{R}}}$ System-on-Chip (SoC) for viewing the design files	11.3
FlashPro Programming Software	11.3
SoftConsole	3.4

### The Cortex-M3 Processor Code Space

The address range from the 0x00000000 to 0x1FFFFFFF (0.5 GB space) is the code space for the Cortex-M3 processor. Following are the SmartFusion2 SoC FPGA memory sections for the code/data space:

- On-chip eNVM (from 0x60000000 to 0x6007FFFF) of 256 KB for code and constant data regions
- On-chip eSRAM (from 0x20000000 to 0x2000FFFF) of 64 KB with SECDED
- On-chip FPGA fabric RAM (FPGA fabric interface controllers (FIC) region 0). This can be mapped via FIC 0 or FIC 1. This region can be accessed by a system bus for instructions and data
- External RAM interfaced through DDR or SDR interfaces (from 0xA0000000 to 0xDFFFFFF) of 1 GB for both code and data regions

This application note focuses on the following regions:

- eNVM from 0x60000000 to 0x6007FFFF
- Internal eSRAM at 0x2000000 (used for stack and heap)
- Internal AHB connected LSRAM using the free address space at 0x30000000

The aim is to partition the executable code into eNVM, eSRAM, and internal LSRAM

The current application note limits itself to demonstrate how this can be done during debug and development. General guidelines are provided at the end for how to deploy such a solution (that is, a release mode build).

## **Linker Script**

Linker scripts are text files. A linker script is written as a series of commands. Each command is either a keyword, possibly followed by arguments or an assignment to a symbol.

The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file (image file).

An executable and linkable format (ELF) file is an example of an object file. Object files participate in program linking (building a program) and program execution (running a program). Object files are created by the assembler and link editor. The object files are binary representations of programs that are intended to be executed directly on a processor.

The GNU linker tool, ld, combines a number of object and archive files, relocates their data, and ties up symbol references. Usually the last step in compiling a program is to run ld. The purpose of this section is to familiarize the user with the keywords necessary for implementation. For a comprehensive list, refer to the ld manual: *https://sourceware.org/binutils/docs/ld/* 

The most fundamental command for ld is the SECTIONS command which specifies the output sections. Every meaningful linker script must have a SECTIONS command. It specifies a picture of the output file's layout, in varying degrees of detail.

The MEMORY command complements SECTIONS command by describing the available memory in the target architecture. This command is optional.

Comments may be included in linker scripts just as in C: delimited by '/\*' and '\*/'. As in C, comments are syntactically equivalent to whitespace.

#### Syntax of Commands Used in this Application Note

This section covers the following commands required to understand the examples provided along with this application note.

- MEMORY Command
- SECTIONS Command
- '.' : The Location Counter
- EXCLUDE FILE Command

#### MEMORY Command

The MEMORY command describes the location and size of blocks of memory in the target system. This command specifies details of the memory regions that may be used by the linker, and the ones it must avoid. Though the linker does not shuffle sections to fit into the available regions, it does move the requested sections into the correct regions and issues errors when the regions become too full.

The following section from GNU linker document explains the command syntax.

```
MEMORY
{
  name (attr) : ORIGIN = origin, LENGTH = len
  ...
}
```

where,

name is the name used internally by the linker to refer to the region. Any symbol name may be used. The region names are stored in a separate name space, and do not conflict with symbols, file names, or section names. Distinct names should be used to specify multiple regions.

(attr) is an optional list of attributes. Valid attribute lists must be made up of the characters "LIRWX". If the attribute list is omitted, the parentheses around it must be omitted as well.

origin is the start address of the region in physical memory. It is an expression that must evaluate to a constant before memory allocation is performed. The keyword ORIGIN may be abbreviated to org or o (but not, for example, 'ORG').

len is the size in bytes of the region (an expression). The keyword LENGTH may be abbreviated to 'len' or 'l'.



For example, consider the following line taken from the linker script contained in the design files used to run the entire code from Fabric SRAM [Implementation1].

#### Example 1

```
MEMORY
{
  ram (rwx) : ORIGIN = 0x30000000, LENGTH = 16k /* fabric SRAM address and length*/
  esram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}
```

In example 1, two memory regions are defined: one for storing code (ram), and another for stack (esram).Once a memory region is defined, '>region' directs the linker to place specific output sections into that memory region.

For example, if there is a memory region named ram, use '>ram' in the output section definition.

If no address is specified for the output section, the linker sets the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker issues an error message.

See the example below:

```
.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
```

In the above example, the .data section is loaded into a memory region called ram. This region would be defined earlier in the linker script using the MEMORY command.

#### SECTIONS Command

The SECTIONS command controls how the input sections are combined into output sections, as well as their order in the output file. A maximum of one SECTIONS command may be used in a script file, but it can have as many statements within it. Statements within the SECTIONS command can do one of three things:

- Define the entry point
- · Assign a value to a symbol
- · Describe the placement of a named output section, and which input sections go into it

The SECTIONS command is written as the keyword SECTIONS, followed by a series of symbol assignments and output section descriptions enclosed in curly braces.

The most frequently used statement in the SECTIONS command is the section definition, which specifies the properties of an output section: its location, alignment, contents, fill pattern, and target memory region. Most of these specifications are optional; the simplest form of a section definition is:

```
SECTIONS { ...
secname : {
contents
}
... }
```

where,

 ${\tt secname}$  is the name of the output section

contents specifies what goes in the output section, for example, a list of input files or sections of input files.



For example, let's suppose, code (that is, the text section) needs to be loaded at 0x20000000, and the data section needs to be loaded at the 0x30000000 location, which are the RAM and eSRAM memory regions as declared in the memory command in memory section. Below is a linker script that performs the task:

#### Example 2

SECTIONS

```
{
.text :
{
CREATE OBJECT SYMBOLS
text load = LOADADDR(.text);
__text_start = .;
 vector table vma base address = .;
*(.isr_vector)
*(.text)
} >esram
.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 }
```

Here, by using the '>' token at the end of .data, the linker is directed to place the .data in the specified memory region ram. Similarly, ".text" is placed in eSRAM.

The first line defines an output section, '.text'. The colon following the .text is required syntax that may be ignored for now. Within the curly braces after the output section name, list the names of the input sections that must be placed into this output section. The '\*' is a wildcard that matches any file name. The expression '\*(.text)' means all '.text' input sections in all input object (that is, .o) files. This text section is loaded into the eSRAM location (>eSRAM) that starts at 0x20000000. The data section is loaded at the ram location (>ram) that starts at 0x30000000.



#### '"' : The Location Counter

The special linker variable dot '.' always contains the current output location counter. Since the '.' always refers to a location in an output section, it must always appear in an expression within a SECTIONS command. The '.' symbol may appear anywhere that an ordinary symbol is allowed in an expression, but its assignments have a side effect. Assigning a value to the '.' symbol causes the location counter to be moved. This may be used to create holes in the output section and place the data/code at a specific location. The location counter may never be moved backwards.

#### Example 3

```
SECTIONS
{
  .text :
  {
  file1(.text)
  . = . + 1000;
  file2(.text)
  . += 1000;
  file3(.text)
  } = 0x1234;
  }
```

In the example 3, file1 is located at the beginning of the text section, followed by a 1000 byte gap. Then file2 appears, also with a 1000 byte gap following before file3 is loaded. The notation '= 0x1234' specifies the data that is to be written (filled) in the gaps.

#### EXCLUDE\_FILE Command

Let's consider the .text : { \*(.text) } component from the previous example (example 2). Here, '\*' is a wildcard that matches any filename, hence, in the above example, it includes all input '.text' sections from all input object files.

In this application note, for implementation2 and implementation3, we need subroutine to be excluded from the list, so that it can be loaded into a different memory location. So, the EXCLUDE\_FILE is used to exclude the particular subroutine.o file from loading into the text section.

```
*(EXCLUDE FILE (*subroutine.o) .text.*)
```

In the above command all files are loaded except files that match \*subroutine.o.

To load subroutine.o file into the LSRAM in the fabric, use:

```
.mytext :
{
*subroutine.o(.text.*)
} >lsram
```

Here, LSRAM is a section in memory that needs to be defined using the MEMORY command, covered earlier in the document.



## **Declaring Function Pointers to Avoid Veneer Generation**

Veneers are small sections of code generated by the linker and inserted into your program. 'armlink' generates veneers when a branch involves a destination beyond the branching range of the current range.

In implementation2 and implementation3), this is certain when the code is partitioned across regions starting at 0x20000000 and 0x30000000.

The range of a branch long (BL) instruction is 32 MB for ARM and 4 MB for Thumb. A veneer can, therefore, extend the range of the branch by becoming the intermediate target of the instruction and then setting the PC to the destination address.

The disadvantage with veneers is that single stepping through a function located beyond the range of the BL instruction becomes impossible. This happens because the entire function is executed in the veneer function called. To enable debugging code partitioned across many regions spaced far apart, function pointers have to be used to call the function. Using this approach, the function then can be single stepped while debugging.

## **Design Description**

The design example in this application note uses MSS, FIC, AHBLSRAM, eSRAM, and eNVM memory. The design consists of MSS with FIC\_0 enabled for AHB master interface. AHBLSRAM has been instantiated with the size of 16 K locations of 32 bits each. Fabric oscillator is used as a clock source, which is then given to the FCCC. The output of FCCC is the clock for the MSS. CoreAHBLite is instantiated to connect the FIC\_O and AHBLSRAM.

#### **Hardware Implementation**

The hardware implementation involves configuring MSS, Fabric, CCC, oscillator, sysreset, and AHBLSRAM. Figure 1 shows the top level SmartDesign of the application.

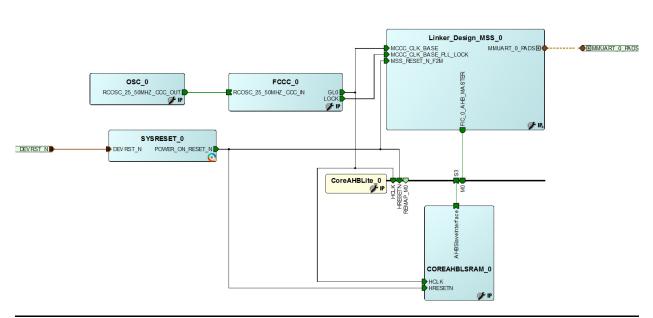
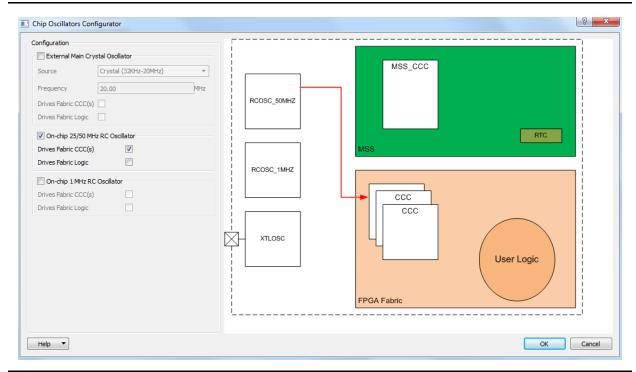


Figure 1 • Top-Level SmartDesign



On-chip oscillator of 25/50 MHz has been configured as the source for Fabric CCC. Figure 2 shows the **Oscillator Configuration** window.



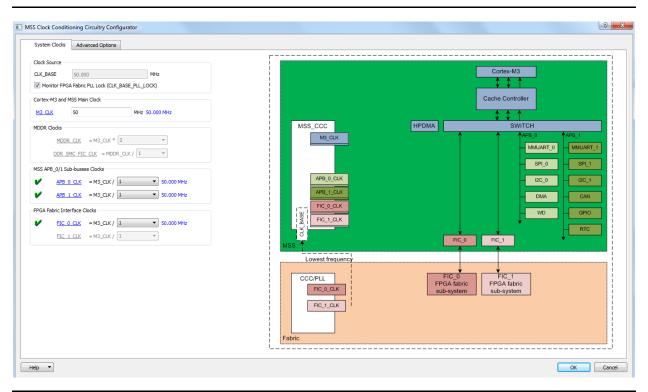
#### Figure 2 • Oscillator Configuration

Fabric CCC has been configured to take 50 MHz on-chip oscillator and give an output of 50 MHz at GL0. This GL0 output is used by the MSS\_CCC and provides a clock of 50 MHz as shown in Figure 3 and Figure 4 on page 9.

Reference Clock	ссс	C GLO	Frequency	Actual	
		🗖 GL1	50 MHz 100 MHz	50.000 MHz	
		GL2	100 MHz	MHz	
			200		

Figure 3 • Fabric PLL Configuration





#### Figure 4 • MSS CCC Configuration

MSS reset is configured to 'Enable FPGA Fabric to MSS Reset (MSS\_RESET\_N\_F2M)' as shown in Figure 5.

S Configuring RESET (MSS_RESET - 1.0.100)
Configuration
Enable FPGA Fabric to MSS Reset (MSS_RESET_N_F2M)
Enable FPGA Fabric to M3 Reset (M3_RESET_N)
Enable MSS to FPGA Fabric Reset (MSS_RESET_N_M2F)
Help  Cancel

Figure 5 • MSS RESET Configuration



FIC\_0 is configured to use the ABHLite master interface as shown in Figure 6.

SS To FPGA Fabric Interface	
nterface Type AHBLite	Cortex-M3
se Master Interface	
ise Slave Interface	Cache Controller
dvanced AHBLite Options	Cache Componer
Ise Bypass Mode (AHBLite only)	
xpose Master Identity Port	SWITCH
, , _	
PGA Fabric Address Regions (MSS Master View)	
FIC32_0 FIC32_1	FIC_n (n=0.1)
abric Region 0 (0x30000000 - 0x3FFFFFFF)	
abric Region 1 (0x50000000 - 0x5FFFFFFF)	
abric Region 2 (0x70000000 - 0x7FFFFFFF) 💿 💿	MSS
abric Region 3 (0x80000000 - 0x8FFFFFFF)	
abric Region 4 (0x90000000 - 0x9FFFFFF) (0) (0) abric Region 5 (0xF0000000 - 0xFFFFFFFF) (0) (0)	
abric Region 5 (0xF0000000 - 0xFFFFFFFF) 🔘 💿	Fabric

#### Figure 6 • FIC Configuration

Core AHBLSRAM is configured for the space of 16 K locations of 32 bits each. The number of locations specified should be a multiple of 2048.

Note: The LSRAM depth refers to number of locations, that is, 16 K locations of 32 bits each.

	M_0 (COREAHBLSRAM - 2.0.113)
Configuration	
AHB Data Width:	32
AHB Address Width:	32
Select SRAM Type	
ISRAM	M 🔘 uSRAM
LSRAM Depth	
Number of bytes	s of memory: 16384
uSRAM Depth	
Number of bytes	s of memory: 512
Testbench:	User 🔹
License:	
RTL	Obfuscated
Help 🔻	OK Cancel

Figure 7 • Fabric LSRAM Configuration



Core AHBLite is configured with a **Memory space** of 4 GB addressable space with 16 slots of 256 MB each. The Selecting **M0 can access slot 3** under **Enable Master access** gives the address of slave as 0x30000000.

ration							
Memory space							
	Mem	nory space: 4GB addressable space app	ortioned into	16 slave slots, each of size 256MB		•	
	Ad	ddress range seen by slave connected to	huge (2GB)	slot interface:			
		Ox0000000 - 0x7FFFFFF	-	0x80000000 - 0xFFFFFFFF			
Allocate memory space to combined r	egion slave						
Slot 0:		Slot 1:		Slot 2:		Slot 3:	
Slot 4:		Slot 5:		Slot 6:		Slot 7:	
Slot 8:		Slot 9:		Slot 10:		Slot 11:	
Slot 12:		Slot 13:		Slot 14:		Slot 15:	
Enable Master access							
M0 can access slot 0:		M1 can access slot 0:		M2 can access slot 0:		M3 can access slot 0:	
M0 can access slot 1:		M1 can access slot 1:		M2 can access slot 1:		M3 can access slot 1:	
M0 can access slot 2:		M1 can access slot 2:		M2 can access slot 2:		M3 can access slot 2:	
M0 can access slot 3:		M1 can access slot 3:		M2 can access slot 3:		M3 can access slot 3:	
M0 can access slot 4:		M1 can access slot 4:		M2 can access slot 4:		M3 can access slot 4:	
M0 can access slot 5:		M1 can access slot 5:		M2 can access slot 5:		M3 can access slot 5:	
M0 can access slot 6:		M1 can access slot 6:		M2 can access slot 6:		M3 can access slot 6:	
M0 can access slot 7:		M1 can access slot 7:		M2 can access slot 7:		M3 can access slot 7:	
M0 can access slot 8:		M1 can access slot 8:		M2 can access slot 8:		M3 can access slot 8:	
M0 can access slot 9:		M1 can access slot 9:		M2 can access slot 9:		M3 can access slot 9:	
M0 can access slot 10:		M1 can access slot 10:		M2 can access slot 10:		M3 can access slot 10:	
M0 can access slot 11:		M1 can access slot 11:		M2 can access slot 11:	<b>[</b> ]]	M3 can access slot 11:	

Figure 8 • AHBLite Configuration



#### **Software Implementation**

This AN covers three ways to split the application code across the memory regions. They are:

- Running the Entire Code in Fabric LSRAM [Implementation1]
- Subroutine in eSRAM and Main Code in Fabric LSRAM [Implementation2]
- Stack in eSRAM and Splitting the Code Between eNVM and the Fabric LSRAM [Implementation3]

#### Running the Entire Code in Fabric LSRAM [Implementation1]

To run the entire code in Fabric LSRAM, the memory section of the linker script of eSRAM needs to be modified. Modify the RAM origin address to 0x30000000, which is the address of the LSRAM in FABRIC, and set the length as 16 K as LSRAM is configured for 16 K locations of 32 bits each. The following details show how to do this.

MEMORY

}

```
/* SmartFusion2 internal LSRAM */
ram (rwx) : ORIGIN = 0x30000000, LENGTH = 16k
```

The following sections also need to be modified to set the stack size and address in the memory space available.

```
RAM_START_ADDRESS = 0x30000000; /* Must be the same value MEMORY region ram ORIGIN
as above. */
RAM_SIZE = 16k; /* Must be the same value MEMORY region ram LENGTH as above. */
MAIN_STACK_SIZE = 8k; /* Cortex main stack size. */
PROCESS_STACK_SIZE= 4k; /* Cortex process stack size (only available with OS
extensions).*7
```

To verify this, use the following simple application code that writes into the memory at 0x20000000 (eSRAM memory region), reads back the data from there, adds a value to it, and writes it into another variable. Running this code displays the values getting updated.

```
p = 0x20000000;
*p=100;
*p+=20;
q=*p;
```

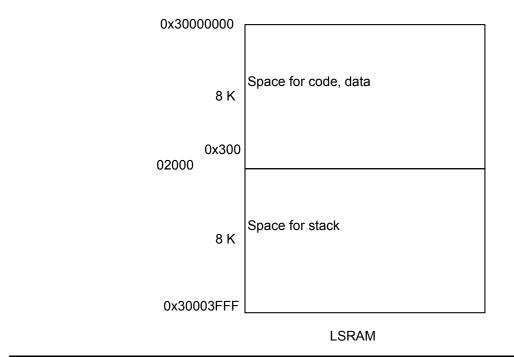
The disassembly window displays the address of the instructions that start from 0x3000000, which is the address of the LSRAM.

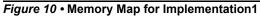


Upon completion of this step-by-step execution, the value of variables getting updated can be seen. Finally, the value of q will be updated to 120.

Debug - Linker Design MSS_CM3_app/main.c - Microsemi SoftConsole IDE v3.4			- 0
File Edit Source Refactor Navigate Search Project Run Window Help			
·····································	•		腔 (珍 Debug) 昭 C/C+・
<sup>™</sup>			
SC Linker_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target]	Name	s at modules	Value
✓ Embedded GDB (4/15/14 10:21 AM) (Suspended)	p p		0x2000000
<ul> <li>Thread [1] (Suspended)</li> </ul>	(e)- g		120
I main() e:\ranjith\linker_design\softconsole\linker_design_mss_cm3\linker_design C:\Microsemi\Libero_v11.2\SoftConsole\Sourcery-G++\bin\arm-none-eabi-gdb.exe (4/1)			
C. (wicrosennicheelo_viii:2/sortconsole(sourcely-ov+ (on (ann-none-eaol-gub.exe (v)))			
	4		×
main.c 🛙 📄 debug-in-microsemi-smartfusion2-esr	ion2-en C -1 <symbol available="" is="" not=""></symbol>	🗖 🖥 📴 Disassembly 🛛	□
a mainte 🔅 🔚 debug-in-microsemi-smartrusionz-esi 📄 debug-in-microsemi-smartru	ionz-en C1 < symbol is not available>	A I I	
2 int main()		0x30000320 <main>: push (r7)</main>	í literatura de la companya de la co
3 (		0x30000322 <main+2>: sub sp, #12</main+2>	
4 5 int *p,q;		<pre>0x30000324 <main+4>: add r7, sp, p = 0x20000000;</main+4></pre>	#0
6		0x30000326 <main+6>: mov.w r3, #53</main+6>	
7 p = 0x20000000;		0x3000032a <main+10>: str r3, [r7</main+10>	, #0]
8 *p=100; 9 *p+=20;		*p=100; 0x3000032c <main+12>: ldr r3, [r7</main+12>	. #01
10 g=*p;		0x3000032e <main+14>: mov.w r2, #10</main+14>	0 ; 0x64
11 12 while(1)		0x30000332 <main+18>: str r2, [r3</main+18>	i, #0]
12 while(1) 13 (		*p+=20; 0x30000334 <main+20>: ldr r3, [r7</main+20>	. #01
14 }		0x30000336 <main+22>: ldr r3, [r3</main+22>	
15}		0x30000338 <main+24>: add.w r2, r3,</main+24>	
16		0x3000033c <main+28>: ldr r3, [r7 0x3000033e <main+30>: str r2, [r3</main+30></main+28>	
		q=*p;	
		0x30000340 <main+32>: ldr r3, [r7</main+32>	
4		<pre>* 0x30000342 <main+34>: ldr r3, [r3 *</main+34></pre>	i, #U]
Console 🕱 🖉 Tasks 🖹 Problems 🕡 Executables 🚦 Memory			× % 🔩 🖬 🔜 🖳 💭 🗗 🖢 • 📬 •
version and the second seco	onsole\Sourcerv-G++\bin\arm-none-eabi-adb.ex	(4/15/14 10:21 AM)	
m-none-eabi-sprite: Got packet: 'p1f'		· And and a second second	
rm-none-eabi-sprite: Sent response: '00000000'			
cm-none-eabi-sprite: Got packet: 'p20'	m		
T*		Writable	Smart Insert 14:1
		viitable	

Figure 9 • Softconsole Debug Showing Memory Address for Implementation1







#### Subroutine in eSRAM and Main Code in Fabric LSRAM [Implementation2]

In this implementation, the main code is loaded into the Fabric LSRAM, and the subroutine file is loaded into the eSRAM.

Two memory regions need to be declared: one for Fabric LSRAM, and one for eSRAM with the appropriate address and lengths as shown below.

```
MEMORY
{
    ram (rwx) : ORIGIN = 0x3000000, LENGTH = 16k
    esram (rwx) : ORIGIN = 0x2000000, LENGTH = 64k
}
We need to make the following changes, so the stack is loaded into the eSRAM:
RAM_START_ADDRESS = 0x20000000; /* Must be the same value MEMORY region ram ORIGIN
as above. */
RAM_SIZE = 64k; /* Must be the same value MEMORY region ram LENGTH as above. */
MAIN_STACK_SIZE = 8k; /* Cortex main stack size. */
PROCESS_STACK_SIZE= 4k; /* Cortex process stack size (only available with OS
extensions).*7
```

In the text section of the linker script, everything is loaded into the LSRAM except the subroutine.o file as shown below:

```
*(.text)
*(EXCLUDE FILE (*subroutine.o) .text.*)
```

Another section called  $\tt.mytext$  is declared and the subroutine.o file is loaded into the eSRAM region as shown below:

```
.mytext :
  {
    {
        subroutine.o(.text.*)
    } >esram
```

The declared functions add, sub, and mul perform the addition, subtraction, and multiplication respectively of two numbers.

These functions are in the subroutine.c file. To avoid the generation of veneers, declare the function pointers to these functions as shown below.

```
int (*add_ptr)(int , int ) ;//function pointer to add
int (*sub_ptr)(int , int );//function pointer to sub
int (*mul_ptr)(int , int ); //function pointer to mul
```



Run the design to see the values returned by these function pointers in the eSRAM region as the subroutine is loaded into the eSRAM, as shown in Figure 11.

e Edit Source Refactor Navigate Search Project Run Window Help								
1 • 🛛 🛆 📄 🐘 · Ø • Ø • 🧶 • 🧶 🖋 • 📝 ½ · ½ • ∞ • ↔ ↔ •								
								🖹 🏇 Debug 🗟 C/C
: Debug 🛛 🥂 🖓 🗱 🛷 🕪 💷 🖬 🕺 🗫 🕫 🕫 😾 🤯 🏹 🛱 🏹 🗖	🕫 Variables 🛛 🗣 Breakpoints	Registers 🛋 Modules					🖄 🏘 🕒	🏟 💕 🗶 💥 🖓 🖓
SC Linker_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target]	Name				Value			
A & Embedded GDB (4/15/14 10:13 AM) (Suspended)	(x): a				100			
Thread [1] (Suspended) 1 main() e:\ranjith\linker_design\softconsole\linker_design_mss_cm3\linker_design	(x)+ b				50			
C\Microsemi\Libero v11.2\SoftConsole\Sourcery-G++\bin\arm-none-eabi-gdb.exe (4/15	(A)- C				805307580			
	(x)+ d				4 805307141			
	⇒ add_ptr				0x2000000	1		
	sub_ptr				0x2000002	9		
	mul_ptr				0x2000005	5		
	•							
····· •	4							
main.c 🛛 📄 debuq-in-microsemi-smartf 🛛 📄 debuq-in-microsemi-smartf 🛛 🔜 su	broutine.o	bl 🗋 🗖 🔡 Outli	e 🗟 Disassembly 🛛					
1#include "subroutine.h"		A 3						
2			00370 <main>:</main>	push {r7,	lr}			
3int main()			00372 <main+2>:</main+2>					
4 <b>(</b> 5			00374 <main+4>: =100:</main+4>	add r7, s	sp, #0			
5 int a,b,c,d,e;			-100; 00376 <main+6>:</main+6>	mov.w r3. d	100 ; 0x64			
7			0037a <main+10>:</main+10>					
<pre>8 int (*add_ptr)(int , int ) ;</pre>			-50;					
<pre>9 int (*sub_ptr)(int , int);</pre>			0037c <main+12>:</main+12>					
<pre>0 int (*mul_ptr)(int, int); 1</pre>			00380 <main+16>: dd ptr = &amp;add</main+16>	str r3,	r/, #4]			
2 a=100;			00382 <main+18>:</main+18>	movw r3, #	1			
3 b=50;			00386 <main+22>:</main+22>			00		
<pre>4 add_ptr = &amp;add</pre>			0038a <main+26>:</main+26>	str r3,	r7, #20]			
5 sub_ptr = ⊂ 6 mul_ptr = &mul			ub_ptr = ⊂ 0038c <main+28>:</main+28>		41 . 0020			
7 mui_per = amui,			00390 <main+32>:</main+32>			00		
8 c=(*add ptr)(a,b);			00394 <main+36>:</main+36>					
<pre>g d=(*sub_ptr)(a,b);</pre>			ul_ptr = &mul					
0 e=(*mul_ptr)(a,b);			00396 <main+38>: 0039a <main+42>:</main+42></main+38>					
2 while(1)			0039a <main+42>: 0039e <main+46>:</main+46></main+42>			100		
								,
Console 🖾 🖉 Tasks 🖹 Problems 🜔 Executables 🚺 Memory						× %	🔒 👪 🔜 🖳	1 : [ - [ - [
	onsole\Sourcery-G++\bin\arm-none-eabi-	db.exe (4/15/14 10:13 AM)						
er_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target] C:\Microsemi\Libero_v11.2\SoftC	,							
ker_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target] C:\Microsemi\Libero_v11.2\SoftC	,							,

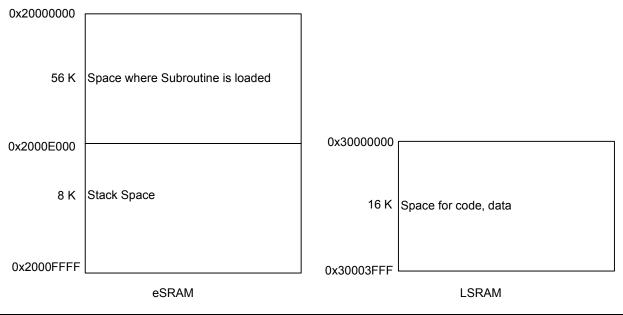
Figure 11 • Debug Window Showing Address Pointers Getting Updated for Implementation2

On performing step-by-step execution, the disassemble window shows these functions located in the eSRAM memory region and the appropriate values getting updated in the variable window as shown in Figure 12.

Debug - Linker_Design_MSS_CM3_app/subroutine.c - Microsemi SoftConsole IDE v3.4		
Edit Source Refactor Navigate Search Project Run Window Help		
3 • 🖃 👜 🐞 • 🖸 • 🏊 • 🍅 🖋 • 🌙 🖢 • 쥐 • 🌣 🗇 •	•	🖺 🏇 Debug 💀 C/0
Debug 🖄 💦 🍇 🛷 🕩 💷 🛤 🕉 💀 🕫 🗮 🝻 🕱 🏟 🍸 🗖	🕪 Variables 🛛 💊 Breakpoints 🗰 Registers 🛋 Modules	🖄 🍕 📄 🏟 🗳 💥 🏹 🔽
SC Linker_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target]	Name Value	
a 🔗 Embedded GDB (4/15/14 10:07 AM) (Suspended)	04 <sup>2</sup> a 100	
Invead [1] (Suspended) 2 add() e\ranjith\linker_design\softconsole\linker_design_mss_cm3\linker_design_	64 <sup>2</sup> b 50	
2 add) e:\ranjith\linker_design\softconsole\linker_design_mss_cm3\linker_design 1 main() e:\ranjith\linker_design\softconsole\linker_design_mss_cm3\linker_design	(v): x 100	
C:\Microsemi\Libero_v11.2\SoftConsole\Sourcery-G++\bin\arm-none-eabi-gdb.exe (4/15	60+ y 50 00≠ z 150	
	1.0	
	« [m	
m. •	(	
main.c 👔 debug-in-microsemi-smartfusion2- 👔 debug-in-microsemi-smartfusion2-	🗟 subroutine.o 🛛 😨 🔍 🖓 🖓 🚼 Outline 🔜 Disassembly 🕅	▽
#include "subroutine.h"	A {	
2	0x20000000 <add>: push {r7}</add>	
3 int add(int a, int b)	0x20000002 <add+2>: sub sp, #28</add+2>	
4{ 5 int x, y, z;	0x20000004 <add+4>: add r7, sp, #0 0x2000006 <add+6>: str r0, [r7, #4]</add+6></add+4>	
6 x=a;	0x20000008 <add+8>: str r1, [r7, #0]</add+8>	
7 y=b;	□ x=a;	
8 z=a+b;	0x2000000a <add+10>: ldr r3, [r7, #4]</add+10>	
9 return(z); 0)	<pre></pre>	
1	0x2000000e <add+14>: ldr r3, [r7, #0]</add+14>	
2 int sub(int a, int b)	0x20000010 <add+16>: str r3, [r7, #16]</add+16>	
3 (	z=a+b;	
4 int x, y, z; 5 x=a;	0x20000012 <add+18>: ldr r2, [r7, #4] 0x20000014 <add+20>: ldr r3, [r7, #0]</add+20></add+18>	
5 x=a; 6 v=b;	0x20000014 <add+205: #0]<br="" 1dr="" [f7,="" f3,="">0x20000016 <add+225: add="" r2<="" r3,="" td=""><td></td></add+225:></add+205:>	
7 z=a-b;	0x20000018 <add+24>: str r3, [r7, #20]</add+24>	
8 return(z);	return(z);	
9}	• 0x2000001a <add+26>: ldr r3, [r7, #20]</add+26>	
0 lint mul(int a, int b)	) 0x2000001c <add+28>: mov r0, r3</add+28>	
2{	0x2000001e <add+30>: mdv 10, 13 0x2000001e <add+30>: add.w r7, r7, #28</add+30></add+30>	
a de la calencia de l	· · · · · · · · · · · · · · · · · · ·	
Console 🛿 🧟 Tasks 💽 Problems 💽 Executables 🔋 Memory		× 🔆 🔓 🚮 🔛 🙂 🖅 🗗 • 🗅 • '
ker_Design_MSS_CM3_app Debug [Microsemi Cortex+M3 Target] C:\Microsemi\Libero_v11.2\SoftC	nsole\Sourcery-G++\bin\arm-none-eabi-gdb.exe (4/15/14 10:07 AM)	
¢		

Figure 12 • Debug Window Showing Memory Execution Address for Implementation2









## Stack in eSRAM and Splitting the Code Between eNVM and the Fabric LSRAM [Implementation3]

In this implementation, the entire code is loaded into the eNVM excluding subroutine (that is loaded into LSRAM) and the stack is in the eSRAM. To do this, the eNVM linker script needs to be modified as follows:

1. Add the LSRAM memory region in the memory command.

```
MEMORY
{
  /*
   * WARNING: The words "SOFTCONSOLE", "FLASH", and "USE", the colon ":",
^{\star} and the name of the type of flash memory are all in a specific order.
   * Please do not modify that comment line, in order to ensure
* debugging of your application will use the flash memory correctly.
   */
  /* SOFTCONSOLE FLASH USE: microsemi-smartfusion2-envm */
  rom (rx) : ORIGIN = 0x60000000, LENGTH = 256k
  /* SmartFusion2 internal eNVM mirrored to 0x00000000 */
  romMirror (rx) : ORIGIN = 0x00000000, LENGTH = 256k
  /* SmartFusion2 internal eSRAM */
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
  /* SmartFusion 2 LSRAM Block, This will store subroutines*/
  lsram (rwx) : ORIGIN = 0x30000000, LENGTH = 16k
}
Exclude the subroutine.o from loading into the eNVM
*(EXCLUDE FILE (*subroutine.o) .text.*)
Load the subroutine.o into the LSRAM.
.mytext :
  {
   *subroutine.o(.text.*)
} >lsram
```

The declared functions add, sub, and mul perform addition, subtraction, and multiplication respectively of two numbers.



Run the design to see the values returned by these function pointers as expected (pointers point to the LSRAM region) shown in Figure 14.

Debug - Linker_Design_MSS_CM3_app/main.c - Microsemi SoftConsole IDE v3.4							
Edit Source Refactor Navigate Search Project Run Window Help 3 ▼ 🔛 🗁 🙀 ▼ 🚱 ▼ 💁 ▼ 💁 🛷 ▼ 🔔 🖋 🖓 ▼ 💭 ▼ 🖓 ▼ 🗇 ♥ ♥ ↔				ff 体 Debug 晒 C/C+			
Debug 🛛 🥂 🍇 🖋 🖑 🕩 💷 😹 🔍 🕉 . 🕫 📰 🖬 🛠 🖉 🖓 🏹 🖓 🖓 🖓	00- Variables 🖄 🗣 Breakpoints 👯 R	egisters 🛋 Modules		£∋ 44 ⊟   🍫 🖇 💥 🎽 🗆 🗆			
SC Linker_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target]	Name		Value				
a Sembedded GDB (4/15/14 10:14 AM) (Suspended) a m <sup>0</sup> Thread [1] (Suspended)	(x)+ a			100			
1 miles [1] (suspended) 1 main() e\ranjith\linker_design\softconsole\linker_design_mss_cm3\linker_design	(x) b		50 1212				
C:\Microsemi\Libero_v11.2\SoftConsole\Sourcery-G++\bin\arm-none-eabi-gdb.exe (4/15,	(x)+ c (x)+ d		4				
	(x)- e		773				
	add_ptr		0x30000001				
	sub_ptr		0x30000029				
	mul_ptr		0x30000055				
<pre>2 31st main() 4( 5 5 7 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</pre>		0x00000374 <main- a=100; 0x00000376 <main- 0x00000376 <main- b=50; 0x0000037c <main- 0x00000380 <main- add_ptr = 6ac</main- </main- </main- </main- </main- 	<pre>+2&gt;: aub sp, #32 +4&gt;: aud r7, sp, #0 +6&gt;: mov.w r3, #100 ; 0x6 +6&gt;: mov.w r3, #100 ; 0x6 +10&gt;: str r3, [r7, #0] +12&gt;: mov.w r3, #50 ; 0x32 +16&gt;: str r3, [r7, #4] id;</pre>				
2 a=100; 3 b=50;			+18>: movw r3, #1 +22>: movt r3, #12288 ; 0x30	999			
4 add_ptr = &add			+26>: str r3, [r7, #20]				
5 sub_ptr = ⊂		sub_ptr = &su					
6 mul_ptr = amul; 7			+28>: movw r3, #41 ; 0x29 +32>: movt r3, #12288 ; 0x30	000			
<pre>8 c=(*add ptr)(a,b);</pre>			+36>: str r3, [r7, #241	*			
		• •					
Console 🛿 🖉 Tasks 🔝 Problems 🚺 Executables 🚦 Memory				× 🔆 📭 🖬 🔛 🙂 🖅 🖉 🕶 🖬 🕶 🖓			
ker_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target] C:\Microsemi\Libero_v11.2\SoftCo	nsole\Sourcery-G++\bin\arm-none-eabi-go	lb.exe (4/15/14 10:14 AM)					
n-none-eabi-sprite: Sent response: '00000000' n-none-eabi-sprite: Got packet: 'p20'							
m-none-eabi-sprite: Sent response: '00000000'							

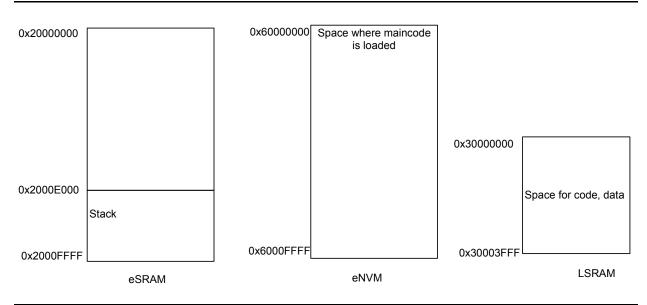
Figure 14 • Debug Window Showing Function Pointers for Implementation3



While performing a step-by-step execution, the disassembly window displays the functions located in the LSRAM memory region as shown in Figure 15.

Debug - Linker_Design_MSS_CM3_app/subroutine.c - Microsemi SoftConsole IDE v3.4					-	
ile Edit Source Refactor Navigate Search Project Run Window Help						
1 • 🔛 🔄 🐞 • 🛈 • 🏊 • 🄌 🛷 • 🥒 🖉 • 🖓 • 🖓 • 🖓	<b>x</b> )					😭 🏇 Debug 🐻 C/C+
🌣 Debug 🖄 💦 🍇 🖉 🕪 💷 🔳 🕼 🕉 🧔 🖉 🗮 🙀 💎 " 🗊	🕪 Variables 🕴 💊 Breakpoints 🚻 Registers 🛋 Modules				🖄 🏘 📄	🌼 🜮 🗙 💥 🔻 🖻 D
SC Linker_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target]	Name	Value				
a	66 <sup>1</sup> a	100				
2 add() e\ranjith\linker_design\softconsole\linker_design_mss_cm3\linker_design_i	d (%)	50 100				
1 main() e:\ranjith\linker_design\softconsole\linker_design_mss_cm3\linker_design	00: y	50				
C:\Microsemi\Libero_v11.2\SoftConsole\Sourcery-G++\bin\arm-none-eabi-gdb.exe (4/15)	60- z	150				
		-				+
main.c 📄 debug-in-microsemi-smartfusion2-	in subroutine.c Σ C Disassembly Σ	3				~ •
1#include "subroutine.h" 2	<pre>^ y=b; 0x3000000e <add+14></add+14></pre>	: 1dr r3, [r7	401			^
Sint add(int a, int b)	0x30000010 <add+14></add+14>					
4 {	z=a+b;					
5 int x, y, z; 6 x=a:	0x30000012 <add+18> = 0x30000014 <add+20></add+20></add+18>					
7 y=b;	0x30000016 <add+22></add+22>		, +0]			
8 z=a+b; 9 return(z):	0x30000018 <add+24></add+24>	: str r3, [r7	, #20]			
9 return(z); 10}	return(z);	• 1dm = 13 [m7	#201			
11	)		, 120]			
12 int sub(int a, int b)	0x3000001c <add+28></add+28>		12111			E
13 { 14 int x,y,z;	0x3000001e <add+30> 0x30000022 <add+34></add+34></add+30>		#28			
15 x=a;	0x30000024 <add+36></add+36>					
16 y=b;	0x30000026 <add+38></add+38>	: bx lr				
17 z=a-b; 18 return(z);	-					-
()	<u>ه</u>					Þ
Console 🛛 🖉 Tasks 🔛 Problems 🚺 Executables 🔋 Memory				××	🔒 🚮 🔛 🙂 🕞 🖱	] 🛃 📮 🕶 📑 🕶 "
nker_Design_MSS_CM3_app Debug [Microsemi Cortex-M3 Target] C:\Microsemi\Libero_v11.2\SoftCo	nsole\Sourcery-G++\bin\arm-none-eabi-gdb.exe (4/15/14 10:14 AM)					
rm-none-eabi-sprite: Sent response: '00000000' rm-none-eabi-sprite: Got packet: 'p20'						
m-none-eabi-sprite: Got packet: 'p20' m-none-eabi-sprite: Sent response: '00000000'						
-						
D*		Writable	Smart Insert	18:1		
		Writable	smart Insert	18:1		

Figure 15 • Debug Window Showing Execution Address for Implementation3







#### **Running the Implementations**

This application note provides the Design Files for all three implementations. There is one set of design files for the implementation1, and another set for the implementation2 and implementation3.

To get the design files for LSRAM (implementation1), refer to

http://soc.microsemi.com/download/rsc/?f=M2S\_AC417\_Implementation1\_DF

To get the design files for eSRAM-LSRAM and eNVM-LSRAM(implementation2 and 3), refer to http://soc.microsemi.com/download/rsc/?f=M2S\_AC417\_Implementation2\_3\_DF

Select the appropriate linker script in the Softconsole for running the above described scenarios.

 To run implementation 1 that includes running only in LSRAM, select the linker script as shown in Figure 17

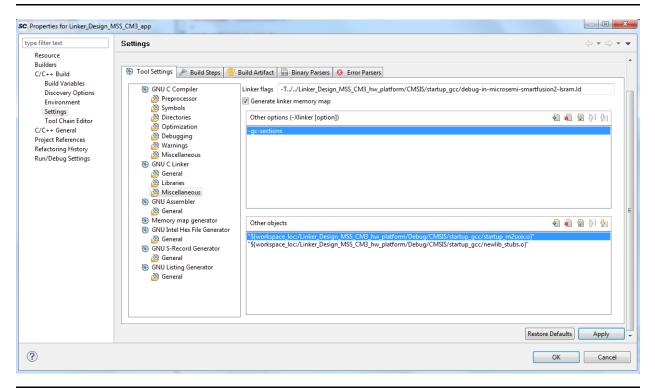


Figure 17 • Selecting Linker Script for Implementation1



2. To run implementation 2, which includes loading the subroutine into eSRAM and all other codes into the LSRAM, the eSRAM-LSRAM linker script needs to be selected as shown in Figure 18.

ilter text	Settings		⇒ → → ▼
source silders /C++ Build Build Variables Discovery Options Environment Settings Tool Chain Editor /C++ General oject References efactoring History un/Debug Settings	<ul> <li>Tool Settings Build Steps</li> <li>GNU C Compiler</li> <li>Preprocessor</li> <li>Symbols</li> <li>Directories</li> <li>Optimization</li> <li>Debugging</li> <li>Warnings</li> <li>Miscellaneous</li> <li>GNU C Linker</li> <li>General</li> <li>Libraries</li> <li>Miscellaneous</li> <li>GNU Assembler</li> <li>General</li> <li>Sinvershie Generator</li> <li>GNU Intel Her File Generator</li> <li>GNU Shecord Generator</li> <li>GNU Listing Generator</li> <li>GNU Shecord Generator</li> <li>GNU Listing Generator</li> <li>GNU Listing Generator</li> <li>GNU Listing Generator</li> <li>GNU Listing Generator</li> </ul>	Build Artifact       Binary Parsers       Error Parsers         Linker flags       -T/./Linker_Design_MSS_CM3_hw_platform/CMSIS/startup_gcc/debug-in-microsemi-sm         Image: Senerate linker memory map       Other options (-Xlinker [option])         Egc-sections       Image: Senerate linker memory map         Other options (-Xlinker [option])       Image: Senerate linker [option]         Image: Senerate linker memory map       Image: Senerate linker [option]         Other options (-Xlinker [option])       Image: Senerate linker [option]         Image: Senerate linker memory map       Image: Senerate linker [option]         Other options (-Xlinker [option])       Image: Senerate linker [option]         Image: Senerate linker [option]       Image: Senerate linker [o	<ul> <li>원 원 월 등 값</li> <li>원 원 월 등 값</li> </ul>
			Restore Defaults Apply

Figure 18 • Selecting Linker Script eSRAM-LSRAM for Implementation 2

- SC Properties for Linker\_Design\_MSS\_CM3\_app type filter text Settings ⇔ - ⇒ - -Resource Manage Configurations... Configuration: Debug [Active] Builders C/C++ Build **Build Variables Discovery Options** 🛞 Tool Settings 🎤 Build Steps P Build Artifact 🗟 Binary Parsers 😣 Error Parsers Environment Settings a 🛞 GNU C Compiler Linker flags -T../../Linker\_Design\_MSS\_CM3\_hw\_platform/CMSIS/startup\_gcc/debug-in-microsemi-smartfusion2-envm-Isram.Id Tool Chain Editor Preprocessor Generate linker memory map C/C++ General Project References Directories Other options (-Xlinker [option]) 🗐 🗐 🗟 🖗 🔄 Refactoring History Optimization
  Debugging Run/Debug Settings 🖉 Warnings Miscellaneous a 🛞 GNU C Linker 🖄 General Libraries Miscellaneous a 🛞 GNU Assembler 🚵 General Memory map generator Other objects 🕘 🌒 🗟 🖓 😓 🛞 GNU Intel Hex File Generator 🖄 General '\${workspace\_loc:/Linker\_Design\_MSS\_CM3\_hw\_platform/Debug/CMSIS/startup\_gcc/newlib\_stubs.o} A 🛞 GNU S-Record Generator 🖄 General a 🛞 GNU Listing Generator 🖄 General ? OK Cancel
- 3. To run implementation 3, where the code is loaded into eNVM and the subroutine is loaded into LSRAM in fabric, the eNVM-LSRAM linker script needs to be selected as shown in Figure 19.

Figure 19 • Selecting Linker Script eNVM-LSRAM for Implementation 3



## Speeding Up Code Execution by Copying into Internal SRAM at Boot-time

This section describes the method to load the code into internal SRAM before the execution of the code begins. This is done by using the Linker script production-relocate-executable.ld, which is available under the startup\_gcc folder as shown in Figure 20.

Project Explorer 🔀	■ 🕏 🍷 🗖
a 🎏 Linker_Design_MSS_CM3_app	
Binaries	
Includes	
Debug	
Release	
⊳ c main.c	
a 😂 Linker_Design_MSS_CM3_hw_platform	
Archives	
Includes	
🖌 🔁 CMSIS	
▲ bartup_gcc	
Description in the study of	
Is startup_m2scoc.s	
debug-in-external-ram.ld	
🔳 debug-in-microsemi-smartfu	usion2-envm-Isram.ld
🔳 debug-in-microsemi-smartfu	usion2-esram-Isram.Id
production-execute-in-place	
production-relocate-executation	ble.ld
⊳ 🖻 hw_reg_io.h	
⊳ 🖻 m2ssos.h	
Implement in the implementation is a second and interval in the implementation in the implementation is a second and in the implementation is c	
b isys_init_cfg_types.h	
b system_m2ssoc.c	
b system_m2sxxx.h	
Debug	
drivers	
b > drivers_config	
b > bal	
Linker_Design_hw_platform.h	
L	I

#### Figure 20 • Location of production-relocate-executable.ld

In the linker script under the memory command, there is an external\_ram section that is required to modify this section address according to the address of the SRAM implemented. For example, if the address is 0x30000000 and the size is 16 KB (as it is in this design example) modify the memory section as follows:

internal ram (rwx) : ORIGIN = 0x30000000, LENGTH = 16K



After this modification has been made in the memory command, AT command should be used for the Linker to identify which part of the code is to be loaded into the internal SRAM. For example, to load the data section into the internal SRAM, specify the memory region as follows:

```
.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 = .;
    } >internal_ram AT>rom
```

Here,  $internal_ram AT>rom$  means that the data is first loaded into the rom and that before the execution begins, when the Reset\_handler is run, this data will be copied into the  $internal_ram$ .

Relocation of the data section at runtime is done by the startup\_m2sxxx.s which has the copy data section code (shown below):

```
/*-----
* Copy data section.
*/
copy_data:
    ldr r0, =__data_load
    ldr r1, =__data_start
    ldr r2, =_edata
    cmp r0, r1
    beq clear_bss
copy_data_loop:
    cmp r1, r2
    itt ne
    ldrne r3, [r0], #4
    strne r3, [r1], #4
    bne copy_data_loop
```



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 E-mail: sales.support@microsemi.com Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for communications, defense and security, aerospace, and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs, and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; security technologies and scalable anti-tamper products; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, Calif. and has approximately 3,400 employees globally. Learn more at **www.microsemi.com**.

© 2014 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.