SmartFusion2 - Distributing and Running Code from Multiple Memory Regions

Table of Contents

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®2 System-on-Chip (SoC) field programmable gate array (FPGA) devices integrate an ARM® Cortex™-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.

© 2014 Microsemi Corporation

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® 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	• Windows XP SP2 Operating System - 32-bit/64-bit • Windows 7 Operating System - 32-bit/64-bit
Software Resources	
Libero® 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 0xDFFFFFFF) 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,

`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.

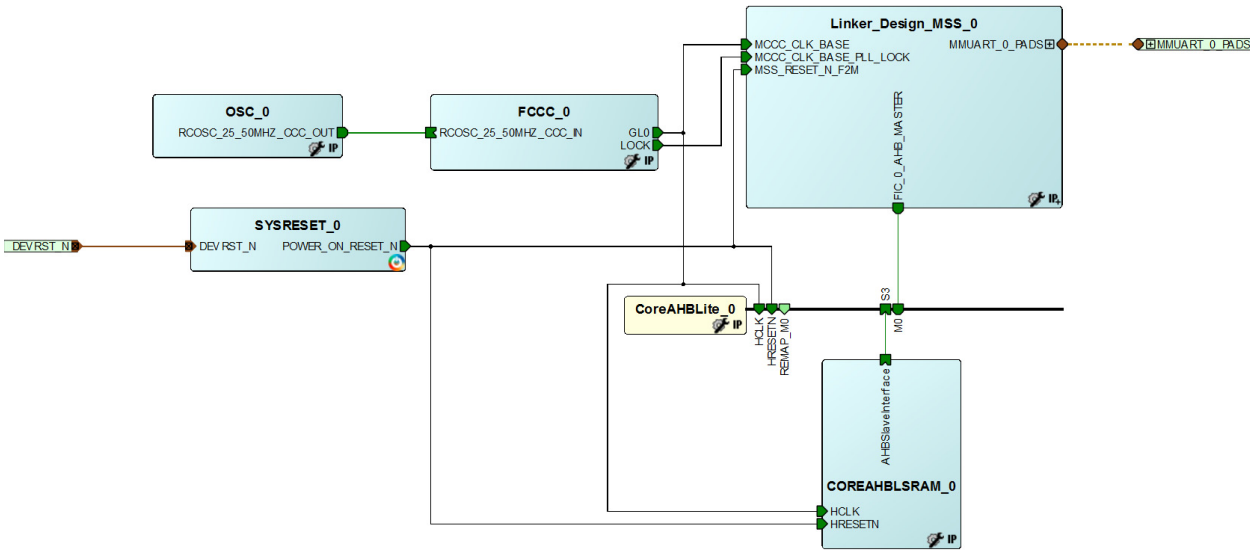

Figure 1 **• Top-Level SmartDesign**

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

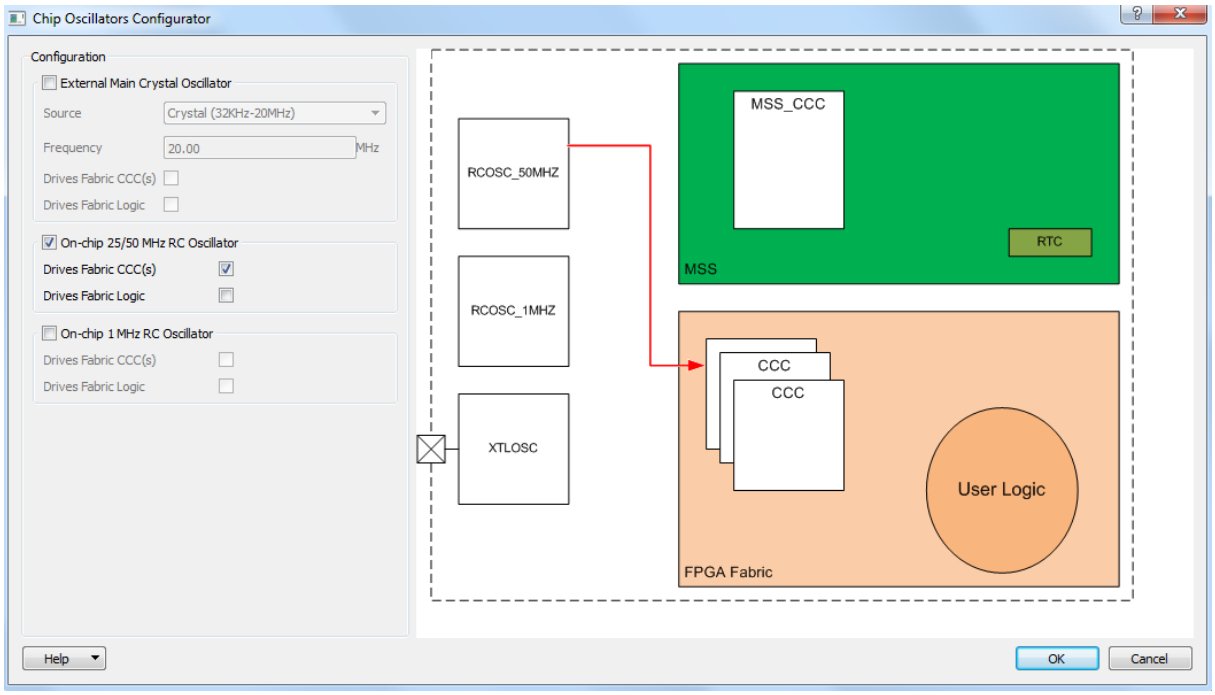

Figure 2 **• Oscillator Configuration**

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.

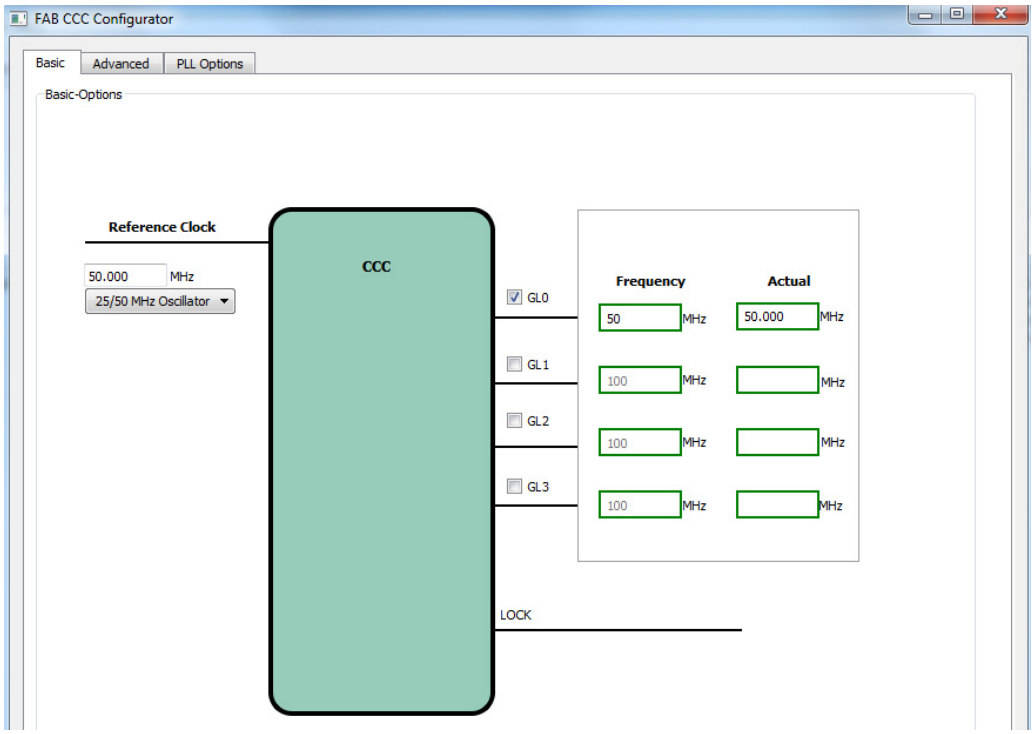

Figure 3 **• Fabric PLL Configuration**

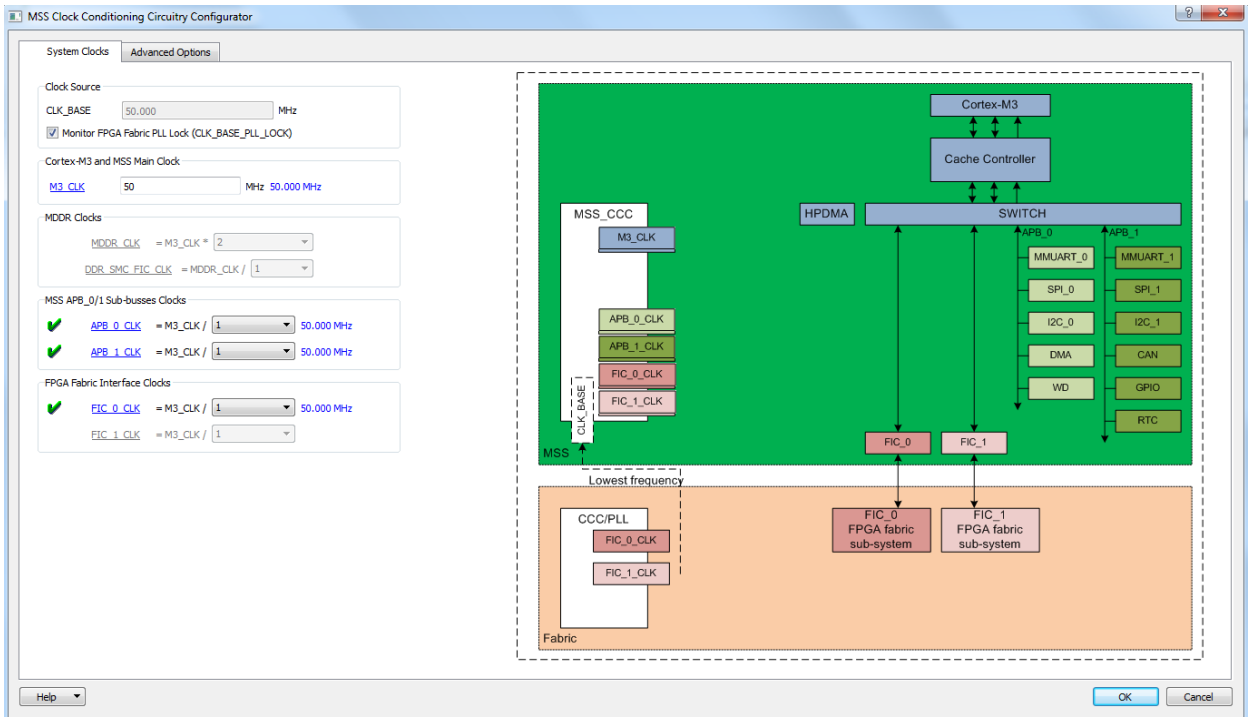

Figure 4 **• MSS CCC Configuration**

MSS reset is configured to 'Enable FPGA Fabric to MSS Reset (MSS_RESET_N_F2M)' as shown in Figure 5.

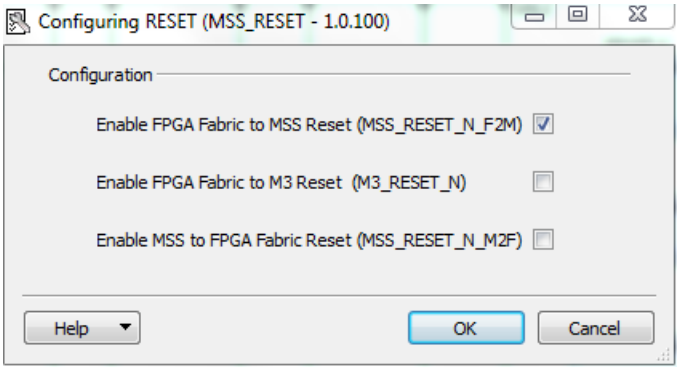

Figure 5 **• MSS RESET Configuration**

FIC_0 is configured to use the ABHLite master interface as shown in Figure 6.

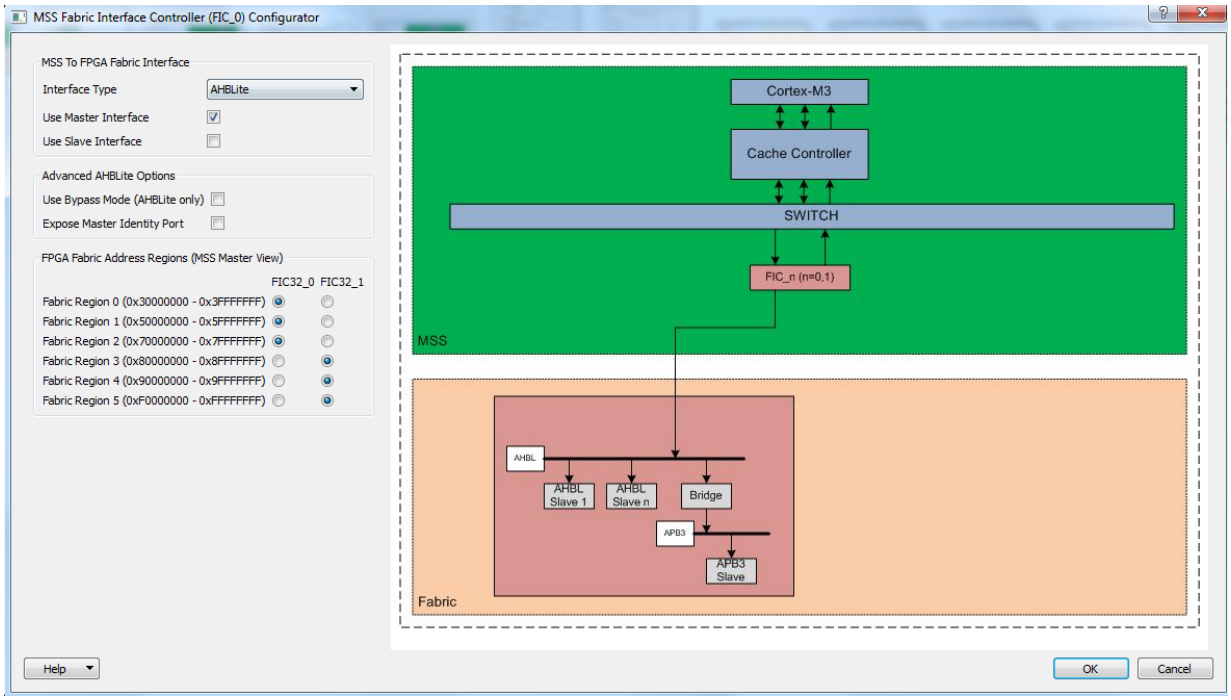

***Figure 6* • FIC Configuration**

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.

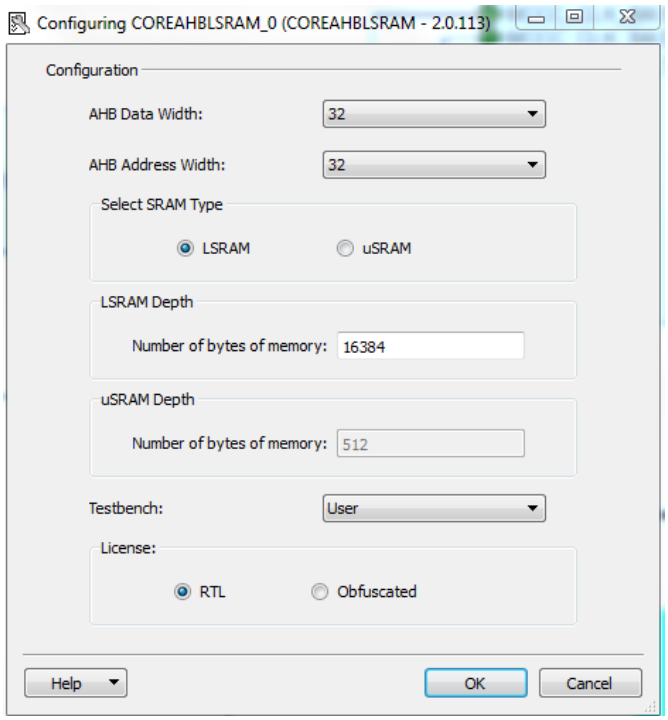

***Figure 7* • Fabric LSRAM Configuration**

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.

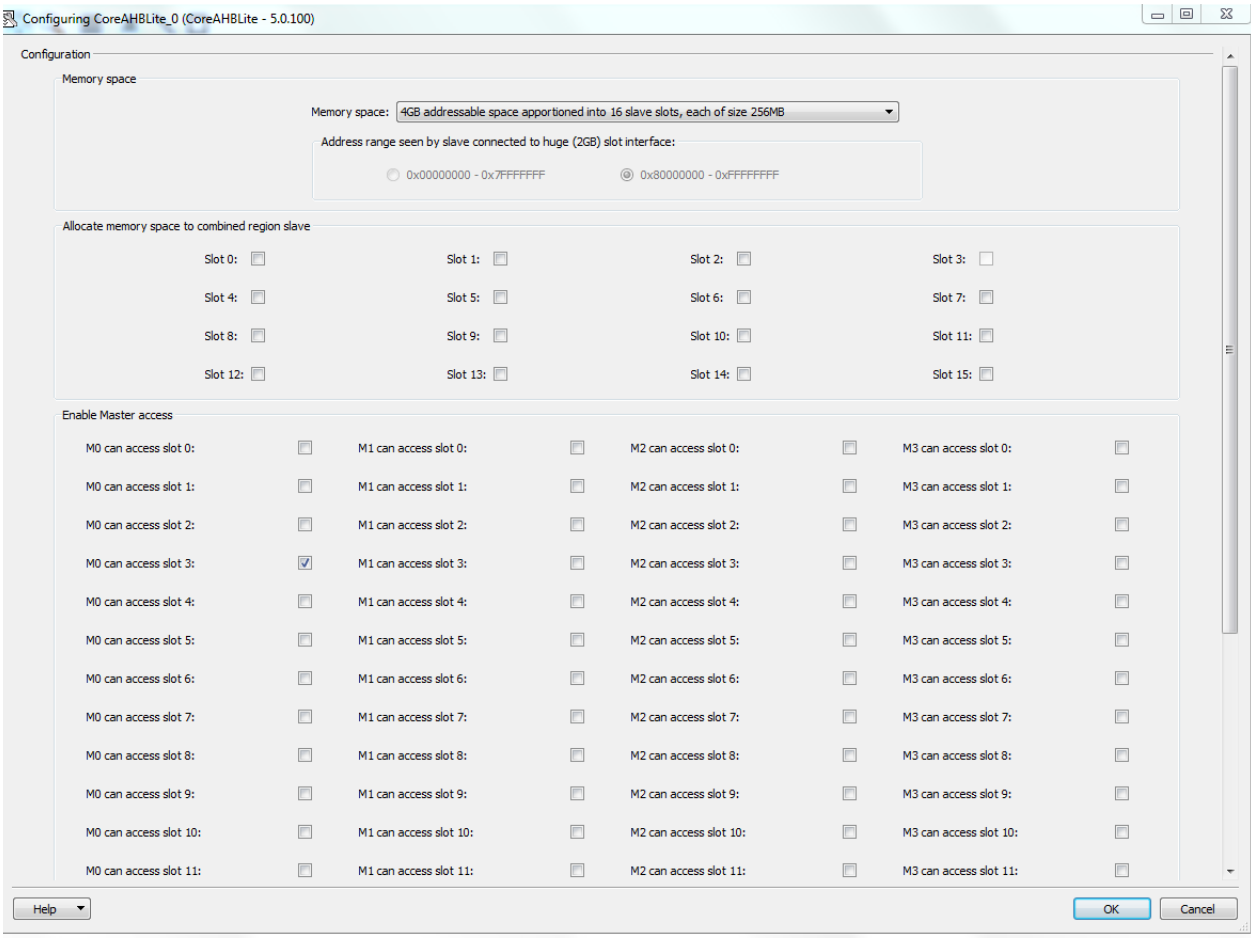

Figure 8 **• AHBLite Configuration**

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).*/
```

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 0x30000000, 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.

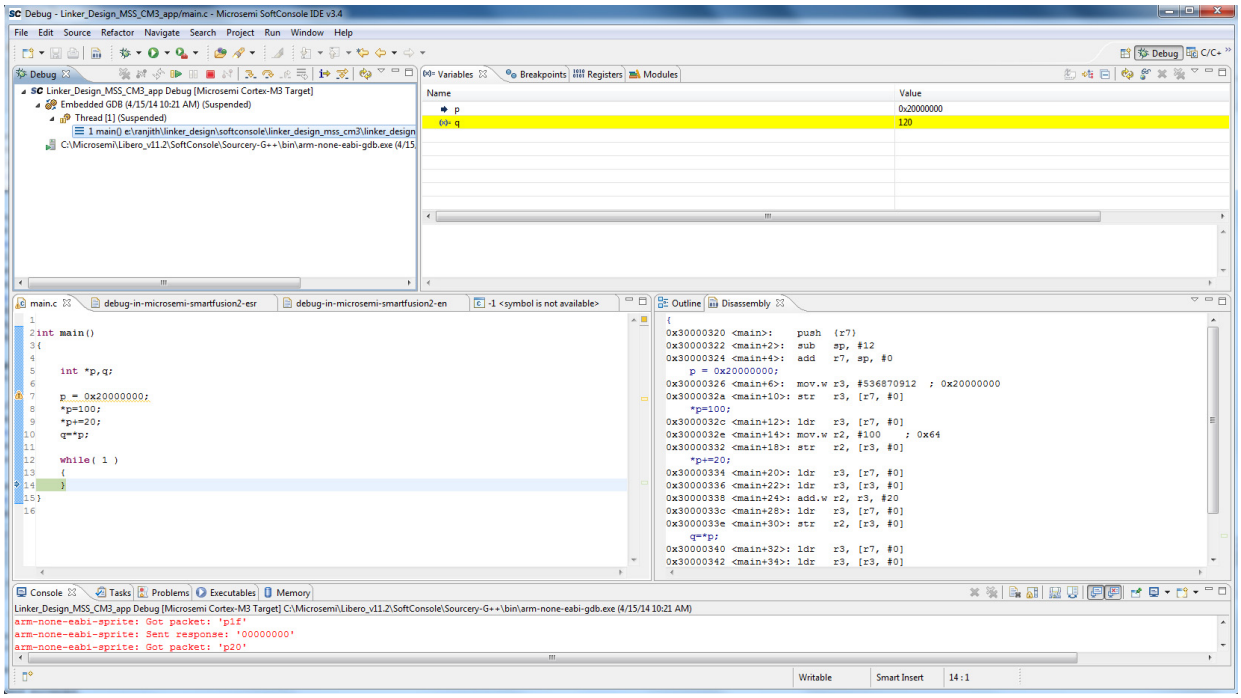

Figure 9 **• Softconsole Debug Showing Memory Address for Implementation1**

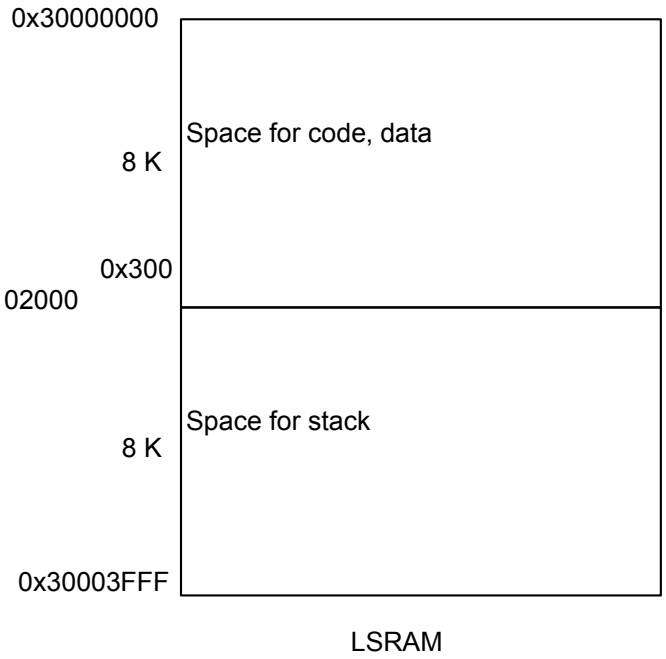

Figure 10 **• Memory Map for Implementation1**

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 = 0x30000000, LENGTH = 16k
  esram (rwx) : ORIGIN = 0x20000000, 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).*/
```

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 `.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.

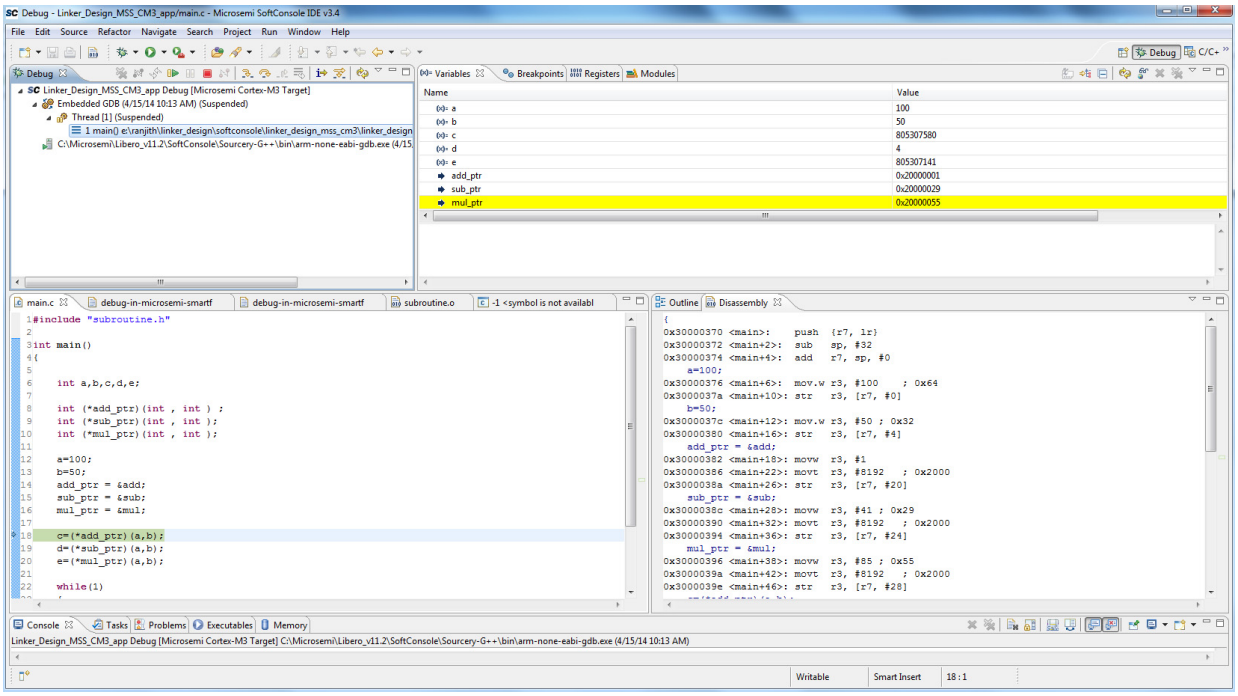

Figure 11 **• Debug Window Showing Address Pointers Getting Updated for Implementation2**

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.

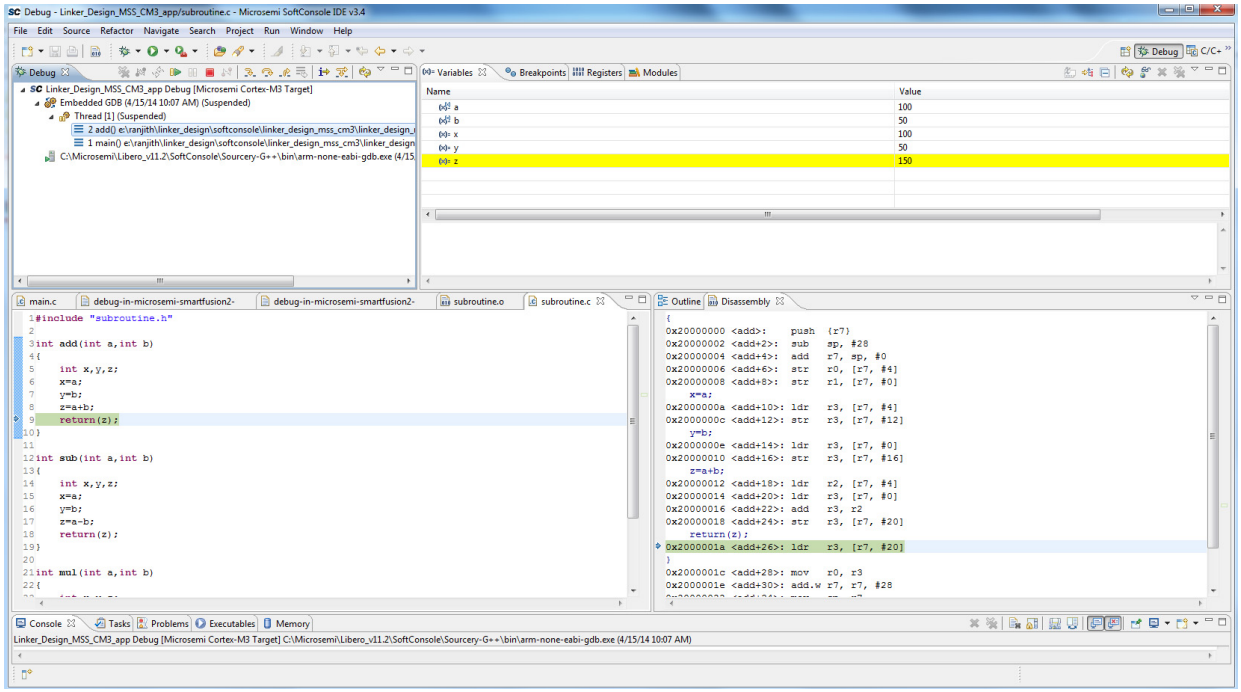

Figure 12 **• Debug Window Showing Memory Execution Address for Implementation2**

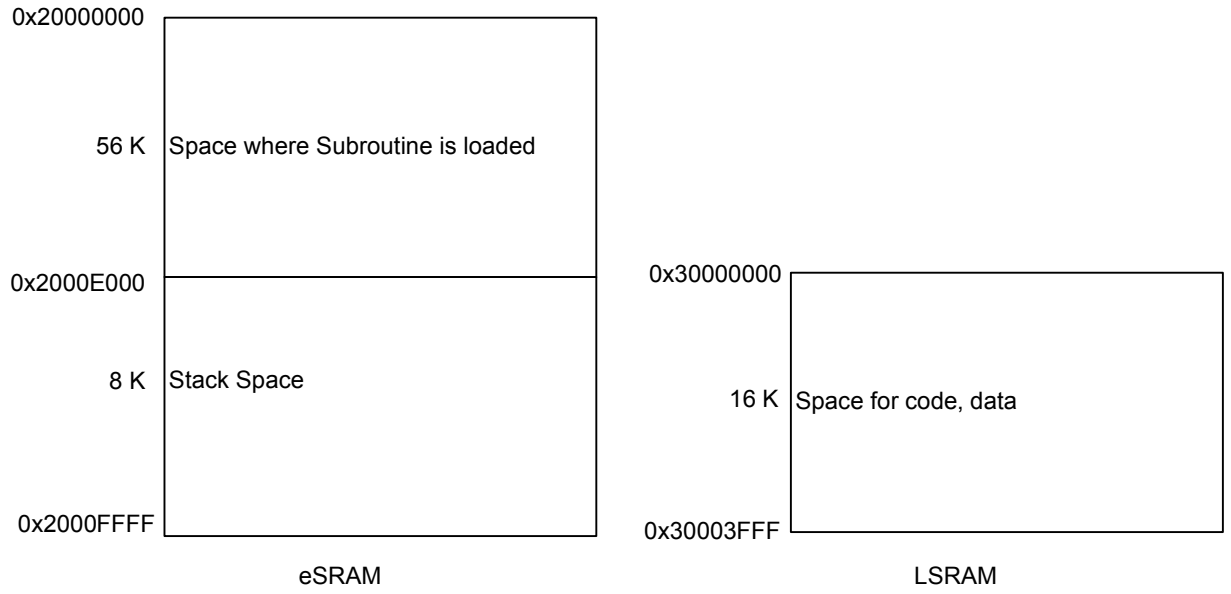

Figure 13 **• Memory Map for Implementation2**

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 ":",
* 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.

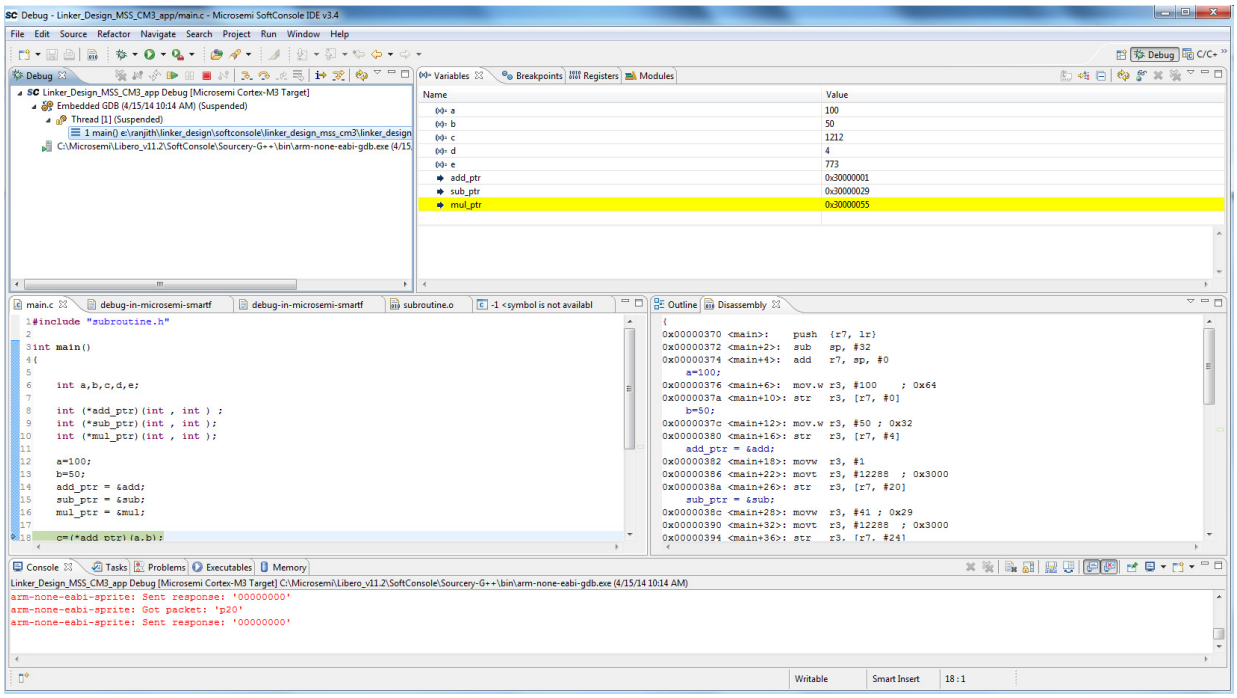

Figure 14 **• Debug Window Showing Function Pointers for Implementation3**

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

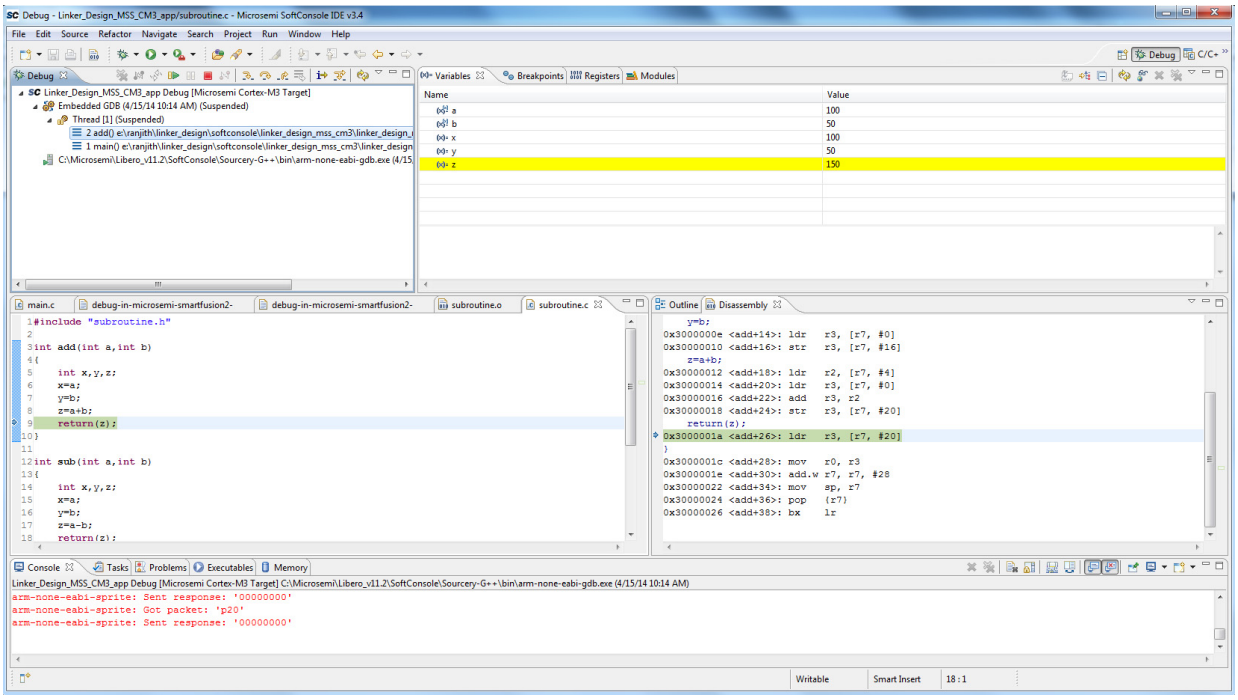

Figure 15 **• Debug Window Showing Execution Address for Implementation3**

Figure 16 **• Memory Map 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.

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

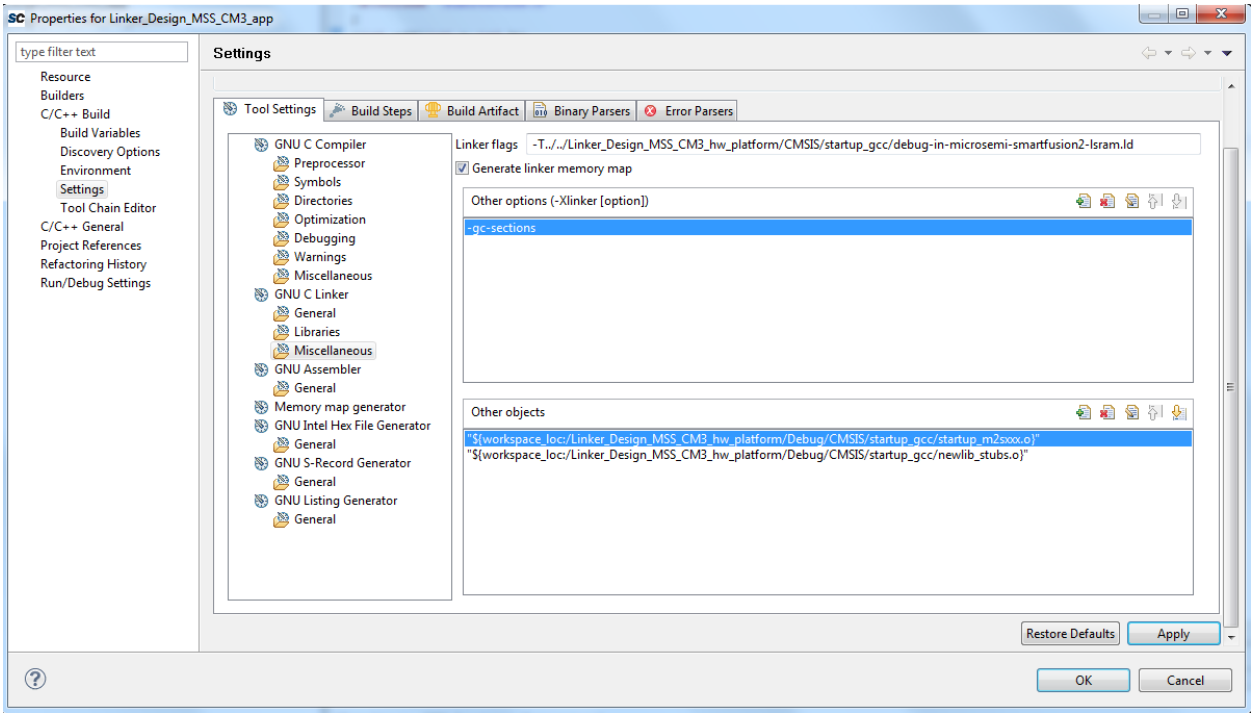

Figure 17 **• Selecting Linker Script for Implementation1**

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.

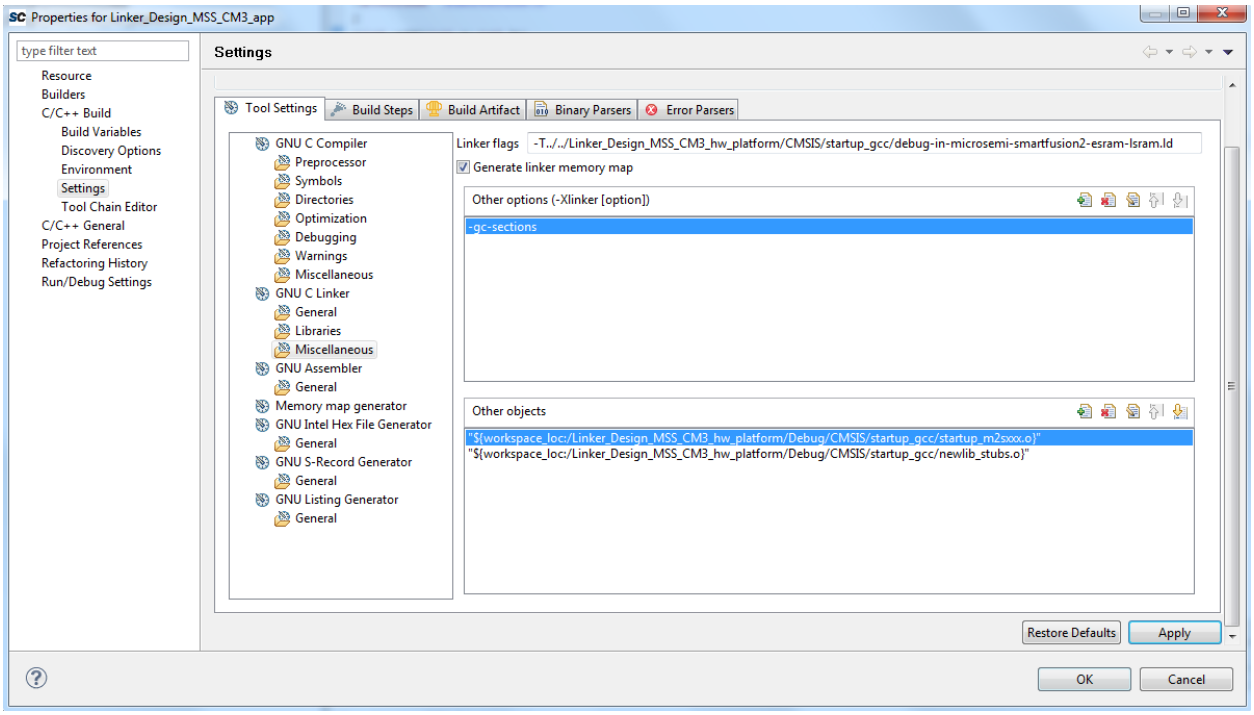

Figure 18 **• Selecting Linker Script eSRAM-LSRAM for Implementation 2**

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.

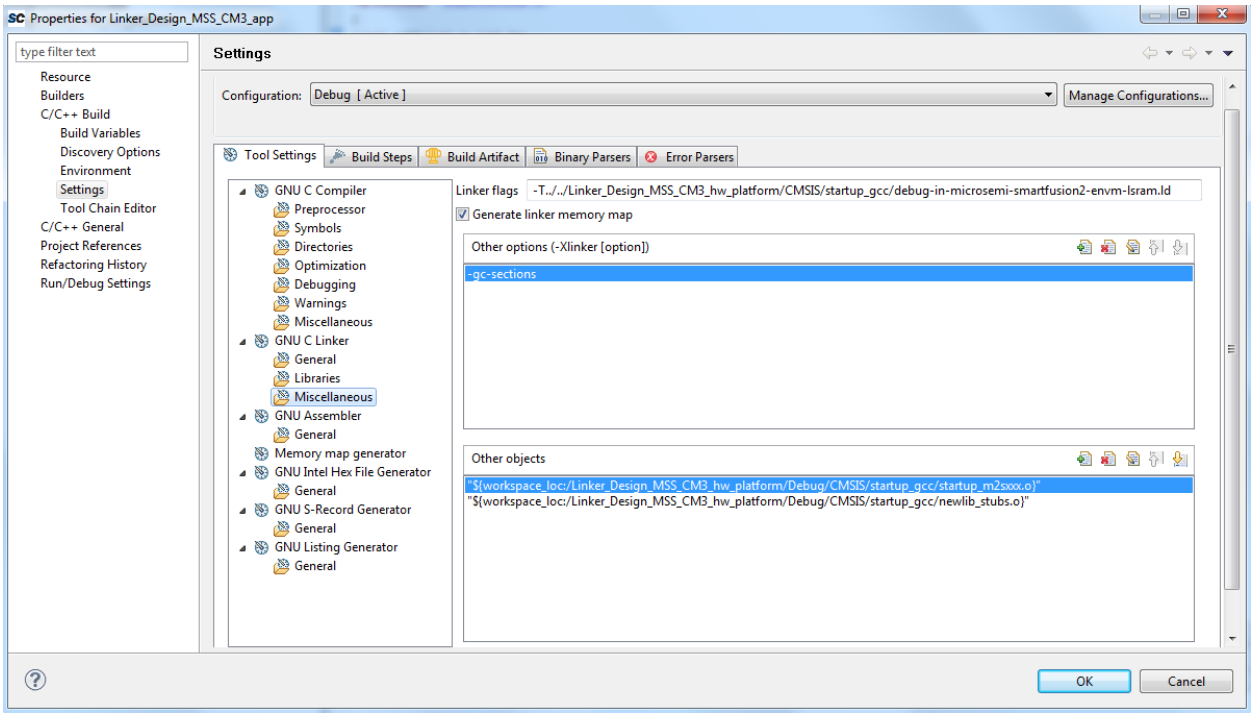

Figure 19 **• Selecting Linker Script eNVM-LSRAM for Implementation 3**

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.

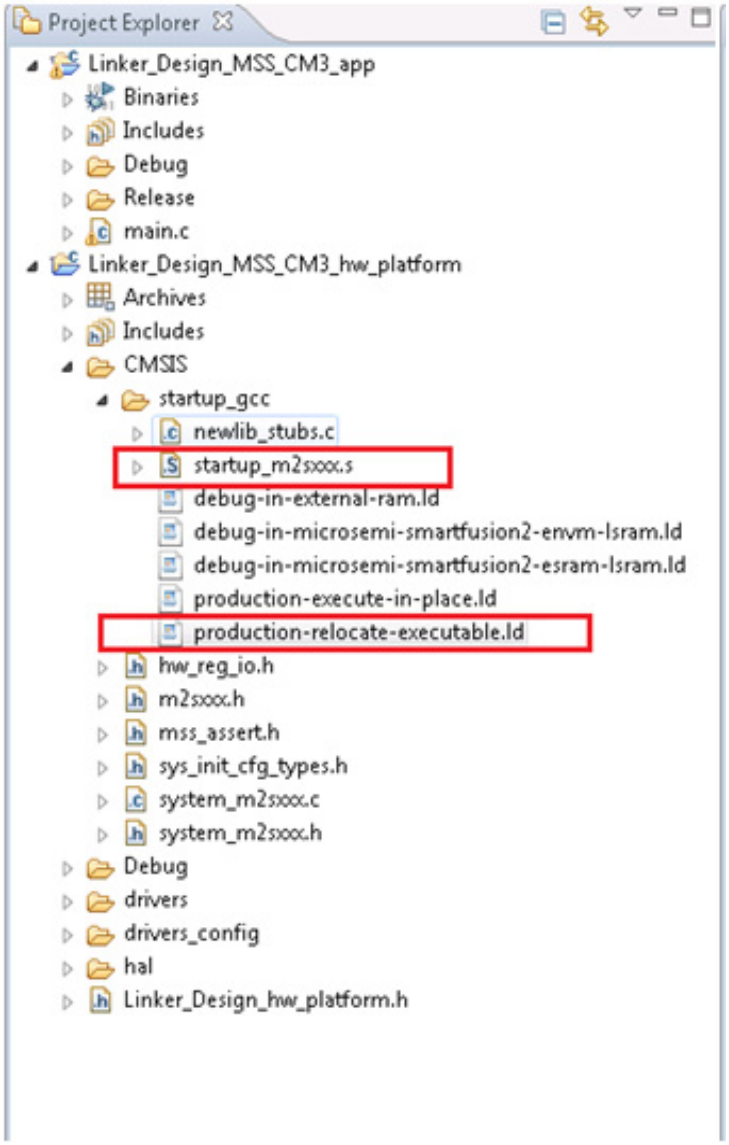

***Figure 20* • Location of production-relocate-executable.ld**

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.

51900285-1/04.14