Relocating Code and Data Using the CW GCC Linker File (.ld) for Kinetis

By: Carlos Musich

1) Introduction

This document provides guidance for relocating Code and Data within the MCU memory map. As part of this process it explains how create new memory segments and sections by editing the GCC Linker File (.ld) for Kinetis Architectures.

For detailed information on the GCC Linker please refer to "*The GNU Linker*" by Steve Chamberlain and Ian Lance Taylor.

You can see ***Porting Freescale ARM Compiler-based Projects to use ARM GCC - Porting_ARM_GCC.pdf*** for information about the main differences between Freescale ARM and GCC linker files. You can find it in *{CW10.x installation path}\MCU\Help\PDF*

2) Preliminary Background

A linker or link editor is a program that takes one or more objects generated by the compiler to combine them, relocate their data and tie up symbol references to generate an executable file. This is usually the last step in compiling a program, to do it the linker uses a linker file or linker script. In order to relocate code and data in a specific memory area it is necessary to edit the linker file.

The following chapters explain how the linker place functions in the memory and how to relocate them in flash, internal RAM, and external RAM using K60 or K70 Kinetis devices with CodeWarrior and GCC toolchain.

Contents

3) Linker File (.ld) Overview

Freescale linker files are divided in 2 main parts.

3.1) Memory Segment

The memory segment is used to divide the Microcontroller memory into segments. Each segment can have read, write and execute attributes. The address and the length of each segment are defined as well. An example is shown in listing 1.

```
MEMORY
{
  m_interrupts   (rx) : ORIGIN = 0x00000000, LENGTH = 0x1E8
  m_cfmprotrom   (rx) : ORIGIN = 0x00000400, LENGTH = 0x10
  m_text         (rx) : ORIGIN = 0x00000800, LENGTH = 1M - 0x800
  m_data         (rwx) : ORIGIN = 0x1FFF0000, LENGTH = 64K
  m_data2        (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}
```

Listing 1 – K70 Memory segment

3.2) Sections Segment

In sections segment are defined the contents of target-memory sections. In other words, a section indicates which parts of your application will be allocated in each memory segment. Main sections are '*.text*' which contains all the code and the constants of an application, '*.data*' which contains all initialized data, and '*.bss*' which contains all non-initialized data.

Below you can see section '*.text*' of an application using K70. As you can notice it is contained in segment '*m_text*'.

```
.text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    *(.rodata)         /* .rodata sections (constants, strings, etc.) */
    *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
    *(.glue_7)         /* glue arm to thumb code */
    *(.glue_7t)        /* glue thumb to arm code */
    *(.eh_frame)

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

    . = ALIGN(4);
    _etext = .;        /* define a global symbols at end of code */
  } > m_text
```

Listing 2 – K70 section .text

4) Relocating Code

The code generated by the compiler is usually placed in section '*.text'*. Sometimes, however it is necessary to have certain particular functions to appear in special sections or in a specific address. The '*section*' attribute specifies that a function lives in a particular section. e.g.

```
void vfnDummy (void) __attribute__ ((section ("mySec")));
```

The example above places function '*vfnDummy*' in section '*mySec*'.

In this application note we are going to write 6 functions that toggle the TWR-K60 or TWR-K70 on board LEDs when pushing an onboard switch (SW2). Such functions are going to be allocated/relocated in different memory areas.

4.1) Prerequisites

- Create a new bareboard project using K60 or K70 and be sure you select GCC toolchain in the New Project wizard.
- Before using the GPIOs you need to initialize them, use function **init_gpio()** shown in listing 3 for this purpose. You will also need function **delay()** shown in listing 4 to provide a short delay. The following defines are necessary as well.

```
#define GPIO_PIN_MASK           0x1Fu
#define GPIO_PIN(x)             (((1)<<(x & GPIO_PIN_MASK)))
```

```
void init_gpio()
{
        /* Enable all of the port clocks */
        SIM_SCGC5 |= (SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTB_MASK | SIM_SCGC5_PORTC_MASK |
        SIM_SCGC5_PORTD_MASK | SIM_SCGC5_PORTE_MASK | SIM_SCGC5_PORTF_MASK );
        // Set PTD0 and PTE26 (connected to SW1 and SW2) for GPIO functionality, falling IRQ,
        // and to use internal pull-ups. (pin defaults to input state)
        PORTD_PCR0=PORT_PCR_MUX(1)|PORT_PCR_IRQC(0xA)|PORT_PCR_PE_MASK|PORT_PCR_PS_MASK;
        PORTE_PCR26=PORT_PCR_MUX(1)|PORT_PCR_IRQC(0xA)|PORT_PCR_PE_MASK|PORT_PCR_PS_MASK;

        // Set PTA10, PTA11, PTA28, and PTA29 (connected to LED's) for GPIO functionality
        PORTA_PCR10=(0|PORT_PCR_MUX(1));
        PORTA_PCR11=(0|PORT_PCR_MUX(1));
        PORTA_PCR28=(0|PORT_PCR_MUX(1));
        PORTA_PCR29=(0|PORT_PCR_MUX(1));

        // Change PTA10, PTA11, PTA28, PTA29 to outputs */
        GPIOA_PDDR=GPIO_PDDR_PDD(GPIO_PIN(10) | GPIO_PIN(11) | GPIO_PIN(28) | GPIO_PIN(29) );
}
```

Listing 3 – Function init_gpio

```
void delay()
{
        unsigned int i, n;
        for(i=0;i<3000;i++)
        {
                for(n=0;n<1000;n++)
                {
                        asm("nop");
                }
        }
}
```

Listing 4 – Function delay

Call function **init_gpio** from function **main**, then enter in and endless loop calling function **delay** inside the loop. Function **main** must look as shown in listing 5.

```
int main (void)
{
        init_gpio();
        while(1)
        {
                delay();
        }
        return 0;
}
```

Listing 5 – Function main

Go to menu Project > Build Configurations > Set Active > FLASH to select flash configuration. Then go to menu Project > Build Project to build the project. You can alternately click the hammer button.

4.2) Relocating Code in ROM

Listing 6 shows function **toggle_LED_allocated_in_Flash** which toggles blue LED 3 times. Copy this function into your project and call it each time SW2 is pressed. Function **main** must look as shown in listing 7.

```
/*Toggle LEDs - This functions toggles blue LED*/
void toggle_LED_allocated_in_Flash()
{
        unsigned int x;
        for(x = 0; x < 6; x++ )
        {
                GPIOA_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(10));
                delay();
        }
}
```

Listing 6 – Function toggle_LED_allocated_in_Flash

```
int main (void)
{
        init_gpio();
        while(1)
        {
                //Look at status of SW2 on PTE26
                if((GPIOE_PDIR & GPIO_PDIR_PDI(GPIO_PIN(26)))==0)  //If pressed...
                {
                        toggle_LED_allocated_in_Flash();
                }
                delay();
        }
        return 0;
}
```

Listing 7 – Function main

Go to menu Project > Build Project and then search for the *.map file inside {Project_path}/FLASH. Here you can see that function **toggle_LED_allocated_in_Flash** is placed in a flash address. This is shown in listing 8.

```
.text.toggle_LED_allocated_in_Flash
                0x000009a0       0x40 ./Sources/main.o
                0x000009a0                toggle_LED_allocated_in_Flash
```

Listing 8 – Function toggle_LED_allocated_in_Flash in map file

Now we are going to use attribute 'section' to create a section. This time the section is caled '*.myROM*' and use it to relocate a function that toggles on-board green LED in address 0x000FF000. Listing 9 shows how this function should see.

```
/*Toggle LEDs - This functions toggles green LED and is relocated in Flash address 0x00007F00*/
__attribute__ ((section(".myROM"))) void toggle_LED_relocated_in_Flash_address_0x000FF000()
{
        unsigned int x;
        for(x = 0; x < 6; x++ )
        {
                GPIOA_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(29));
                delay();
        }
}
```

Listing 9 – Function toggle_LED_relocated_in_Flash_address_0x000FF000

Now we need to edit linker file (.ld) to create a new segment where this function is going to be relocated. Compare listing 10 with listing 1 and notice that 0x1000 bytes were subtracted from segment '*m_text*' to create segment '*my_text*'.

```
MEMORY
{
  m_interrupts   (rx) : ORIGIN = 0x00000000, LENGTH = 0x1E8
  m_cfmprotrom   (rx) : ORIGIN = 0x00000400, LENGTH = 0x10
  m_text         (rx) : ORIGIN = 0x00000800, LENGTH = 0x1M - 0x1800
  my_text        (rx) : ORIGIN = 0x000FF000, LENGTH = 0x1000              /* New ROM Segment */
  m_data         (rwx) : ORIGIN = 0x1FFF0000, LENGTH = 64K
  m_data2        (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}
```

Listing 10 – Memory segment edited to create segment 'my_text'

Now create a new section in linker file to place '*.myROM*' content. You can call this section '*.my_ROM*' and write it just before section '*.data*'.

```
/* Section created to relocate code in specific Flash address  */
  .my_ROM :
  {
        . = ALIGN(4);
        *(.myROM)
        . = ALIGN(4);

  } > my_text
```

Listing 11 – Section .my_ROM

Finally, write a call to function **toggle_LED_relocated_in_Flash_address_0x000FF000** after SW2 is pressed in function **main**.

```
int main (void)
{
        init_gpio();
        while(1)
        {
                //Look at status of SW2 on PTE26
                if((GPIOE_PDIR & GPIO_PDIR_PDI(GPIO_PIN(26)))==0)  //If pressed...
                {
                        toggle_LED_allocated_in_Flash();
                        toggle_LED_relocated_in_Flash_address_0x000FF000();
                }
                delay();
        }
        return 0;
}
```

Listing 12 – Section .my_ROM

Go to menu Project > Build Project and then search for the *.map file inside {Project_path}/FLASH. Here you can see that function **toggle_LED_relocated_in_Flash_address_0x000FF000** is placed exactly where it is expected and it is 40 bytes long. This is shown on listing 13.

```
.my_ROM         0x000ff000       0x40
                0x000ff000                . = ALIGN (0x4)
 *(.myROM)
 .myROM         0x000ff000       0x40 ./Sources/main.o
                0x000ff000                toggle_LED_relocated_in_Flash_address_0x000FF000
                0x000ff040                . = ALIGN (0x4)
```

Listing 13 – Function toggle_LED_relocated_in_Flash_address_0x000FF000 in map file

4.3) Relocating Code in RAM

Sometimes it is required to copy code to RAM for faster execution. The easiest way to do this is to add the function to section '.data'. Doing this the linker will place the function in the first address available in RAM and the startup code will copy the function form Flash to RAM automatically. Please notice that this approach only works if you don't need to place the function in any specific RAM address. Next chapter discusses how to relocate a function in a specific RAM location.

There are just 2 steps:

- Use attribute '*section*' to place the function in a new section. In this case it is called '*.mydata*' as shown in listing 14.
- Add content of '*.mydata*' in '*.data*' section as shown on listing 15.

```
/*Toggle LEDs - This functions toggles orange and is relocated in RAM*/
__attribute__ ((section(".mydata"))) void toggle_LED_relocated_in_any_RAM_address()
{
        int x;
        for(x = 0; x < 6; x++ )
        {
                GPIOA_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(28));
                delay();
        }
}
```

Listing 14 – Function toggle_LED_relocated_in_any_RAM_address

```
/* Initialized data sections goes into RAM, load LMA copy after code */
  .data : AT(___ROM_AT)
  {
    . = ALIGN(4);
    _sdata = .;        /* create a global symbol at data start */
    *(.data)           /* .data sections */
    *(.data*)          /* .data* sections */
    *(.mydata)        /* .mydata relocates a function on any RAM address */
    . = ALIGN(4);
    _edata = .;        /* define a global symbol at data end */
  } > m_data
```

Listing 15 – Adding content of '*.mydata*' in '*.data*' section

Write a call to function **toggle_LED_relocated_in_any_RAM_address** after SW2 is pressed in function **main**.

```
int main (void)
{
        init_gpio();
        while(1)
        {
                //Look at status of SW2 on PTE26
                if((GPIOE_PDIR & GPIO_PDIR_PDI(GPIO_PIN(26)))==0)  //If pressed...
                {
                        toggle_LED_allocated_in_Flash();
                        toggle_LED_relocated_in_Flash_address_0x0000FF00();
                        toggle_LED_relocated_in_any_RAM_address();
                }
                delay();
        }
        return 0;
}
```

Listing 16 – Function main

Go to menu Project > Build Project and then search for the *.map file inside {Project_path}/FLASH. Here you can see that function **toggle_LED_relocated_in_any_RAM_address** is placed in RAM and it is 40 bytes long. This is shown on listing 17.

```
.data           0x1fff0000       0x50 load address 0x00001c2c
                0x1fff0000                . = ALIGN (0x4)
                0x1fff0000                _sdata = .
 *(.data)
 *(.data*)
 *(.mydata)
 .mydata        0x1fff0000       0x40 ./Sources/main.o
                0x1fff0000                toggle_LED_relocated_in_any_RAM_address
```

Listing 17 – Function toggle_LED_relocated_in_any_RAM_address in map file

4.4) Relocating Code in a specific RAM address

To relocate a function in a specific RAM section it is necessary to create a new memory segment and a new section in the linker file. Be aware that you must save this function in Flash and then it must be copied to RAM. The startup code can do this copy for us if we edit section '*.romp*' in the linker file with the correct information.

The first step is to write the function using attribute '*section*'. In this example the section is called '*myRAM*'.

```
/*Toggle LEDs - This functions toggles red LED and is relocated in RAM address 0x1FFFE000*/
__attribute__ ((section(".myRAM"))) void toggle_LED_relocated_in_RAM_address_0x1FFFE000()
{
        unsigned int x;
        for(x = 0; x < 6; x++ )
        {
                GPIOA_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(11));
                delay();
        }
}
```

Listing 18 – Function toggle_LED_relocated_in_RAM_address_0x1FFFE000

Next step is to create a segment in the linker file. This segment must start in the address where the function needs to be relocated; in this case it is address 0x1FFFE00.
Notice that 0x2000 bytes are subtracted from segment '*m_data*' to create segment '*my_data*'.

```
MEMORY
{
  m_interrupts   (rx) : ORIGIN = 0x00000000, LENGTH = 0x1E8
  m_cfmprotrom   (rx) : ORIGIN = 0x00000400, LENGTH = 0x10
  m_text         (rx) : ORIGIN = 0x00000800, LENGTH = 0x1M - 0x800
  m_data         (rwx) : ORIGIN = 0x1FFF0000, LENGTH = 0xE000
  my_data        (rwx) : ORIGIN = 0x1FFFE000, LENGTH = 0x2000            /* New RAM Segment */
  m_data2        (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}
```

Listing 19 – Memory segment edited to add segment my_data

Then a section must be created in the linker file, this section must be contained in segment '*my_data*', let us call it '*.my_ram*'. As it was mentioned before, the function must be saved in flash and copied to RAM in run time. The instruction '*AT*' indicates the flash address where the function will be resident before being copied.

If you search the for label '*___ROM_AT*' in the linker file you will find that it points to the first available address in flash, here is where section '*.data*' is resident, therefore, the address where section '*.my_ram*' must reside is after section '*.data*'. To calculate this address the instruction '*SIZEOF*' is used. This is shown in listing 20.

```
 /* Section created to relocate code in specific RAM address */
.my_ram :  AT(___ROM_AT + SIZEOF(.data))
{
      . = ALIGN(4);
      _mySection = .;        /* create a global symbol at myRAM */
      *(.myRAM)
      . = ALIGN(4);

} > my_data
```

Listing 20 – Section .my_ram

As we inserted a new RAM section that was not considered by the linker we must make a couple of adjustments.

At this moment label `___m_data2_ROMStart` overlaps with the address where section '*.my_ram*' resides. This label must be edited to skip section '*.my_ram*'.

```
___m_data2_ROMStart = ___ROM_AT + SIZEOF(.data) + SIZEOF(.my_ram);
```

Label `_romp_at` must also be edited to consider section '*.my_ram*'.

```
 _romp_at = ___ROM_AT + SIZEOF(.data) + SIZEOF(.user_data2) + SIZEOF(.my_ram);
```

The last step is to copy the code from flash to RAM. To do this it is necessary to edit section '*.romp*'. This section indicates to the startup code what is going to be copied from ROM to RAM. This copy table, which the symbol __S_romp identifies, contains a sequence of three word values per entry:

- ROM start address
- RAM start address
- Size

The last entry in this table must be all zeros, this is the reason for the three lines, `LONG(0)` before the table closing brace character.

For the new section to be copied, one new entry must be added to the table. The new entry indicates `___ROM_AT + SIZEOF(.data)` as the source flash address, label `_mySection` as the destiny RAM address and `SIZEOF(.my_ram)` as the size of the section to be copied.

```
_romp_at = ___ROM_AT + SIZEOF(.data) + SIZEOF(.user_data2) + SIZEOF(.my_ram);
.romp : AT(_romp_at)
{
  __S_romp = _romp_at;
  LONG(___ROM_AT);
  LONG(_sdata);
  LONG(___data_size);
  LONG(___m_data2_ROMStart);
  LONG(___m_data2_RAMStart);
  LONG(___m_data2_ROMSize);
  LONG(___ROM_AT + SIZEOF(.data));
  LONG(_mySection);
  LONG(SIZEOF(.my_ram));
  LONG(0);
  LONG(0);
  LONG(0);
} > m_data2
```

Listing 21 – Section .romp

In function **main** write a call to function **toggle_LED_relocated_in_RAM_address_0x1FFFE000** after SW2 is pressed.

If you have added the functions in previous chapters, function main should look as follows.

```
int main (void)
{
        init_gpio();
        while(1)
        {
                //Look at status of SW2 on PTE26
                if((GPIOE_PDIR & GPIO_PDIR_PDI(GPIO_PIN(26)))==0)  //If pressed...
                {
                        toggle_LED_allocated_in_Flash();
                        toggle_LED_relocated_in_Flash_address_0x0000FF00();
                        toggle_LED_relocated_in_any_RAM_address();
                        toggle_LED_relocated_in_RAM_address_0x1FFFE000();
                }
                delay();
        }
        return 0;
}
```

Listing 22 – Function main

Go to menu Project > Build Project and then search for the *.map file inside {Project_path}/FLASH. Here you can see that function **toggle_LED_relocated_in_RAM_address_0x1FFFE000** is placed exactly where it is expected and it is 40 bytes long. This is shown on listing 23.

```
.my_ram         0x1fffe000       0x50 load address 0x00001c94
                0x1fffe000                . = ALIGN (0x4)
                0x1fffe000                _mySection = .
 *(.myRAM)
 .myRAM         0x1fffe000       0x40 ./Sources/main.o
                0x1fffe000                toggle_LED_relocated_in_RAM_address_0x1FFFE000
```

Listing 23 – Function toggle_LED_relocated_in_RAM_address_0x1FFFE000 in map file

4.5) Relocating Code in external RAM

When the internal RAM of the Microcontroller is not enough for our application it is necessary the use external memories as part of the solution. The process to relocate code in external memories is almost the same as it is for internal RAM. The main difference is that the external memory needs to communicate with the Microcontroller through an interface controller.

K60 and K70 devices provide Flexbus interface which can be used to communicate with external memories, K70 provides also DDR controller. This chapter explains how to relocate a function using such modules in TWR-K60 or TWR-K70.

First write the function using attribute '*section*'. This time the section is called '*myExtRAM*'.

```
/*Toggle LEDs - This functions is relocated in external RAM*/
__attribute__ ((section(".myExtRAM"))) void toggle_LED_relocated_in_external_RAM()
{
        unsigned int x;
        for(x = 0; x < 6; x++ )
        {
                GPIOA_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(10));
                GPIOA_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(11));
                GPIOA_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(28));
                GPIOA_PTOR|=GPIO_PDOR_PDO(GPIO_PIN(29));
                delay();
        }
}
```

Listing 24 – Function toggle_LED_relocated_in_external_RAM

Then create a memory segment for external memory. For Flexbus you can use address 0x60000000, for DDR controller you can use 0x08000000. The segments will be called '*extmram*' and '*extddr*'.

```
MEMORY
{
  m_interrupts   (rx) : ORIGIN = 0x00000000, LENGTH = 0x1E8
  m_cfmprotrom   (rx) : ORIGIN = 0x00000400, LENGTH = 0x10
  m_text         (rx) : ORIGIN = 0x00000800, LENGTH = 0x1M - 0x800
  m_data         (rwx) : ORIGIN = 0x1FFF0000, LENGTH = 64K
  m_data2        (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
  extmram        (rxw) : ORIGIN = 0x60000000, LENGTH = 0x00080000                /* MRAM Address*/
  extddr         (rxw) : ORIGIN = 0x08000000, LENGTH = 0x00080000                /* DDR Address*/
}
```

Listing 25 – Linker file edited to add external MRAM and external DDR segments

As it was explained in the previous chapter, a new section must be created in the linker file, this section must be contained in one of the segments created for external RAM (MRAM or DDR). The new section created for this purpose is called '*.ext_ram*' and the segments where it may be contained are called '*extmram*' or '*extddr'*.

As it was mentioned before, the function must be saved in flash and copied to RAM in runtime. The instruction '*AT*' indicates the flash address where the function will be resident before being copied.

If you search the for label '*___ROM_AT*' in the linker file you will find that it points to the first available address in flash, here is where section '*.data*' is resident, therefore, the address where section '*.ext_ram*' must reside is after section '*.data*'. To calculate this address the instruction '*SIZEOF*' is used.

The listing below shows a new section contained in segment '*mram*' to use TWR-MEM through Flexbus module.

```
   /* Calculating the address where code to be copied to External RAM starts*/
  ___ExtRAMCodeStart = ___ROM_AT + SIZEOF(.data);

   /* Section created to relocate code in External RAM  */
  .ext_ram :  AT(___ExtRAMCodeStart)
  {
        . = ALIGN(4);
        ___ExtRAMStart = .;
        *(.myExtRAM)
        . = ALIGN(4);

  } > extmram

  /* Getting the size of the code to be copied to External RAM */
  ___ExtRAMCodeSize = SIZEOF(.ext_ram);
```

Listing 26 – Section .ext_ram

Use segment '*extddr*' instead of '*extmram*' to use onboard DDR2 on TWR-K70 through the DDR Controller module.

```
  } > extddr
```

As we inserted a new RAM section that was not considered by the linker we must make a couple of adjustments.

At this moment label `___m_data2_ROMStart` overlaps with the address where section '*.my_ram*' resides. This label must be edited to skip section '*.ext_ram*'.

```
___m_data2_ROMStart = ___ROM_AT + SIZEOF(.data) + SIZEOF(.ext_ram);
```

Label `_romp_at` must also be edited to consider section '*.ext_ram*'.

```
  _romp_at = ___ROM_AT + SIZEOF(.data) + SIZEOF(.user_data2) + SIZEOF(.ext_ram);
```

Next write a call to function **toggle_LED_relocated_in_external_RAM** after SW2 is pressed. Function main should look as shown below.

```
int main (void)
{
        init_gpio();
        while(1)
        {
                //Look at status of SW2 on PTE26
                if((GPIOE_PDIR & GPIO_PDIR_PDI(GPIO_PIN(26)))==0)  //If pressed...
                {
                        toggle_LED_relocated_in_external_RAM();
                }
                delay();
        }
        return 0;
}
```

Listing 27 – Function main

At this point you must be able to build the application. Go to menu Project > Build Project and then search for the *.map file inside {Project_path}/FLASH. Here you can see that function **toggle_LED_relocated_in_ext_RAM** is placed in 0x60000000 if you are using MRAM through Flexbus or 0x08000000 if you are using DDR controller.

```
.ext_mram       0x60000000       0x98 load address 0x0000259c
                0x60000000                . = ALIGN (0x4)
                0x60000000                ___ExtRAMStart = .
 *(.myExtRAM)
 .myExtRAM      0x60000000       0x88 ./Sources/GPIOs.o
                0x60000000                toggle_LED_relocated_in_external_RAM
```

Listing 28 – Function toggle_LED_relocated_in_ext_RAM

The last step is to copy the code from flash to RAM. As we need Flexbus or DDR controller module to communicate with the external RAM we need to initialize them before copying the code to the external memory. There are 2 ways to do this.

- Edit init_kinetis.tcl file to add Flexbus or DDR controller initialization, then edit *.mem file to permit the debugger to write to the external memory address range and finally edit section '*.romp*' as it was made in previous chapter. In this appnote we are not getting deep into this approach.
- Initialize Flexbus or DDR controller in your application and copy the code manually. You can find the initialization code of these modules in Appendix A and Appendix B.

The next listing shows the code you need to copy from flash to RAM. As you may notice in listing 26 there are some labels used to determine the address where the section starts and its size. This labels are used by the function that makes the copy.

```
extern unsigned long ___ExtRAMCodeStart[];
#define ExtRAMCodeStart (unsigned long)___ExtRAMCodeStart

extern unsigned long ___ExtRAMCodeSize[];
#define ExtRAMCodeSize (unsigned long)___ExtRAMCodeSize

extern unsigned long ___ExtRAMStart[];
#define ExtRAMStartAddr (unsigned long)___ExtRAMStart

unsigned char *Source;
unsigned char *Destiny;
unsigned int Size;
```

```
void copyToExtRAM(void)
{
	/* Initialize the pointers to start the copy from Flash to RAM */
	Source = (unsigned char *)(ExtRAMCodeStart);
	Destiny = (unsigned char *)(ExtRAMStartAddr);
	Size = (unsigned long)(ExtRAMCodeSize);

	/* Copying the code from Flash to External RAM */
	while(Size--)
	{
		*Destiny++ = *Source++;
	}
}
```

Listing 29 – Function copyToExtRAM

Finally, listing 30 shows function main initializing both, Flexbus and DDR controller. Then the copy from flash to external RAM is executed.

```
int main (void)
{
	/* Initialize GPIO on TWR-K70F120M */
	init_gpio();

	/* Initialize Flexbus on TWR-K70F120M */
	TWRK70_flexbus_init();

	/* Initialize DDR on TWR-K70F120M */
	disable_wdt();
	init_pll();
	init_ddr();

	/* Copy code to external DDR */
	copyToExtRAM();
	while(1)
	{
		//Look at status of SW2 on PTE26
		if((GPIOE_PDIR & GPIO_PDIR_PDI(GPIO_PIN(26)))==0)  //If pressed...
		{
			toggle_LED_relocated_in_external_RAM();
		}
		delay();
	}
	return 0;
}
```

Listing 30 – Function main

5) Relocating Data

Kinetis K family devices provide 2 RAM blocks, any access below 0x2000_0000 will run on the CODE bus, and most accesses at 0x2000_0000 or above will run on the system bus (PPB accesses being the exception).

The default linker file is configured to place all data in '*m_data'* segment which is connected to the code bus. Unfortunately GNU linker is not smart enough to automatically distribute the RAM objects between the sections, so as user you need to manually distribute things.

Relocate data in RAM is a very similar process to relocate a function. First you need to use attribute '*section*' in the variables you wish to relocate as shown in listing 31.

```
__attribute__ ((section(".myRAM"))) int my_data[10000];

int main(void)
{
        int i = 0;
        int counter = 0;

        for(i = 0; i < 10000; i++) {
                counter++;
                my_data[i] = counter;
        }

        return 0;
}
```

Listing 31 – Relocating data

Then add a new section in the linker file contained in segment '*m_data2'* which is accessed by the system bus.

```
.mySection :
{
         *(.myRAM)
} > m_data2
```

Listing 32 – Creating new linker section to relocate data

As you can see on listing 33, '*my_data[]'* has been relocated to address 0x20000000.

```
.mySection      0x20000000     0x4e20
 *(.myRAM)
 .myRAM         0x20000000     0x4e20 ./Sources/main.o
                0x20000000                my_data
```

Listing 33 – Memory map showing main.c data relocated

This could be complicated if there are a big amount of variables which you want to relocate, in this case you can relocate all the variables of a whole source file using the default section COMMON for that specific file. Please note that in this case attribute '*section*' is not necessary. Listings 34 and 35 show an example.

```
int my_data[5000];
int my_data1[5000];

int main(void)
{
        int i = 0;
        int counter = 0;

        for(i = 0; i < 10000; i++) {
                counter++;
                my_data[i] = counter;
                my_data1[i] = counter + 1;
        }

        return 0;
}
```

Listing 34 – Relocating data

```
.mySection :
  {
         *main.o(COMMON)
   } > m_data2
```

Listing 35 – Relocate data of main.c file

As you can see in listing 36, all data in main.c was relocated to address 0x20000000.

```
.mySection      0x20000000     0x9c40
 *main.o(COMMON)
 COMMON         0x20000000     0x9c40 ./Sources/main.o
                0x20000000                my_data
                0x20004e20                my_data1
```

Listing 35 – Memory map showing main.c data relocated

Note

To an application, there isn't much distinction between the CODE and system buses; however, there is a difference in the performance of the two buses. CODE bus cycles have no delay added at the core. System bus cycle timing depends on the type of access. System bus data accesses have no delay added at the core, but instruction accesses add one wait state at the core.

6) Linker File for RAM Project

The difference between a ROM project and a RAM project is that in the RAM project, the code and data reside in RAM, therefore there is no need to copy from ROM to RAM, so sections '*.romp'* and '*.cfmprotect*' are not required in the linker file. This is used to download all the application and be able to debug out of RAM.

The following listing shows an example of the memory distribution.

```
MEMORY
{
  m_interrupts   (rx) : ORIGIN = 0x1FFF0000, LENGTH = 0x1E8
  m_text         (rx) : ORIGIN = 0x1FFF01E8, LENGTH = 64K-0x1E8          /* Lower SRAM */
  m_data         (rw) : ORIGIN = 0x20000000, LENGTH = 64K                /* Upper SRAM */
}
```

Listing 36 – Memory Segment in a RAM Project

7) Debugging out of External RAM

It is easier to debug an application from RAM as you can skip the flashing process each time you make edits in your application. If internal RAM is not big enough for your application you can configure your project to debug out of external RAM. You can follow the next steps to create a new external RAM configuration and debug session.

1) Click menu Project > Build Configurations > Manage and click 'New' button.

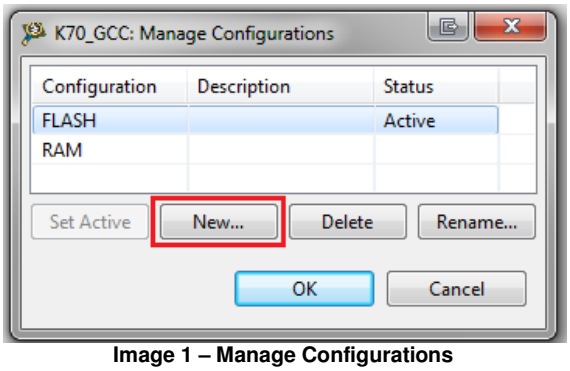

Image 1 – Manage Configurations

2) In the window prompted write a name for your new Build Configuration, e.g. MRAM' or 'DDR'. Then select 'Copy Settings from Existing Configuration', choose 'RAM' and click 'OK' button.

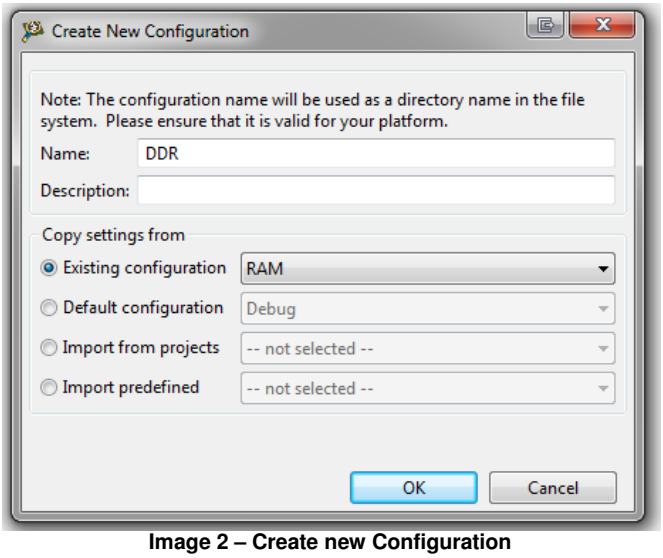

Image 2 – Create new Configuration

3) Set your new configuration as active using ‘Set Active’ button.

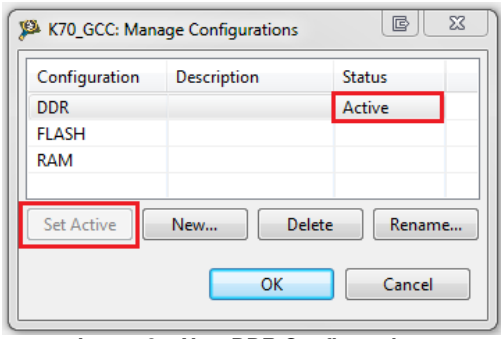

Image 3 – New DDR Configuration

4) Right click on ‘Linker_Files’ folder which is located in {Project_path}/Project_Settings and select New > Other…

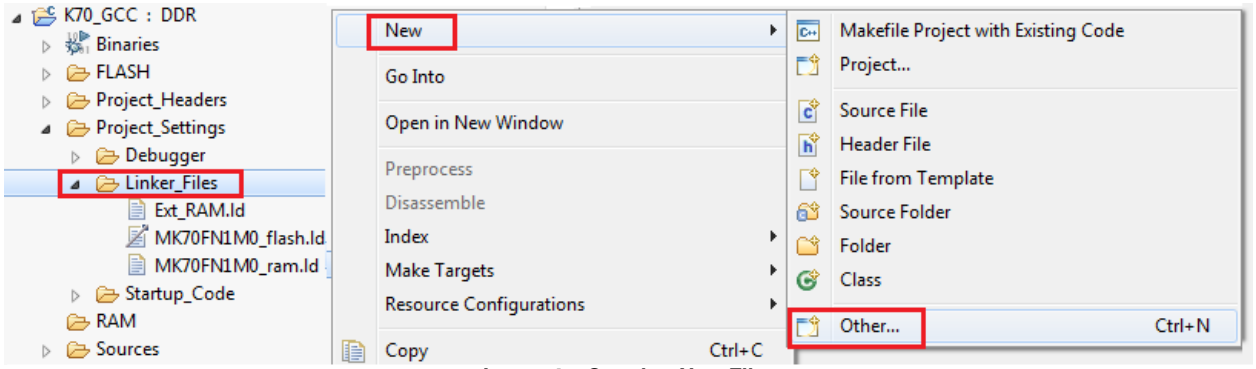

Image 4 – Creating New File

5) In the new window select General > File and click ‘Next’, then write a name for your new *.ld file, e.g. ‘Ext_RAM.ld’, then click ‘Finish’.

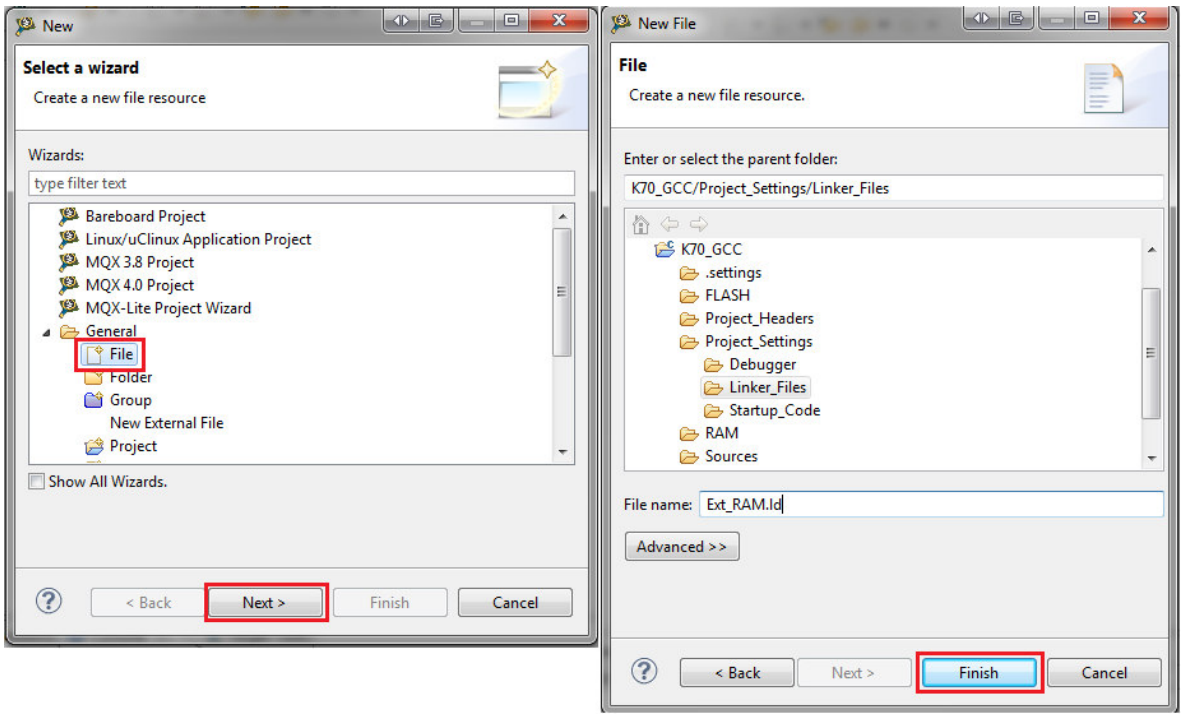

Image 5 – New File Wizard

6) Copy the content of the ram linker file (xxxx_ram.ld) located in {Project_path}/Project_Settings/Linker_Files and paste it into the new .ld file that has been created.

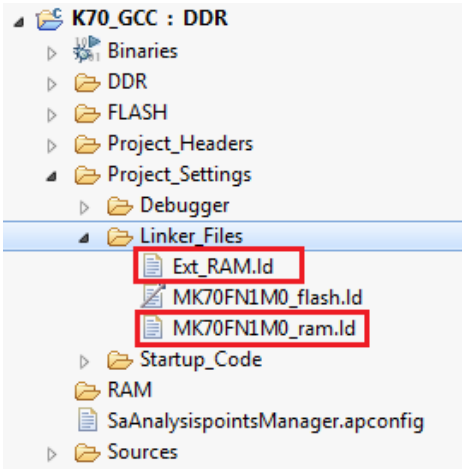

Image 6 – Project Source tree

7) The only difference between a RAM linker file and an external RAM linker file is the memory segment. Listing 37 and 38 show DDR and MRAM examples, as you can see interrupts are kept in internal RAM while the rest of the segments are allocated in external memory addresses.

```
MEMORY
{
  m_interrupts   (rx) : ORIGIN = 0x1FFF0000, LENGTH = 0x1E8              /* Internal SRAM */
  m_text         (rx) : ORIGIN = 0x08000000, LENGTH = 0xE0000            /* DDR2 */
  m_data         (rw) : ORIGIN = 0x080F8000, LENGTH = 0x20000            /* DDR2 */
}
```

Listing 37 – Memory Segment in a K70 External DDR Project

```
MEMORY
{
  m_interrupts   (rx) : ORIGIN = 0x1FFF0000, LENGTH = 0x1E0              /* Internal SRAM */
  m_text         (rx) : ORIGIN = 0x60000000, LENGTH = 0x70000            /* MRAM */
  m_data         (rw) : ORIGIN = 0x60070000, LENGTH = 0x10000            /* MRAM */
}
```

Listing 38 – Memory Segment in a K60 External MRAM Project

8) Go to menu Project > Properties > C/C++ Build > Settings > ARM Ltd Windows GCC C Linker > General > script file (-T) and browse for your new *.ld file, then click 'OK'.

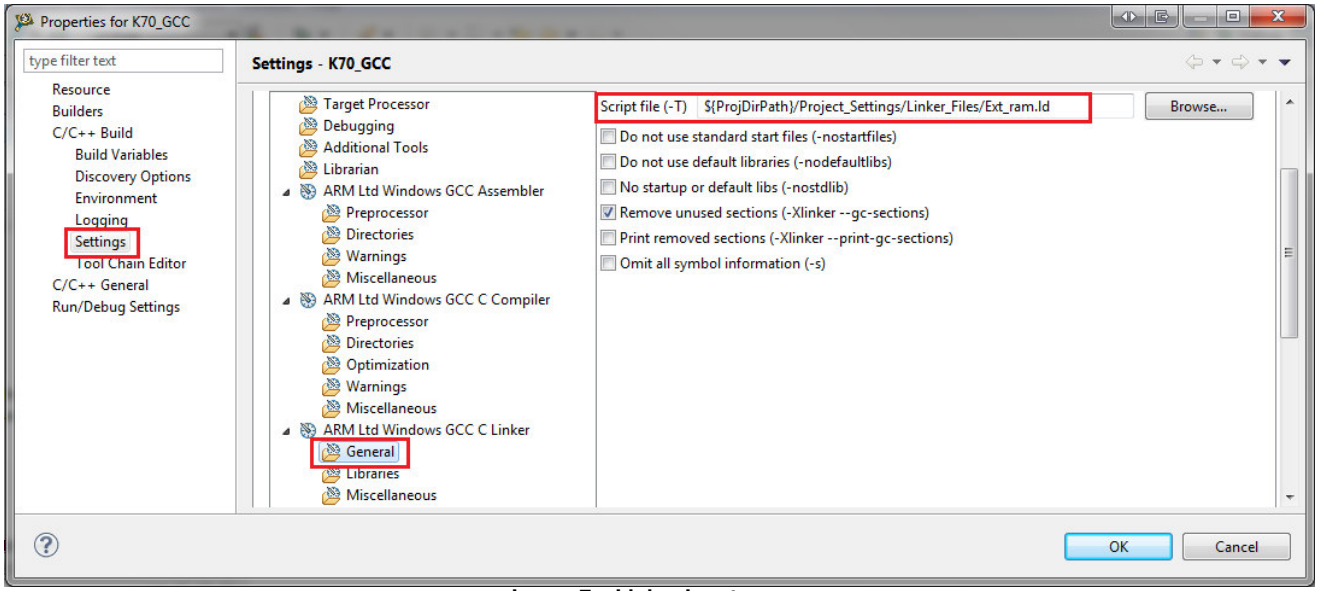

Image 7 – Linker Input

9) Build your application. Notice that a folder with the name of the new Build Configuration is created.

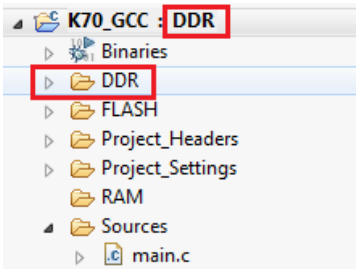

Image 8 – External RAM Build Configuration

10) Inside ‘Debugger’ folder which default location is {Project_path}/Projects_Settings you will find a *.mem file and init_kinetis.tcl.

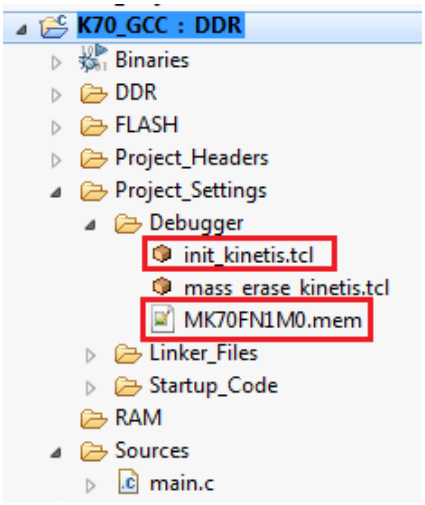

Image 9 – Initialization files

11) You must edit init_kinetis.tcl to allow the Debugger to configure FlexBus module or DDR controller in order to communicate with external MRAM or DDR2 to start a external RAM debug sesion. Then edit .mem file to let the BDM know where to find external RAM addresses.

    a. In **Appendix C** you can find the content of init_kinetis.tcl file to initialize DDR controller in K70.
    b. In **Appendix D** you can find the content of ‘K70FN1M0.mem’ including the memory ranges used by DDR.
    c. In **Appendix E** you can find the content of init_kinetis.tcl file to initialize Flexbus. Don’t forget to call the Flexbus routine from main as highlighted below.
    d. In **Appendix F** you can find the content of MK60N512.mem file which includes MRAM Flexbus address range used by MRAM.

12) So far we have configured the project to create an external RAM executable file. Now you need to create a debug session to be able to debug out of External RAM. Select menu Run > Debug Configurations. Right click on Internal RAM debug configuration and select ‘Duplicate’.

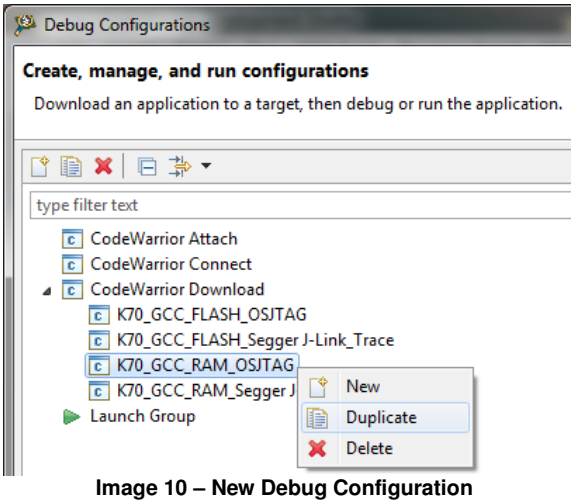

Image 10 – New Debug Configuration

13) A new Debug Configuration will be created. In the Application box select the .elf file created inside the folder with the same name of the Build Configuration you created. In the image below it is DDR/K70_GCC.

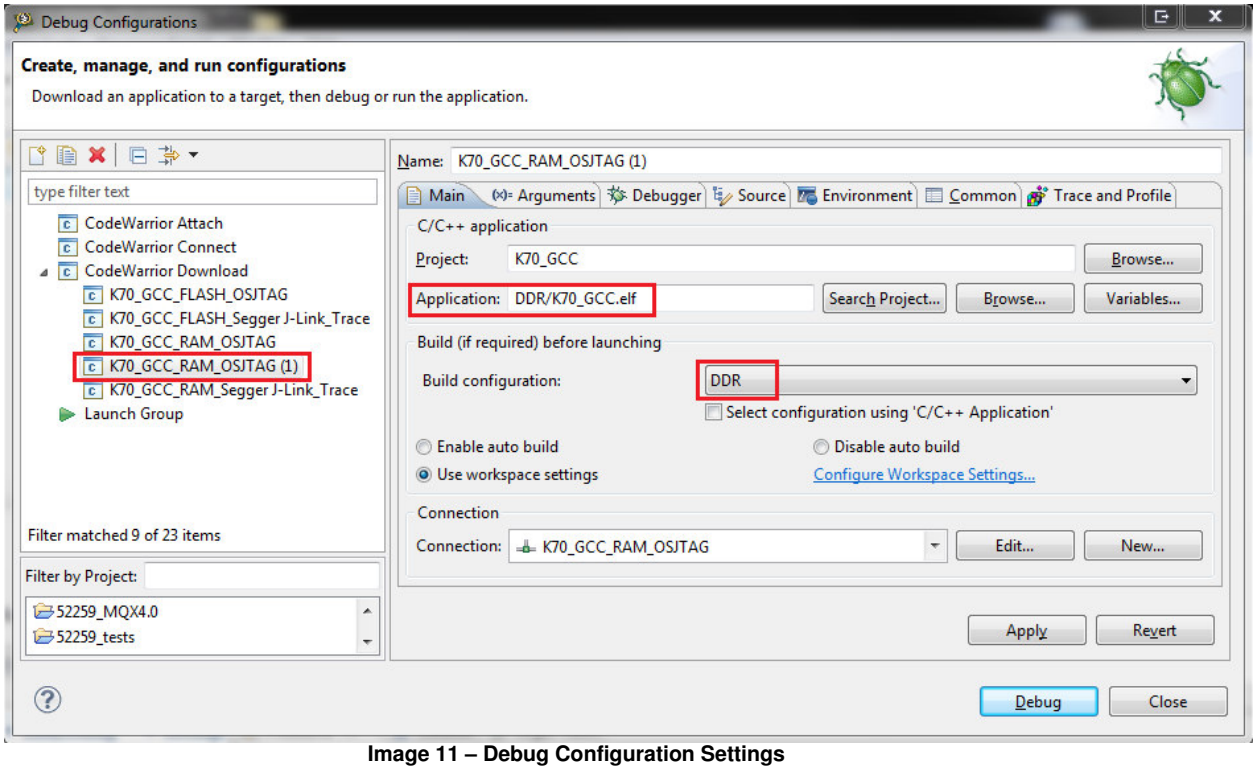

Image 11 – Debug Configuration Settings

14) Finally click ‘Apply’ button and then ‘Debug’ button. Now you are debugging from external RAM.

Appendix A

```
#define MRAM_START_ADDRESS (*(volatile unsigned char*)(0x60000000))

void TWRK70_flexbus_init(void)
{
/* Enable the FlexBus
 * Configure the FlexBus Registers for 8-bit port size with separate address and data using chip select 0
 * These configurations are specific to communicating with the MRAM used in this example
 * For K60 tower module - do not set byte lane shift so that data comes out on AD[31:24]
*/
         //Set Base address
          FB_CSAR0 = (unsigned int)&MRAM_START_ADDRESS;

          FB_CSCR0  =   FB_CSCR_PS(1)      // 8-bit port
                //                | FB_CSCR_BSTR_MASK  // Burst read enable
                                | FB_CSCR_AA_MASK    // auto-acknowledge
                      | FB_CSCR_ASET(0x1)  // assert chip select on second clock edge after address is
asserted
                      | FB_CSCR_WS(0x1)    // 1 wait state
                      ;

          FB_CSMR0  =   FB_CSMR_BAM(0x7)  //Set base address mask for 512K address space
                      | FB_CSMR_V_MASK    //Enable cs signal
                      ;

          //fb clock divider 3
          SIM_CLKDIV1 |= SIM_CLKDIV1_OUTDIV3(0x3);

          /* Configure the pins needed to FlexBus Function (Alt 5) */
          //address
          PORTB_PCR11 = PORT_PCR_MUX(5);           //  fb_ad[18]
          PORTB_PCR16 = PORT_PCR_MUX(5);           //  fb_ad[17]
          PORTB_PCR17 = PORT_PCR_MUX(5);           //  fb_ad[16]
          PORTB_PCR18 = PORT_PCR_MUX(5);           //  fb_ad[15]
          PORTC_PCR0  = PORT_PCR_MUX(5);           //  fb_ad[14]
          PORTC_PCR1  = PORT_PCR_MUX(5);           //  fb_ad[13]
          PORTC_PCR2  = PORT_PCR_MUX(5);           //  fb_ad[12]
          PORTC_PCR4  = PORT_PCR_MUX(5);           //  fb_ad[11]
          PORTC_PCR5  = PORT_PCR_MUX(5);           //  fb_ad[10]
          PORTC_PCR6  = PORT_PCR_MUX(5);           //  fb_ad[9]
          PORTC_PCR7  = PORT_PCR_MUX(5);           //  fb_ad[8]
          PORTC_PCR8  = PORT_PCR_MUX(5);           //  fb_ad[7]
          PORTC_PCR9  = PORT_PCR_MUX(5);           //  fb_ad[6]
          PORTC_PCR10 = PORT_PCR_MUX(5);           //  fb_ad[5]
          PORTD_PCR2  = PORT_PCR_MUX(5);           //  fb_ad[4]
          PORTD_PCR3  = PORT_PCR_MUX(5);           //  fb_ad[3]
          PORTD_PCR4  = PORT_PCR_MUX(5);           //  fb_ad[2]
          PORTD_PCR5  = PORT_PCR_MUX(5);           //  fb_ad[1]
          PORTD_PCR6  = PORT_PCR_MUX(5);           //  fb_ad[0]

          //data
          PORTB_PCR20 = PORT_PCR_MUX(5);           //  fb_ad[31] used as d[7]
          PORTB_PCR21 = PORT_PCR_MUX(5);           //  fb_ad[30] used as d[6]
          PORTB_PCR22 = PORT_PCR_MUX(5);           //  fb_ad[29] used as d[5]
          PORTB_PCR23 = PORT_PCR_MUX(5);           //  fb_ad[28] used as d[4]
          PORTC_PCR12 = PORT_PCR_MUX(5);           //  fb_ad[27] used as d[3]
          PORTC_PCR13 = PORT_PCR_MUX(5);           //  fb_ad[26] used as d[2]
          PORTC_PCR14 = PORT_PCR_MUX(5);           //  fb_ad[25] used as d[1]
          PORTC_PCR15 = PORT_PCR_MUX(5);           //  fb_ad[24] used as d[0]

          //control signals
          PORTB_PCR19 = PORT_PCR_MUX(5);          // fb_oe_b
          PORTC_PCR11 = PORT_PCR_MUX(5);          // fb_rw_b
```

```
          PORTD_PCR1  = PORT_PCR_MUX(5);          // fb_cs0_b
          PORTD_PCR0  = PORT_PCR_MUX(5);          // fb_ale
}
```

Appendix B

```
void disable_wdt(void)
{
        // First unlock the watchdog so that we can write to registers */
    // Write 0xC520 to the unlock register WDOG_UNLOCK*/
    WDOG_UNLOCK = 0xC520;

        // Followed by 0xD928 to complete the unlock */
    WDOG_UNLOCK = 0xD928;

        // Clear the WDOGEN bit to disable the watchdog */
    WDOG_STCTRLH = 0x01D2;
}

void init_pll(void)
{
        // Initialize SIM dividers
        SIM_SCGC5 = 0x00047F82;
        SIM_CLKDIV1 = 0x01250000;

        // Initialize PLL1
        MCG_C2 = 0x10;
        MCG_C1 = 0xA8;
        MCG_C5 = 0x04;
        MCG_C6 = 0x68;
        MCG_C5 = 0x44;
        MCG_C1 = 0x28;

        // Initialize PLL1
        MCG_C10 = 0x14;
        MCG_C12 = 0x0E;
        MCG_C11 = 0x44;

        // Delay to allow the PLL time to lock
        delay_ddr();
}

void init_ddr(void)
{

    //  Enable DDR controller clock
        SIM_SCGC3 = 0x00004000;

        // Enable DDR pads and set slew rate
        SIM_MCR = 0x1C4;

        delay_ddr();

        SIM_MCR = 0x0C4;

        // I/O Pad Control (PAD_CTRL) register.*/
        * (volatile unsigned int *)(0x400ae1ac) = 0x01030203;

        // Initialize the DDR controller
        DDR_CR00 = 0x00000400;
        DDR_CR01 = 0x01000000;
        DDR_CR02 = 0x02000031;
        DDR_CR03 = 0x02020506;
        DDR_CR04 = 0x06090202;
        DDR_CR05 = 0x02020302;
        DDR_CR06 = 0x02904002;
        DDR_CR07 = 0x01000303;
        DDR_CR08 = 0x05030201;
```

```
        DDR_CR09 = 0x020000c8;
        DDR_CR10 = 0x03003207;
        DDR_CR11 = 0x01000000;
        DDR_CR12 = 0x04920031;
        DDR_CR13 = 0x00000005;
        DDR_CR14 = 0x00C80002;
        DDR_CR15 = 0x00000032;
        DDR_CR16 = 0x00000001;
        DDR_CR20 = 0x00030300;
        DDR_CR21 = 0x00040232;
        DDR_CR22 = 0x00000000;
        DDR_CR23 = 0x00040302;
        DDR_CR25 = 0x0A010201;
        DDR_CR26 = 0x0101FFFF;
        DDR_CR27 = 0x01010101;
        DDR_CR28 = 0x00000003;
        DDR_CR29 = 0x00000000;
        DDR_CR30 = 0x00000001;
        DDR_CR34 = 0x02020101;
        DDR_CR36 = 0x01010201;
        DDR_CR37 = 0x00000200;
        DDR_CR38 = 0x00200000;
        DDR_CR39 = 0x01010020;
        DDR_CR40 = 0x00002000;
        DDR_CR41 = 0x01010020;
        DDR_CR42 = 0x00002000;
        DDR_CR43 = 0x01010020;
        DDR_CR44 = 0x00000000;
        DDR_CR45 = 0x03030303;
        DDR_CR46 = 0x02006401;
        DDR_CR47 = 0x01020202;
        DDR_CR48 = 0x01010064;
        DDR_CR49 = 0x00020101;
        DDR_CR50 = 0x00000064;
        DDR_CR52 = 0x02000602;
        DDR_CR53 = 0x03c80000;
        DDR_CR54 = 0x03c803c8;
        DDR_CR55 = 0x03c803c8;
        DDR_CR56 = 0x020303c8;
        DDR_CR57 = 0x01010002;

        // Set the START bit
        DDR_CR00 = 0x00000401;

        // Set the SDRAM size in the MCM
        MCM_CR = 0x00100000;
}

void delay_ddr(void)
{
        unsigned int i, n;
        for(i=0;i<1000;i++)
        {
                for(n=0;n<1000;n++)
                {
                        asm("nop");
                }
        }
}
```

Appendix C

```
# this method initializes debug modules which are not affected by software reset
# register names should be referenced including the register group name to improve performance

proc init_debug_modules {} {
    # clear DWT function registers
        reg "Core Debug Registers/DEMCR" = 0x1000001
        reg "Data Watchpoint and Trace Unit Registers/DWT_FUNCTION0" = 0x0
        reg "Data Watchpoint and Trace Unit Registers/DWT_FUNCTION1" = 0x0
        reg "Data Watchpoint and Trace Unit Registers/DWT_FUNCTION2" = 0x0
        reg "Data Watchpoint and Trace Unit Registers/DWT_FUNCTION3" = 0x0
        # clear FPB comparators
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP0" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP1" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP2" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP3" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP4" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP5" = 0x0
}

proc init_trace_modules {} {
    # clear DWT registers
        reg "Data Watchpoint and Trace Unit Registers/DWT_CTRL" =0x40000000
        reg "Data Watchpoint and Trace Unit Registers/DWT_CYCCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_CPICNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_EXCCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_SLEEPCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_LSUCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_FOLDCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_COMP0" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_COMP1" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_COMP2" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_COMP3" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_MASK0" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_MASK1" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_MASK2" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_MASK3" =0x0
        # clear ITM registers
    reg "Instrumentation Trace Macrocell Registers/ITM_LAR" =0xc5acce55
        reg "Instrumentation Trace Macrocell Registers/ITM_TER" =0x0
        reg "Instrumentation Trace Macrocell Registers/ITM_TPR" =0x0
    reg "Instrumentation Trace Macrocell Registers/ITM_TCR" =0x0
    reg "Instrumentation Trace Macrocell Registers/ITM_LAR" =0x1
    # reset Funnel registers
    reg "Embedded Trace Funnel Registers/ETF_FCR" =0x300
    # clear MCM registers
    reg "Core Platform Miscellaneous Control Module (MCM) Registers/MCM_ETBCC" =0x0
    reg "Core Platform Miscellaneous Control Module (MCM) Registers/MCM_ETBRL" =0x0
    # set SCB_VTOR register for RAM
    reg "System Control Registers/SCB_VTOR" =0x20000000
}

proc envsetup {} {
        # Environment Setup
        radix x
        config hexprefix 0x
        config MemIdentifier p
        config MemWidth 32
        config MemAccess 32
        config MemSwap off
}

proc init_stack_pointer {} {
        reg SP = 0x2000FFF8
}

proc disable_wdt {} {
        # First unlock the watchdog so that we can write to registers */
```

```
    # Write 0xC520 to the unlock register WDOG_UNLOCK*/
    reg "Generation 2008 Watchdog Timer (WDOG)/WDOG_UNLOCK" =0xC520

        # Followed by 0xD928 to complete the unlock */
    reg "Generation 2008 Watchdog Timer (WDOG)/WDOG_UNLOCK" =0xD928

        # Clear the WDOGEN bit to disable the watchdog */
    reg "Generation 2008 Watchdog Timer (WDOG)/WDOG_STCTRLH" =0x01D2
}

proc init_pll {} {
        # Initialize SIM dividers
        reg "System Integration Module (SIM)/SIM_SCGC5" = 0x00047F82
        reg "System Integration Module (SIM)/SIM_CLKDIV1" = 0x01250000

        # Initialize PLL1
        reg "Multipurpose Clock Generator module (MCG)/MCG_C2" = 0x10
        reg "Multipurpose Clock Generator module (MCG)/MCG_C1" = 0xA8
        reg "Multipurpose Clock Generator module (MCG)/MCG_C5" = 0x04
        reg "Multipurpose Clock Generator module (MCG)/MCG_C6" = 0x68
        reg "Multipurpose Clock Generator module (MCG)/MCG_C5" = 0x44
        reg "Multipurpose Clock Generator module (MCG)/MCG_C1" = 0x28

        # Initialize PLL1
        reg "Multipurpose Clock Generator module (MCG)/MCG_C10" = 0x14
        reg "Multipurpose Clock Generator module (MCG)/MCG_C12" = 0x0E
        reg "Multipurpose Clock Generator module (MCG)/MCG_C11" = 0x44

        # Delay to allow the PLL time to lock
        wait (100)
}

proc init_ddr {} {

    #  Enable DDR controller clock
        reg "System Integration Module (SIM)/SIM_SCGC3" = 0x00004000

        # Enable DDR pads and set slew rate
        reg "System Integration Module (SIM)/SIM_MCR" = 0x1C4

        wait (10)

        reg "System Integration Module (SIM)/SIM_MCR" = 0x0C4

        # I/O Pad Control (PAD_CTRL) register.*/
        mem 0x400Ae1ac = 0x01030203

        # Initialize the DDR controller
        reg "DRAM Controller (DDR)/DDR_CR00" = 0x00000400
        reg "DRAM Controller (DDR)/DDR_CR01" = 0x01000000
        reg "DRAM Controller (DDR)/DDR_CR02" = 0x02000031
        reg "DRAM Controller (DDR)/DDR_CR03" = 0x02020506
        reg "DRAM Controller (DDR)/DDR_CR04" = 0x06090202
        reg "DRAM Controller (DDR)/DDR_CR05" = 0x02020302
        reg "DRAM Controller (DDR)/DDR_CR06" = 0x02904002
        reg "DRAM Controller (DDR)/DDR_CR07" = 0x01000303
        reg "DRAM Controller (DDR)/DDR_CR08" = 0x05030201
        reg "DRAM Controller (DDR)/DDR_CR09" = 0x020000c8
        reg "DRAM Controller (DDR)/DDR_CR10" = 0x03003207
        reg "DRAM Controller (DDR)/DDR_CR11" = 0x01000000
        reg "DRAM Controller (DDR)/DDR_CR12" = 0x04920031
        reg "DRAM Controller (DDR)/DDR_CR13" = 0x00000005
        reg "DRAM Controller (DDR)/DDR_CR14" = 0x00C80002
        reg "DRAM Controller (DDR)/DDR_CR15" = 0x00000032
        reg "DRAM Controller (DDR)/DDR_CR16" = 0x00000001
        reg "DRAM Controller (DDR)/DDR_CR20" = 0x00030300
        reg "DRAM Controller (DDR)/DDR_CR21" = 0x00040232
        reg "DRAM Controller (DDR)/DDR_CR22" = 0x00000000
        reg "DRAM Controller (DDR)/DDR_CR23" = 0x00040302
```

```
        reg "DRAM Controller (DDR)/DDR_CR25" = 0x0A010201
        reg "DRAM Controller (DDR)/DDR_CR26" = 0x0101FFFF
        reg "DRAM Controller (DDR)/DDR_CR27" = 0x01010101
        reg "DRAM Controller (DDR)/DDR_CR28" = 0x00000003
        reg "DRAM Controller (DDR)/DDR_CR29" = 0x00000000
        reg "DRAM Controller (DDR)/DDR_CR30" = 0x00000001
        reg "DRAM Controller (DDR)/DDR_CR34" = 0x02020101
        reg "DRAM Controller (DDR)/DDR_CR36" = 0x01010201
        reg "DRAM Controller (DDR)/DDR_CR37" = 0x00000200
        reg "DRAM Controller (DDR)/DDR_CR38" = 0x00200000
        reg "DRAM Controller (DDR)/DDR_CR39" = 0x01010020
        reg "DRAM Controller (DDR)/DDR_CR40" = 0x00002000
        reg "DRAM Controller (DDR)/DDR_CR41" = 0x01010020
        reg "DRAM Controller (DDR)/DDR_CR42" = 0x00002000
        reg "DRAM Controller (DDR)/DDR_CR43" = 0x01010020
        reg "DRAM Controller (DDR)/DDR_CR44" = 0x00000000
        reg "DRAM Controller (DDR)/DDR_CR45" = 0x03030303
        reg "DRAM Controller (DDR)/DDR_CR46" = 0x02006401
        reg "DRAM Controller (DDR)/DDR_CR47" = 0x01020202
        reg "DRAM Controller (DDR)/DDR_CR48" = 0x01010064
        reg "DRAM Controller (DDR)/DDR_CR49" = 0x00020101
        reg "DRAM Controller (DDR)/DDR_CR50" = 0x00000064
        reg "DRAM Controller (DDR)/DDR_CR52" = 0x02000602
        reg "DRAM Controller (DDR)/DDR_CR53" = 0x03c80000
        reg "DRAM Controller (DDR)/DDR_CR54" = 0x03c803c8
        reg "DRAM Controller (DDR)/DDR_CR55" = 0x03c803c8
        reg "DRAM Controller (DDR)/DDR_CR56" = 0x020303c8
        reg "DRAM Controller (DDR)/DDR_CR57" = 0x01010002

        # Set the START bit
        reg "DRAM Controller (DDR)/DDR_CR00" = 0x00000401

        # Set the SDRAM size in the MCM
        reg "Core Platform Miscellaneous Control Module (MCM) Registers/MCM_CR" = 0x00100000
}
#-------------------------------------------------------------------------------
# Main
#-------------------------------------------------------------------------------

  envsetup
  init_debug_modules
  init_trace_modules
  init_stack_pointer
  disable_wdt
  init_pll
  init_ddr
```

Appendix D

```
// Board:
//  Kinetis K70FN1M0

// All reserved ranges read back 0xBABA...
reservedchar 0xBA

usederivative "MK70F15"

//         Memory Map:
//         ---------------------------------------------------------------------

range      0x00000000 0x000FFFFF 4 ReadWrite    // 1024KB Code Flash
reserved   0x00100000 0x07FFFFFF
range      0x08000000 0x0FFFFFFF 4 ReadWrite     // DDR aliased area for core access
reserved   0x10000000 0x13FFFFFF
range      0x14000000 0x14003FFF 4 ReadWrite    // 16KB Programming accelleration RAM
reserved   0x14004000 0x1FFEFFFF
range      0x1FFF0000 0x1FFFFFFF 4 ReadWrite    // 64KB On chip SRAM (TCML)
range      0x20000000 0x2000FFFF 4 ReadWrite    // 64KB On chip SRAM (TCMU)
reserved   0x20010000 0x21FFFFFF
range      0x22000000 0x221FFFFF 4 ReadWrite    // Aliased to TCMU SRAM bitband
reserved   0x22200000 0x3FFFFFFF
//range    0x40000000 0x400FFFFF 4 ReadWrite     // Bitnad regions
reserved   0x40100000 0x41FFFFFF
range      0x42000000 0x43FFFFFF 4 ReadWrite     // AIPS and GPIO bitband
reserved   0x44000000 0x5FFFFFFF
range      0x60000000 0x6FFFFFFF 4 ReadWrite     // Flexbus for external memory
range      0x70000000 0x7FFFFFFF 4 ReadWrite     // DDR Write-back region
range      0x80000000 0x8FFFFFFF 4 ReadWrite     // DDR Write-through region
range      0x90000000 0x9FFFFFFF 4 ReadWrite     // FlexBus write-through region
range      0xA0000000 0xDFFFFFFF 4 ReadWrite     // FlexBus peripheral (not executable)
//range      0xE0000000 0xE00FFFFF 4 ReadWrite   // Private peripherals
reserved   0xE0100000 0xFFFFFFFF
```

Appendix E

```
# this method initializes debug modules which are not affected by software reset
# register names should be referenced including the register group name to improve performance

proc init_debug_modules {} {
    # clear DWT function registers
        reg "Core Debug Registers/DEMCR" = 0x1000001
        reg "Data Watchpoint and Trace Unit Registers/DWT_FUNCTION0" = 0x0
        reg "Data Watchpoint and Trace Unit Registers/DWT_FUNCTION1" = 0x0
        reg "Data Watchpoint and Trace Unit Registers/DWT_FUNCTION2" = 0x0
        reg "Data Watchpoint and Trace Unit Registers/DWT_FUNCTION3" = 0x0
        # clear FPB comparators
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP0" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP1" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP2" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP3" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP4" = 0x0
        reg "Flash Patch and Breakpoint Unit Registers/FP_COMP5" = 0x0
}

proc init_trace_modules {} {
    # clear DWT registers
        reg "Data Watchpoint and Trace Unit Registers/DWT_CTRL" =0x40000000
        reg "Data Watchpoint and Trace Unit Registers/DWT_CYCCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_CPICNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_EXCCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_SLEEPCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_LSUCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_FOLDCNT" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_COMP0" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_COMP1" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_COMP2" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_COMP3" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_MASK0" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_MASK1" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_MASK2" =0x0
    reg "Data Watchpoint and Trace Unit Registers/DWT_MASK3" =0x0
        # clear ITM registers
    reg "Instrumentation Trace Macrocell Registers/ITM_LAR" =0xc5acce55
        reg "Instrumentation Trace Macrocell Registers/ITM_TER" =0x0
        reg "Instrumentation Trace Macrocell Registers/ITM_TPR" =0x0
    reg "Instrumentation Trace Macrocell Registers/ITM_TCR" =0x0
    reg "Instrumentation Trace Macrocell Registers/ITM_LAR" =0x1
    # reset Funnel registers
    reg "Embedded Trace Funnel Registers/ETF_FCR" =0x300
    # clear MCM registers
    reg "Core Platform Miscellaneous Control Module (MCM) Registers/MCM_ETBCC" =0x0
    reg "Core Platform Miscellaneous Control Module (MCM) Registers/MCM_ETBRL" =0x0
    # set SCB_VTOR register for RAM
    reg "System Control Registers/SCB_VTOR" =0x20000000
}

proc flexbus {} {
          reg FB_CSAR0=0x60000000
          reg FB_CSCR0=0x100540
          reg FB_CSMR0=0x70001
          reg SIM_CLKDIV1=0x310000
          reg SIM_SCGC5=0x43f80
          reg PORTB_PCR11 = 0x500
          reg PORTB_PCR16 = 0x500
          reg PORTB_PCR17 = 0x500
          reg PORTB_PCR18 = 0x500
          reg PORTC_PCR0  = 0x500
          reg PORTC_PCR1  = 0x500
          reg PORTC_PCR2  = 0x500
          reg PORTC_PCR4  = 0x500
```

```
          reg PORTC_PCR5  = 0x500
          reg PORTC_PCR6  = 0x500
          reg PORTC_PCR7  = 0x500
          reg PORTC_PCR8  = 0x500
          reg PORTC_PCR9  = 0x500
          reg PORTC_PCR10 = 0x500
          reg PORTD_PCR2  = 0x500
          reg PORTD_PCR3  = 0x500
          reg PORTD_PCR4  = 0x500
          reg PORTD_PCR5  = 0x500
          reg PORTD_PCR6  = 0x500


          reg PORTB_PCR20 = 0x500
          reg PORTB_PCR21 = 0x500
          reg PORTB_PCR22 = 0x500
          reg PORTB_PCR23 = 0x500
          reg PORTC_PCR12 = 0x500
          reg PORTC_PCR13 = 0x500
          reg PORTC_PCR14 = 0x500
          reg PORTC_PCR15 = 0x500


          reg PORTB_PCR19 = 0x500
          reg PORTC_PCR11 = 0x500
          reg PORTD_PCR1  = 0x500
          reg PORTD_PCR0  = 0x500

          #reg WDOG_UNLOCK = 0xC520
          #reg WDOG_UNLOCK = 0xD928
          #reg WDOG_STCTRLH = 0xD2

}


proc envsetup {} {
        # Environment Setup
        radix x
        config hexprefix 0x
        config MemIdentifier p
        config MemWidth 32
        config MemAccess 32
        config MemSwap off
}

#-------------------------------------------------------------------------------
# Main
#-------------------------------------------------------------------------------

  envsetup
  init_debug_modules
  init_trace_modules
  flexbus
```

Appendix F

```
// Board:
//  Kinetis MK60N512VMD100

// All reserved ranges read back 0xBABA...
reservedchar 0xBA

usederivative "MK60N512VMD100"

//         Memory Map:
//         ---------------------------------------------------------------------

range      0x00000000 0x0007FFFF 4 ReadWrite    // 512KB Code Flash
reserved   0x00080000 0x13FFFFFF
range      0x14000000 0x14000FFF 4 ReadWrite    // 4KB Programming accelleration RAM
reserved   0x14001000 0x1FFEFFFF
range      0x1FFF0000 0x1FFFFFFF 4 ReadWrite    // 64KB On chip SRAM (TCML)
range      0x20000000 0x2000FFFF 4 ReadWrite    // 64KB On chip SRAM (TCMU)
reserved   0x20010000 0x21FFFFFF
range      0x22000000 0x221FFFFF 4 ReadWrite    // Aliased to TCMU SRAM bitband
reserved   0x22200000 0x3FFFFFFF
range      0x60000000 0xDFFFFFFF 4 ReadWrite     // Flexbus for external memory
reserved   0xE0100000 0xFFFFFFFF
```