XMC4000

Microcontroller Series
for Industrial Applications

C-Start and Device Initialization

- ✓ Introduction
- ✓ C-Start tasks
- ✓ Linker scripts
- ✓ Execution profiles
- ✓ Device initialization hints

Device Guide

V1.0 2013-06

Microcontrollers

Edition 2013-06

Published by
Infineon Technologies AG
81726 Munich, Germany

Legal Disclaimer

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation, warranties of non-infringement of intellectual property rights of any third party.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Revision History

Page or Item	Subjects (major changes since previous revision)
V1.0, 2013-06	

Trademarks of Infineon Technologies AG

Other Trademarks

Table of Contents

Introduction

1 Introduction

The purpose of this user guide is to provide a broad overview of device initialization. This guide elaborates upon the various stages of initialization which includes boot-up from a state of reset, C-Start and application initialization.

1.1 C-Start (also known as CStart/Startup)

C-Start is essentially a set of activities that must be performed before giving control to the user application ‘entry point’. A good example of an entry point is the “main” function. Applications containing operating systems may potentially have an alternative entry point.

1.2 C-Start Packaging

In a few configurations, C-Start functionality is a part of the user application image. In others, C-Start and user applications are distinct images such as the U-Boot bootloader-for example. Most embedded systems however have the C-Start functionality combined with the final application.

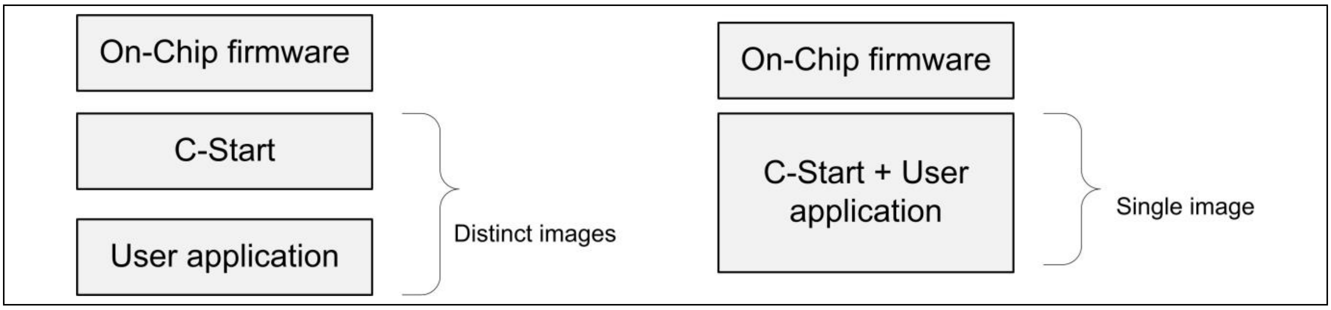

Figure 1 C-Start packaging

1.3 C-Start tasks

There are two fundamental tasks C-Start is expected to perform. They are:

- Device initialization and any errata workaround implementation
- Program loading

These tasks are elaborated in subsequent chapters.

C-Start tasks

2 C-Start tasks

This chapter describes the various tasks of C-Start. A Cortex-M4 CPU based XMC4 device is used in the illustrations that follow. Code fragments have been taken from the following files available with the DAVE distribution:

- startup_XMC4500.s
- System_XMC4500.c

2.1 Device Booting

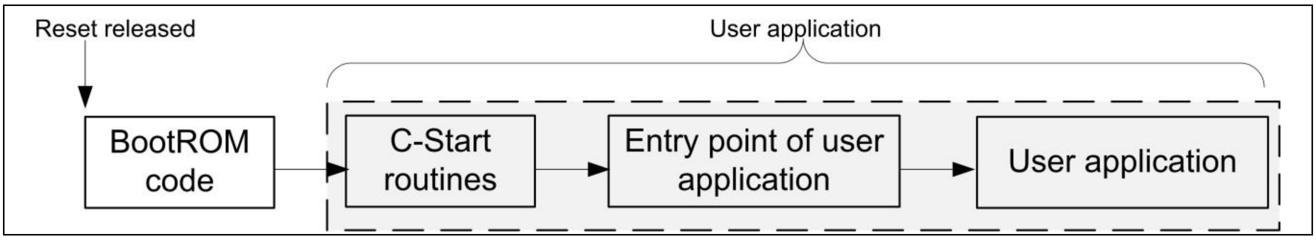

Figure 2 Booting stages

Figure-1 indicates that after the reset (Either Power-On-Reset or System reset) has been released, the CPU starts executing the on-chip firmware stored in the BootROM area of memory. The role of the on-chip firmware is to evaluate the requested chip bootmode and take any necessary actions. As an example, if the chosen bootmode is ASC BSL mode, the on-chip firmware prepares to download the user application by first configuring the UART peripheral for IO exchange and subsequently uploads the user application into PSRAM. The application is received over UART IO lines.

Normal Boot Mode is a commonly deployed boot-mode. On execution, the on-chip firmware reads the user application vector table, typically placed at the start of the Flash area. It extracts the start address of the C-Start routines and then cedes control to the reset handler routine.

The vector table on Cortex-M devices is basically a table of function pointers, used to handle CPU exceptions and device interrupts. The second entry of this table contains the start address of the reset handler.

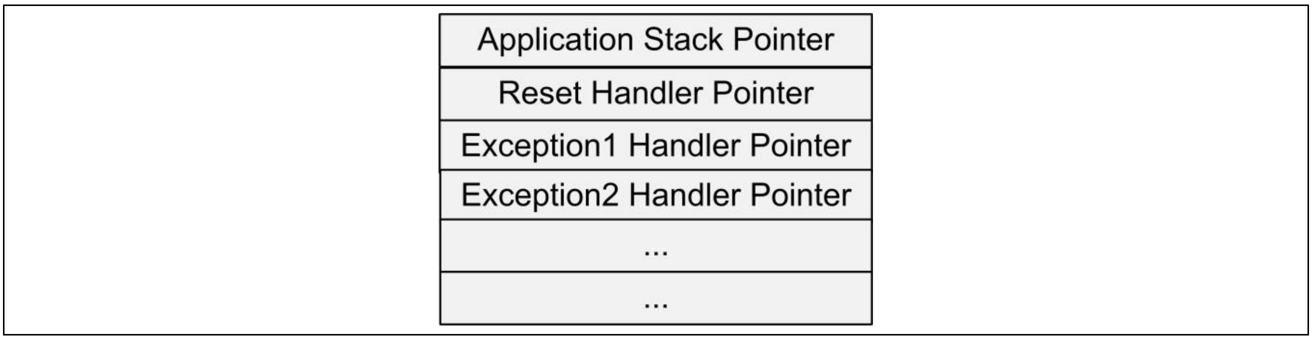

Figure 3 Cortex-M vector table

The following code fragment is an extract of the vector table written for the XMC4500 device for the GNU tool chain.

The Function pointer __Xmc4500_reset_cortex_m is the reset handler and is responsible for device initialization.

```
.syntax unified

.section ".Xmc4500.reset"
.globl   __Xmc4500_interrupt_vector_cortex_m
.type    __Xmc4500_interrupt_vector_cortex_m, %object

__Xmc4500_interrupt_vector_cortex_m:
.long    __Xmc4500_stack             /* Top of Stack                 */
.long    __Xmc4500_reset_cortex_m    /* Reset Handler                */

Entry    NMI_Handler                 /* NMI Handler                  */
Entry    HardFault_Handler           /* Hard Fault Handler           */
Entry    MemManage_Handler           /* MPU Fault Handler            */
Entry    BusFault_Handler            /* Bus Fault Handler            */
Entry    UsageFault_Handler          /* Usage Fault Handler          */
```

Note: The on-chip firmware gives control to this reset handler.

Attention: ***With the exception of the reset handler, all function pointer entries in the vector table have weak implementations in C-Start. The weakly defined CPU exception and device interrupt handler routines provide a default implementation. When users provide an alternative final implementation, these weak definitions are automatically overridden.***

2.2 Device initialization

A typical reset handler written for an XMC4 device (specifically, the XMC4500 device) is shown next. Major functionality is highlighted in **bold**.

```
    .thumb_func
    .globl   __Xmc4500_reset_cortex_m
    .type    __Xmc4500_reset_cortex_m, %function
__Xmc4500_reset_cortex_m:
    .fnstart

    /* Insert workarounds here for silicon bugs as necessary */

    /* C routines are likely to be called. Setup the stack now */
    /* This is already setup by BootROM,hence this step is optional */
    LDR SP,=__Xmc4500_stack

  /* Clock tree, External memory setup etc may be done here */
    LDR      R0, =SystemInit
    BLX      R0

/*
   SystemInit_DAVE3() is provided by DAVE3 code generation engine. It is
   weakly defined here though for a potential override.
*/
    LDR      R0, =SystemInit_DAVE3
    BLX      R0

/* Perform program loading */
  B        __Xmc4500_Program_Loader

    .pool
    .cantunwind
    .fnend
    .size    __Xmc4500_reset_cortex_m,.-__Xmc4500_reset_cortex_m
```

It can be seen from this code fragment that the reset handler invokes a function called SystemInit() which is responsible for clock tree initialization.

Note: *Any user application claiming to conform to the CMSIS standard must invoke the CMSIS routine SystemInit(), and optionally its extension SystemInit_DAVE3() to perform device initialization.*

Users are allowed to insert custom device initialization code in the listing above.

Attention: ***Device initialization related code must abstain from accessing global variables because at this stage program loading is still pending.***

2.3 Program loading

Once the device initialization code and workarounds for silicon errata are executed, control is given to the next stage, known as 'Program Loading'.

The job of a program loader (also known just as a 'loader') is to prepare an environment suitable for user program execution.

A C program typically has the following sections:

- TEXT
- RO-DATA
- DATA
- BSS
- STACK
- HEAP
- USER DEFINED

The job of the program loader is to typically copy TEXT, RO-DATA and DATA from their load address (Load Memory Area LMA) to run address (Virtual Memory Area VMA).

VMA is the address the various sections of the program are linked to. LMA is the address that they are stored at. The concepts of LMA and VMA are elaborated in a subsequent chapter.

The start of LMA/VMA and length of a section to be relocated is obtained from the linker script file. The following is a code snippet of the program loader.

- The program loader is for an application which must eXecute In Place (XIP). DATA is copied from its LMA in Flash to the VMA in SRAM.
- BSS, which has both LMA and VMA in SRAM, is cleared.
- The resulting effect is that the global data variables are already in an initialized state when control is eventually ceded to the user application.

```
.section .Xmc4500.postreset,"x",%progbits
__Xmc4500_Program_Loader:
.fnstart
/* Memories are accessible now*/

/* DATA COPY */
/* R0 = Start address, R1 = Destination address, R2 = Size */
LDR R0, =eROData   /* Obtained from linker script */
LDR R1, =__Xmc4500_sData /* Obtained from linker script */
LDR R2, =__Xmc4500_Data_Size /* Obtained from linker script */

/* Is there anything to be copied? */
CMP R2,#0
BEQ SKIPCOPY

/* For bytecount less than 4, at least 1 word must be copied */
CMP R2,#4
BCS STARTCOPY

/* Byte count < 4 ; so bump it up */
MOV R2,#4
```

```
STARTCOPY:
   /*
      R2 contains byte count. Change it to word count. It is ensured in the
      linker script that the length is always word aligned.
   */
   LSR R2,R2,#2 /* Divide by 4 to obtain word count */

   /* The proverbial loop from the schooldays */
COPYLOOP:
   LDR R3,[R0]
   STR R3,[R1]
   SUBS R2,#1
   BEQ SKIPCOPY
   ADD R0,#4
   ADD R1,#4
   B COPYLOOP

SKIPCOPY:
   /* BSS CLEAR */
   LDR R0, =__Xmc4500_sBSS     /* Start of BSS obtained from linker script*/
   LDR R1, =__Xmc4500_BSS_Size /* BSS size in bytes obtained from linker script */

   /* Find out if there are items assigned to BSS */
   CMP R1,#0
   BEQ SKIPCLEAR

   /* At least 1 word must be copied */
   CMP R1,#4
   BCS STARTCLEAR

   /* Byte count < 4 ; so bump it up to a word*/
   MOV R1,#4

STARTCLEAR:
   LSR R1,R1,#2            /* BSS size in words */

   MOV R2,#0
CLEARLOOP:
   STR R2,[R0]
   SUBS R1,#1
   BEQ SKIPCLEAR
   ADD R0,#4
   B CLEARLOOP

SKIPCLEAR:
   /* Remap vector table */
   /* This is already setup by BootROM,hence this step is optional */
   LDR R0, =__Xmc4500_interrupt_vector_cortex_m
   LDR R1, =SCB_VTOR
   STR R0,[R1]
```

2.4 Giving control to the user application entry point

At this stage

- The device initialization is complete with workarounds, and errata optionally executed
- Application data has been relocated from LMA to VMA

The execution environment has been correctly setup and it is time to cede control to the user application's entry point. While this is typically the main() function for bare-metal programs, alternative entry points are possible.

```
/* Update System Clock */
LDR R0,=SystemCoreClockUpdate
BLX R0

/* C++ : Call global constructors */
LDR R0,=__libc_init_array
BLX R0

/* Reset stack pointer before zipping off to user application, Optional */
LDR SP,=__Xmc4500_stack
MOV R0,#0
MOV R1,#0

/* CEDE CONTROL TO ENTRY POINT OF USER APPLICATION */
LDR PC, =main
```

The Stack pointer is programmed with the value from the top of the stack, and the program counter is adjusted to enable the jump to main() routine.

Attention: ***An alternative application entry point can be programmed into the Program Counter register.***

2.5 Default definition of exception and interrupt handlers

C-Start provides a default definition for each of the CPU exceptions and device interrupt handlers.

The default handlers do no more than implement a self-looping program.

```
.weak NMI_Handler
.type NMI_Handler, %function
NMI_Handler:
B .
```

The code listed above is an example of a weakly defined NMI handler routine. Users may in their application provide an alternative implementation of the NMI Handler. When an alternative implementation is provided, the linker ignores the weak definition and instead considers the object file of the alternative definition for linking purposes.

```
Example: In one of user's C files:
void NMI_Handler(void){}
```

Linker scripts

3 Linker scripts

3.1 Role of a linker script

This chapter elaborates upon the linker script implementation intended for Infineon's XMC devices using the GNU toolchain. Excerpts from linker scripts to be found in the free DAVE tool from Infineon are used in illustrations that follow.

A linker script defines rules and constraints for the linker. The role of the linker is to assign Load and Run addresses to application code and data.

Memories on a typical XMC device

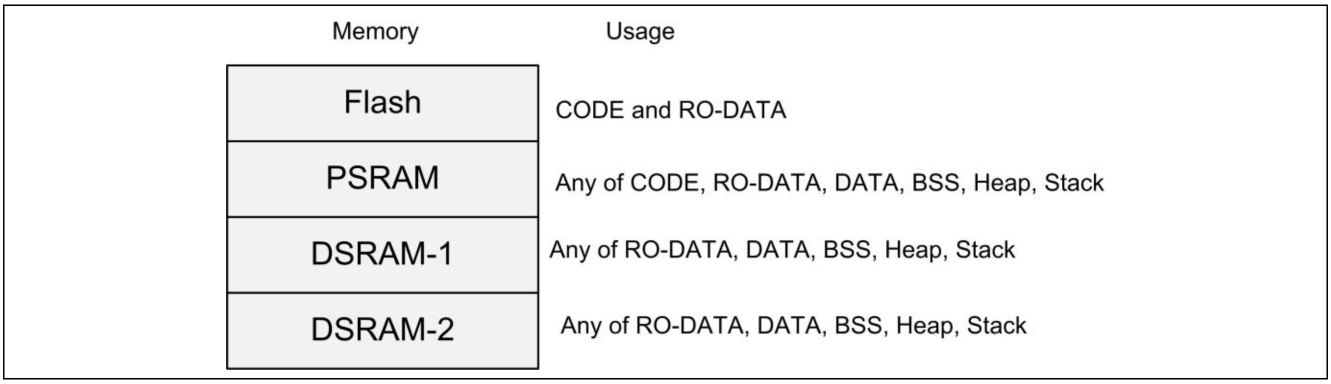

Figure 4 XMC memories and usage

3.2 Concept of LMA and VMA

LMA (Load Memory Address) is the address where the program is physically stored.

VMA (Virtual Memory Address) also known as the Run address is the address where the program is executed from.

Some examples are:

- Program can be stored on flash and executed from SRAM
- Program can be stored and executed from flash
- Some parts of the program can be executed from flash while the rest from SRAM

This concept is pictorially represented here:

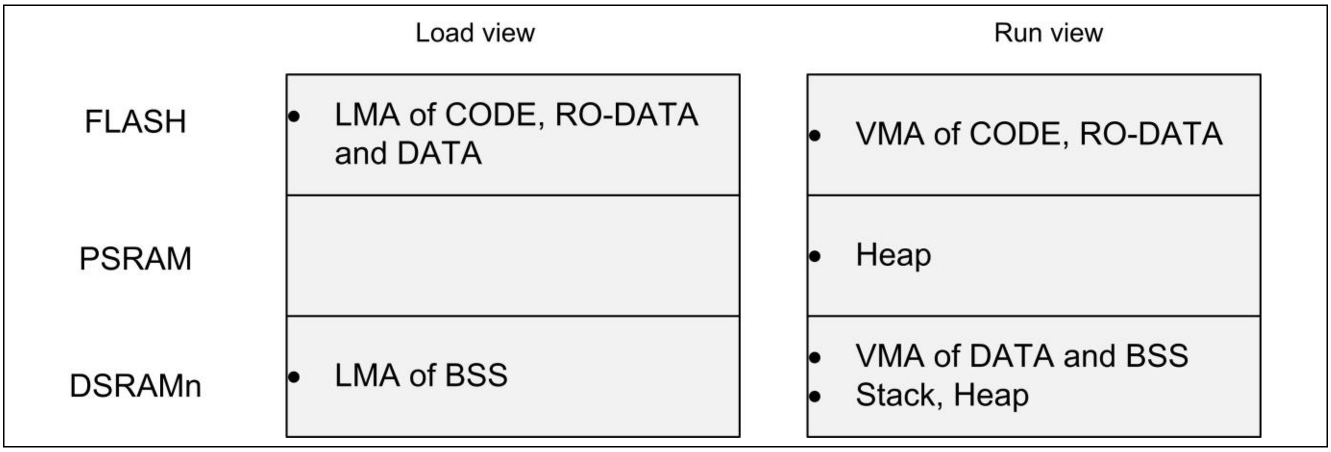

Figure 5 LMA and VMA concept

3.3 Typical program section assignment

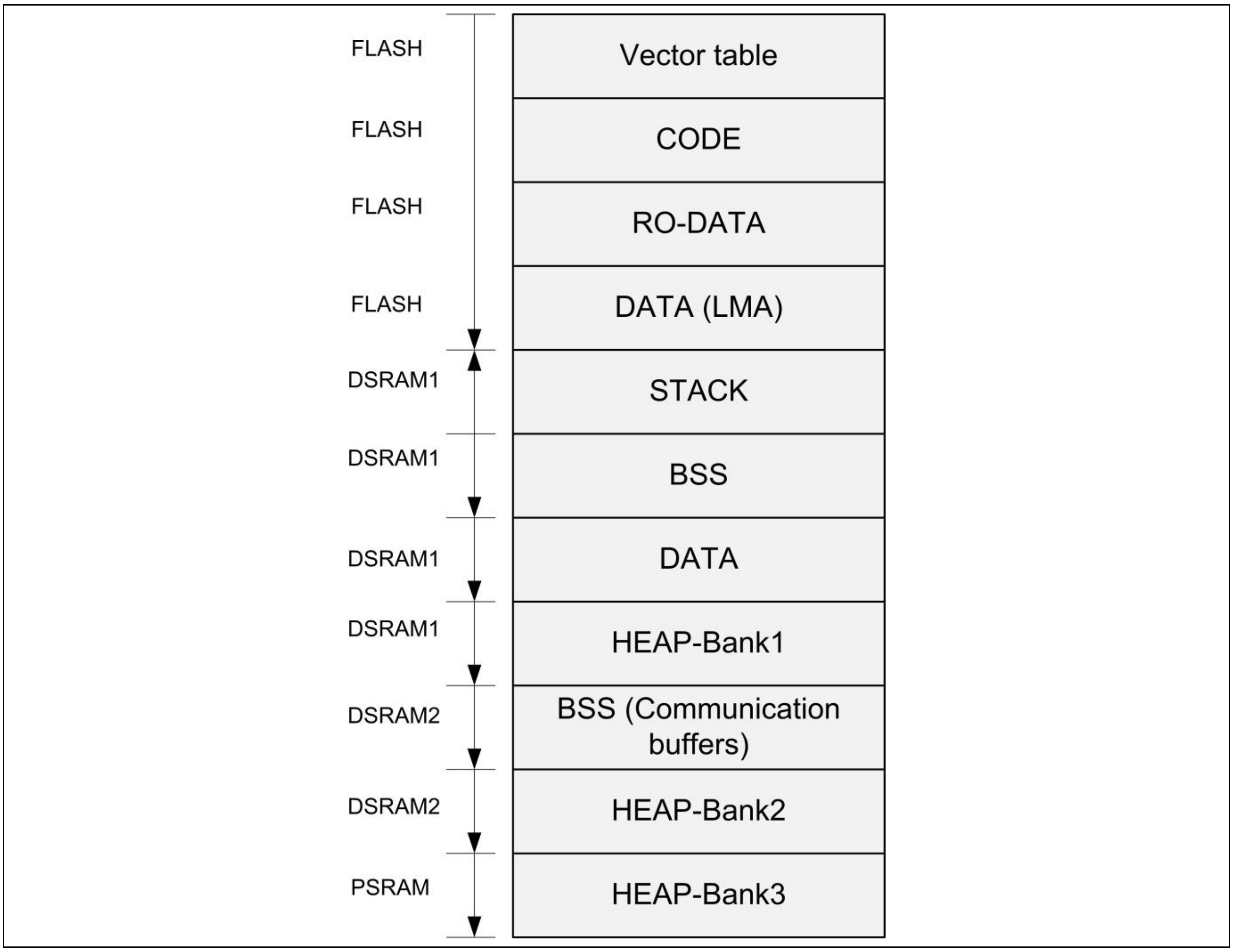

Figure 6 Typical program section placement on a XMC4500 device

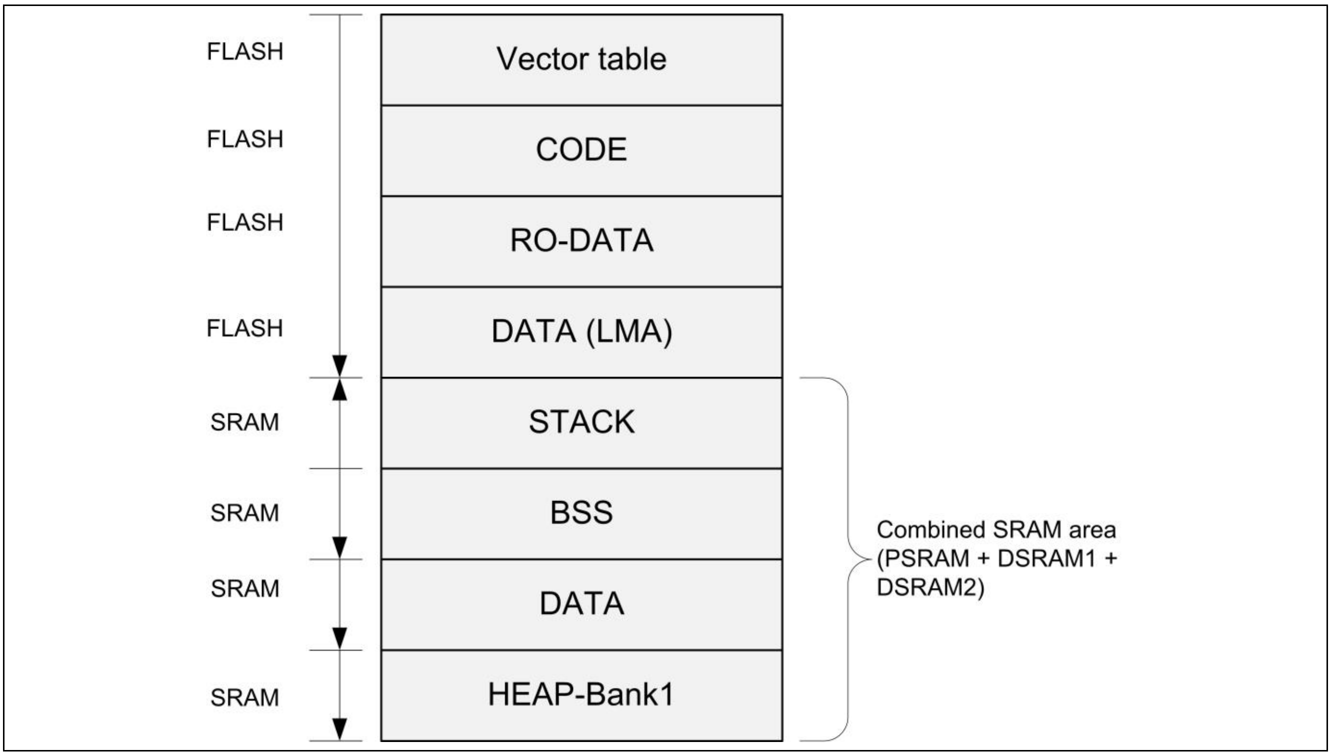

Figure 7 Typical section placement on a XMC4400 and XMC4200 device

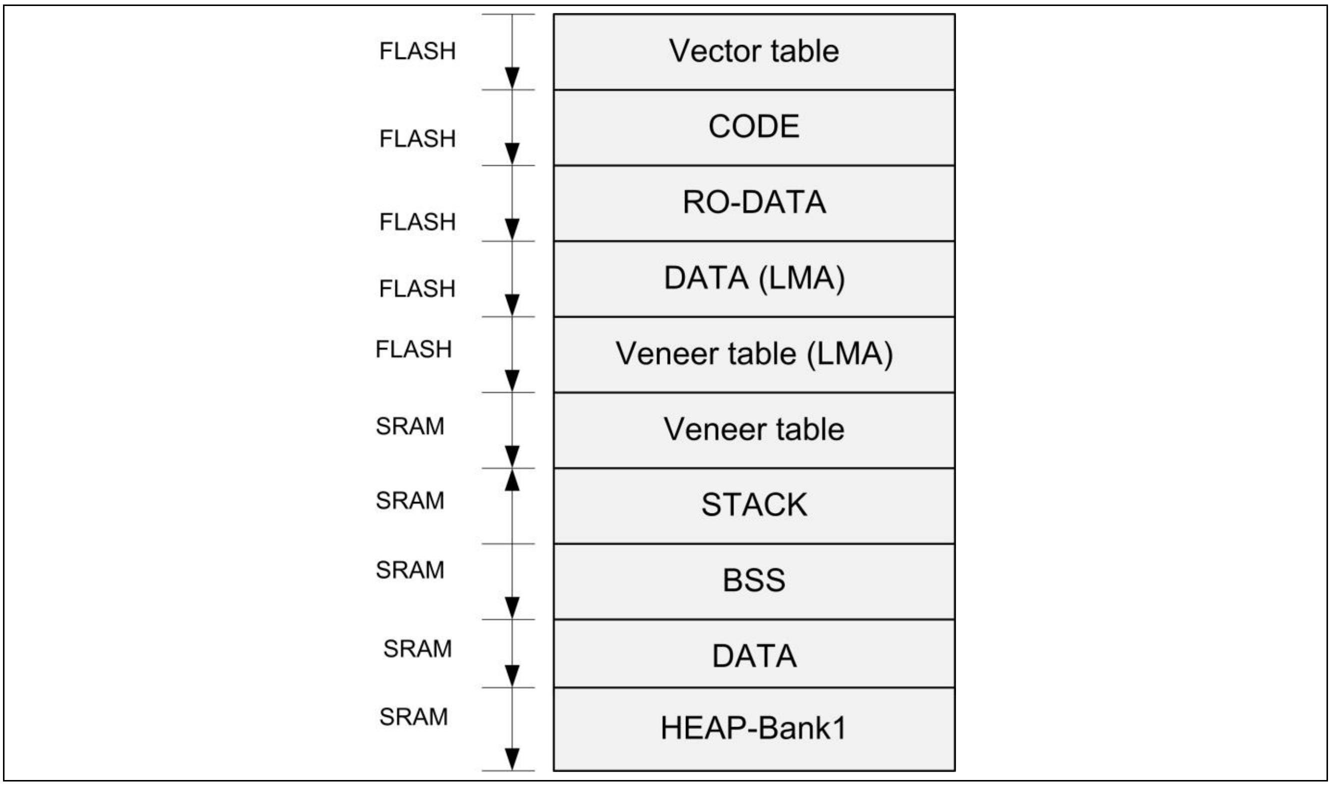

Figure 8 Typical section placement on a XMC1xxx device

3.4 Elaboration of GNU linker scripts written for XMC4 devices

OUTPUT_FORMAT("elf32-littlearm")(Indicates that the endianness of the CPU is little endian)

OUTPUT_ARCH(arm) (Indicates that the CPU architecture is ARM)

ENTRY(__Xmc4400_reset_cortex_m)(Indicates that **the entry point of the application is this function)**

GROUP(-lxmclibcstubs) (Indicates **that a pre-compiled library libxmclibcstubs.a is to be linked in on a need basis)**

-
- MEMORY (Defines various memory regions and their attributes)
- {
- FLASH_1_cached(RX) : ORIGIN = 0x08000000, LENGTH = 0x80000
- FLASH_1_uncached(RX) : ORIGIN = 0x0C000000, LENGTH = 0x80000
- PSRAM_1(!RX) : ORIGIN = 0x1FFFC000, LENGTH = 0x4000

```
    DSRAM_1_system(!RX)    : ORIGIN = 0x20000000, LENGTH = 0x8000
 DSRAM_2_comm(!RX)    : ORIGIN = 0x20008000, LENGTH = 0x8000
 SRAM_combined(!RX)    : ORIGIN = 0x1FFFC000, LENGTH = 0x14000
}
```

stack_size = 2048; (**Defines the stack size and can be modified as required**)

SECTIONS (**All output sections are defined and assigned to memory regions above**)

```
{
  Output_Section : AT (Load address)
```

(To be interpreted as "This output section must be linked to Memory_Region but loaded into Load_Address")

```
  {
   Section_Start_Label = .;
```

(**Used to indicate start address of output section**)

```
   *(Input Section1);
```

(Indicates input sections to be included in this output section)

```
   *(Input Section2);
   Section_End_Label = .;
```

(**Used to indicate end address of output section)**

```
} > Memory_Region

}
```

Without the AT attribute, the LMA and VMA of an output section are the same.

3.4.1 Examples

Example1: Positioning of startup code and standard .text

LMA = Uncached flash, VMA = Cached flash

```
    .text : AT (ORIGIN(FLASH_1_uncached))
    {
        sText = .;
        *(.Xmc4500.reset);
        *(.Xmc4500.postreset);
        *(.XmcStartup);
        *(.text .text.* .gnu.linkonce.t.*);
  }>FLASH_1_cached
```

Example2: Positioning of .data

LMA = Uncached flash, , VMA = SRAM

```
    .data ABSOLUTE(ALIGN(16)): AT(eROData)
    {
        __Xmc4500_sData = .;
        * (.data);
        * (.data*);
        *(*.data);
        *(.gnu.linkonce.d*)
        __Xmc4500_eData = ALIGN(4);
    } > DSRAM_1_system
```

Execution profiles

4 Execution profiles

The purpose of this chapter is to touch upon execution profiles and how end users may modify linker scripts and startup files to suit their own purpose.

Most embedded systems typically support three types of execution profiles

- Execute in Place (XIP) (Code in Non-Volatile memory)
- XIP + Time critical code in volatile memory
- Execution from volatile memory

4.1 Execute in Place (XIP)

This is the default profile available for projects created using the DAVE code generation tool. Code executes entirely from Flash memory. Variable data is hosted on volatile memory PSRAM.

4.2 XIP + Time critical code in volatile memory

Some time-critical parts of the code have a need to execute from a faster memory (typically SRAM).

XMC4 devices have a Program SRAM block (PSRAM) which serves this purpose. Such code is linked to the PSRAM address space. XMC1 devices simply call it the SRAM. During program loading, the program loader copies code from flash to the PSRAM area. There are usually three steps involved in accomplishing this.

- Creating dedicated sections for time critical code and assigning them a VMA in SRAM
- Assigning time critical code to dedicated sections during compilation time
- Loading time critical code from LMA of aforesaid sections to VMA in PSRAM

4.2.1 Example: Creating a dedicated section for time-critical code in the linker script

```
/* Define LMA of dedicated section */
sIRAMCodeLoad = eROData + __Xmc4500_Data_Size;
```

***Attention:** In this example, LMA of the dedicated section is after LMA of DATA section*

```
/* Assign special code to this dedicated section */
IRAM_Code : AT (sIRAMCodeLoad)
  {
   sIRAMCode = ABSOLUTE(.);
   *  (.IRAMCode);  (All  time  critical  code  assigned  to  .IRAMCode  goes  into
IRAM_Code which is linked to PSRAM_1)
   . = ALIGN(4);
   eIRAMCode = ABSOLUTE(.);
  } > PSRAM_1
  IRAMCodeSize      = eIRAMCode - ORIGIN(PSRAM_1);
```

4.2.2 Example: Assigning time critical code to dedicated sections in source code

```
void Time_Critical_Routine(void) __attribute__((section(".IRAMCode"));
```

4.2.3 Example: Program loader modification for loading time critical code to SRAM

Attention: ***BSS clearing code typically follows Data copy code. In this case however, the PSRAM code loader code follows Data copy and is in turn followed by BSS clear code.***

```
   B DATACOPYLOOP

SKIPDATACOPY: (Note the code specifically introduced for copying the dedicated
section from LMA to VMA)
   /* IRAM code copy */
   /* R0 = Start address, R1 = Destination address, R2 = Size */
   LDR R0, =sIRAMCodeLoad (Start of LMA)
   LDR R1, =sIRAMCode (Start of VMA)
   LDR R2, =IRAMCodeSize

   /* Is there anything to be copied? */
   CMP R2,#0
   BEQ SKIPIRAMCODECOPY

   /* For bytecount less than 4, at least 1 word must be copied */
   CMP R2,#4
   BCS STARTIRAMCODECOPY

   /* Byte count < 4 ; so bump it up */
   MOV R2,#4

STARTIRAMCODECOPY:
   /*
      R2 contains byte count. Change it to word count. It is ensured in the
      linker script that the length is always word aligned.
   */
   LSR R2,R2,#2 /* Divide by 4 to obtain word count */

   /* The proverbial loop from the schooldays */
IRAMCODECOPYLOOP:
   LDR R3,[R0]
   STR R3,[R1]
   SUBS R2,#1
   BEQ SKIPIRAMCODECOPY
   ADD R0,#4
   ADD R1,#4
   B IRAMCODECOPYLOOP

SKIPIRAMCODECOPY:
```

```
/* BSS CLEAR */
```

4.3 Complete code execution from SRAM

There are cases where the whole of the TEXT section is to be executed from SRAM. This is accomplished by linking the TEXT section to PSRAM addresses. Optionally, any constant data needed by the TEXT can also be linked to the PSRAM address space.

The linker script and the program loader code change. User applications do not change.

4.3.1 Example: Linker script modification

```
    /* TEXT section */

 Startup : AT(ORIGIN(FLASH_1_uncached))
    {
     StartText = ABSOLUTE(.);
     *(.Xmc4500.reset);
     *(.Xmc4500.postreset);
     *(.XmcStartup);
     . = ALIGN(4);
     eStartup = ABSOLUTE(.);
    } > FLASH_1_cached
```

Note, it is only the vector table and startup code that are linked to flash address space. Standard .text section is separated out and linked to PSRAM section below. LMA of this section is in flash while the VMA is in PSRAM.

```
     StartupSize = eStartup - StartText;
     sUserTextLoad = ORIGIN(FLASH_1_uncached)+ StartupSize;
.text ABSOLUTE(ORIGIN(PSRAM_1)): AT(sUserTextLoad)
    {
      sUserTextRun = ABSOLUTE(.);
      *(.text .text.* .gnu.linkonce.t.*);
      . = ALIGN(4);
      eUserTextRun = ABSOLUTE(.);
   } > PSRAM_1
      UserTextSize = eUserTextRun - sUserTextRun;
```

4.3.2 Example: Program loader modification

```
    /* Copy the TEXT from flash to PSRAM */
    /* R0 = Start address, R1 = Destination address, R2 = Size */
    LDR R0, =sUserTextLoad
    LDR R1, =sUserTextRun
    LDR R2, =UserTextSize

STARTPROGCOPY:
    /*
       R2 contains byte count. Change it to word count. It is ensured in the
       linker script that the length is always word aligned.
    */
    LSR R2,R2,#2 /* Divide by 4 to obtain word count */

    /* The proverbial loop from the schooldays */
PROGCOPYLOOP:
    LDR R3,[R0]
    STR R3,[R1]
    SUBS R2,#1
    BEQ SKIPPROGCOPY
    ADD R0,#4
    ADD R1,#4
    B PROGCOPYLOOP

SKIPPROGCOPY:
     /* Copy RODATA from flash to PSRAM */
    /* R0 = Start address, R1 = Destination address, R2 = Size */
    LDR R0, =sRODataLoad
    LDR R1, =sRODataRun
    LDR R2, =RODataSize

    /* Is there anything to be copied? */
    CMP R2,#0
    BEQ SKIPRODATACOPY

    /* For bytecount less than 4, at least 1 word must be copied */
    CMP R2,#4
    BCS STARTRODATACOPY

    /* Byte count < 4 ; so bump it up */
    MOV R2,#4

STARTRODATACOPY:
    /*
       R2 contains byte count. Change it to word count. It is ensured in the
```

```
      linker script that the length is always word aligned.
   */
   LSR R2,R2,#2 /* Divide by 4 to obtain word count */

   /* The proverbial loop from the schooldays */
RODATACOPYLOOP:
   LDR R3,[R0]
   STR R3,[R1]
   SUBS R2,#1
   BEQ SKIPRODATACOPY
   ADD R0,#4
   ADD R1,#4
   B RODATACOPYLOOP

SKIPRODATACOPY:
   /* Copy DATA from flash to DSRAM1 */
   /* R0 = Start address, R1 = Destination address, R2 = Size */
   LDR R0, =DataLoad
   LDR R1, =__Xmc4500_sData
   LDR R2, =__Xmc4500_Data_Size

   /* Is there anything to be copied? */
   CMP R2,#0
   BEQ SKIPDATACOPY

   /* For bytecount less than 4, at least 1 word must be copied */
   CMP R2,#4
   BCS STARTDATACOPY

   /* Byte count < 4 ; so bump it up */
   MOV R2,#4

STARTDATACOPY:
```

***Attention:* TEXT, RO-DATA and DATA are copied to the VMA area from their LMA area by the program loader.**

Device initialization hints

5 Hints on device initialization

5.1 Timing of device initialization

The purpose of this chapter is to elaborate upon device initialization topics.

Attention: ***It must be expressly stated that the scope of device initialization is entirely dependent on the end user. There are no fixed rules on what constitutes device initialization.***

Some examples that constitute device initialization are:

- Clock tree configuration
- Reset configuration
- Peripheral enabling and initialization
- Interrupt configuration

There are no set rules on the timing of device initialization. It can be performed at any time. In some cases, this is done before program loading, sometimes after program loading and before application entry, and many times this is entirely handled by the user application.

5.2 Clock and reset configuration

5.2.1 System clock configuration algorithm

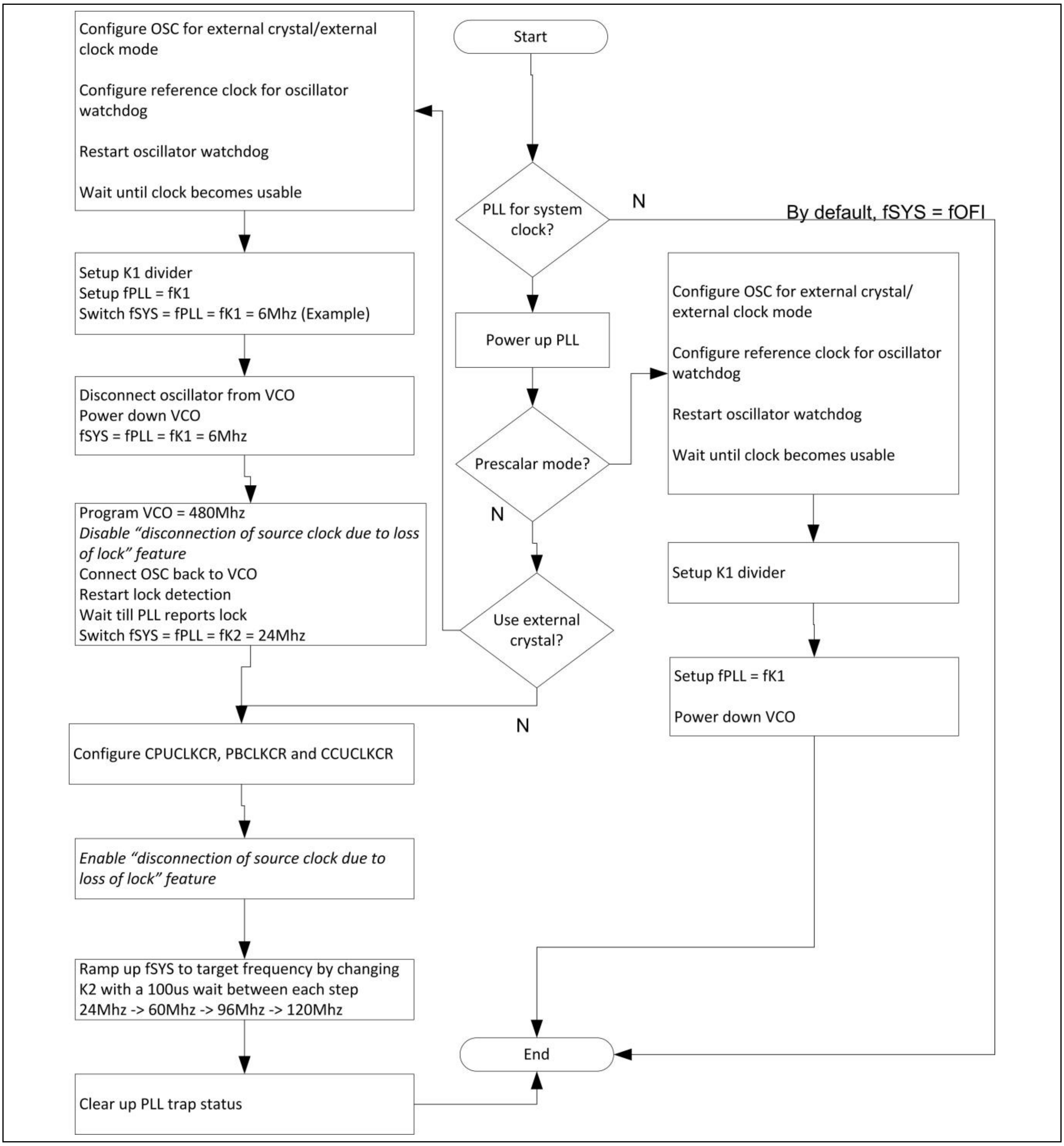

Figure 9 **System Clock selection algorithm**

- The System clock can be sourced by either the backup clock or by the PLL.
- The PLL can be operated in either a prescalar mode or the normal mode.
- Input clock to PLL can either be the backup clock or the external clock.
- The high precision oscillator can be turned off if the backup clock is being used as a clock source (either as an input to PLL or directly used as the system clock).

- The VCO can be powered down if the system clock is to be derived from either the backup clock or the high precision oscillator clock (PLL prescalar mode).

5.2.2 Using the backup clock as system clock (fSYS = fOFI)

```
/* Set fSYS to fOFI */
SCU_CLOCK->SYSCLKCR = 0U;
```

5.2.3 Using the oscillator clock as system clock (fSYS = fPLL)

```
/* Setup oscillator and connect crystal/clock to it - See next hint*/

/* Get PLL out of power saving mode */
SCU_PLL->PLLCON0 &= ~(1U << 16);

/* Select oscillator output as input to P and K1 divider */
SCU->PLLCON2 = 0U;  /* PINSEL = K1INSEL = fOHP */

/* Configure K1 divider */
SCU_PLL->PLLCON1 = 0U; /* No division */

/* Bypass the VCO - fPLL = fK1*/
SCU_PLL->PLLCON1 |= 1U; /* Bypass the VCO */

/* Power the VCO down to save power */
SCU_PLL->PLLCON1 |= 1U << 1U;

/* Set fSYS to fPLL. fSYS = fPLL = fK1 */
SCU_CLOCK->SYSCLKCR = 1U << 16U;
```

5.2.4 Downscaling the System clock (fSYS)

```
/* Program SYSDIV - Divide source of system clock by 8 (Example)*/
SCU_CLOCK->SYSCLKCR |= 7U;
```

5.2.5 Setting up the oscillator

```
/* Configure oscillator mode - E.g 20MHZ Crystal connected to oscillator */
SCU_OSC->OSCHPCTRL = ~(0x3U << 4);

/* Configure OSCVAL to generate correct osc watchdog reference clock of 2.5Mhz*/
SCU_OSC->OSCHPCTRL |= 7U << 16;

/* Get PLL out of power saving mode */
SCU_PLL->PLLCON0 &= ~(1U << 16);

/* Restart oscillator watchdog */
SCU_PLL->PLLCON0 |= 1U << 17;

/* Wait until oscillator output is deemed usable */
while (3 != ( ( (SCU_PLL->PLLSTAT)& (3U << 7)) >> 7) );
```

5.2.6 VCO output configuration and PLL lock

```
/* Example, fRef = 12Mhz, VCO output to 480Mhz, */

/* Get PLL out of power saving mode */
SCU_PLL->PLLCON0 &= ~(1U << 16);

/* Setup either backup clock/osc output as system clock - See previous hints */
/* So that fSYS=fOFI or fSYS=fPLL where fPLL=fK1 */

/* Disable PLL input */
SCU_PLL->PLLCON0 |= (1U << 4);

/* Take VCO out of power down mode */
SCU_PLL->PLLCON0 &= ~(1U << 1);

/* Define trim range for normal operation */
SCU_PLL->PLLCON0 &= ~(1U << 2);

/* Temporarily disable disconnection of source upon loss of PLL lock */
SCU_PLL -> PLLCON0 |= 1U << 6;

/* Program N=40, P=1, K2=1 dividers - Set PDIV, K2DIV and NDIV */
SCU_PLL -> PLLCON1 |= 1U << 24; /* P = PDIV + 1 = 2 */
SCU_PLL -> PLLCON1 &= ~(0x7F << 16); /* K2 = K2DIV + 1 = 1 */
SCU_PLL -> PLLCON1 |= (39U << 8); /* N = NDIV + 1 = 40 */

/* Enable PLL input */
SCU_PLL->PLLCON0 &= ~(1U << 4);

/* Re-Start VCO lock detection - Set RESLD */
SCU_PLL -> PLLCON0 |= 1U << 18;

/* Wait until VCO output locks to input frequency */
while(!(((SCU_PLL->PLLSTAT) &(1U << 2))>>2));

/* Re-enable disconnection of source upon loss of PLL lock */
SCU_PLL -> PLLCON0 &= ~(1U << 6);

/* Enable NORMAL mode of operation - fPLL = fK2 = fVCO / K2*/
SCU_PLL->PLLCON0 &= ~ (1U);
```

5.2.7 Target frequency ramp-up

```
/* Hint to ramp up fPLL to 120Mhz */
/* Example - Entry conditions
PLL = Normal mode
fSYS = fPLL = fK2
VCO output = 480Mhz, Locked to input
N and P dividers programmed, K2 set to 20
fK2 = 24Mhz
*/
/* Rampup fK2 to 60Mhz by changing K2 to 8 */
/* Wait for 50us */

/* Rampup fK2 to 96Mhz by changing K2 to 5 */
/* Wait for 50us */

/* Rampup fK2 to 120Mhz by changing K2 to 4 */
/* Wait for 50us */
```

5.2.8 CPU clock and peripheral bus clock

The CPU clock is sourced by fSYS. It can be halved if desired. Otherwise, fCPU is the same as fSYS.

```
SCU_CLOCK->CPUCLKCR = 1U; /* fCPU = fSYS/2 */
```

The Interface clock for peripherals is derived from fCPU. Usually, fPERIPH is the same as fCPU, but can be halved if required.

```
SCU_CLOCK ->PBCLKCR = 1U; /* fPERIPH = fCPU/2 */
```

5.2.9 Peripheral clock configuration

Most peripheral clocks derived out of fSYS can be downscaled by using dividers. There are dedicated clock control registers available. As an example, CCU clock division can be controlled by programming CCUDIV bit.

```
SCU_CLK->CCUCLKCR = 1U; /* fCCU = fSYS / 2 */
```

5.2.10 Peripheral clock control

Peripheral clocks can be individually controlled using CLKSET and CLKCLR registers.

The status of a peripheral clock can be found by reading the CLKSTAT register.

```
SCU_CLOCK->CLKSET |= (1U << 5U); /* To enable WDG clock */
SCU_CLOCK ->CLKCLR &= ~(1U << 5U); /* To disable WDG clock */
```

5.2.11 Clock gating during sleep modes

The behavior of various peripheral clocks can be controlled on entry to sleep modes. Registers SLEEPCR and DEEPSLEEPCR need to be programmed for this.

For example, in order to turn off the CCU clock during sleep and deep sleep modes, use:

```
SCU_CLK->SLEEPCR |= (1U << 20U);
SCU_CLK->DEEPSLEEPCR |= (1U << 20U);
```

5.2.12 Reset control

All peripherals are kept in a state of reset which must be released before they are programmed for the required functionality. Conversely, an active peripheral can be taken into a state of reset.

```
SCU_RESET -> PRCLR0 |= (1U << 11U); /* Release the reset on USIC0 */
SCU_RESET -> PRSET0 |= (1U << 0U); /* Reset VADC */
```

The module reset state can be found by reading the PRSTAT register.

5.3 Interrupt management

Modules generate events which potentially can lead to interrupts. To accomplish this, Interrupts must be enabled at:

- Module level
- NVIC level
- CPU level

Interrupt handlers must be defined which override the default definitions from C-Start.

Attention:* *Interrupts on XMC devices are known as service requests.

When an interrupt occurs, the CPU stops executing the main program and instead executes the interrupt handler routine. Once an interrupt has been handled, control returns back to the main program.

The code snippet that follows shows how LEDTS interrupts may be handled on a XMC4500 device. LEDTS service request is connected to Node-102 of NVIC.

5.3.1 Enabling of interrupts at module and NVIC level

```
LEDTS0->GLOBCTL |= 1U << 13; /* Enable time slice interrupt */
NVIC_SetPriority(102, 60); /* Assign a priority of 60 to Node-102 */
NVIC_EnableIRQ(102); /* Enable IRQ-102 */
```

5.3.2 Interrupt handler definition (Overriding default handler)

C-Start defines a default interrupt handler for all of the CPU exceptions and device interrupts. For the interrupt enabled above, user applications may define a final handler to meet their particular needs.

```
void LEDTS0_0_IRQHandler(void){} /* overrides the default interrupt handler */
```

5.4 Putting it all together

The following example pieces all of the information together.

```
int main(void)
{
/* Select fOFI as system clock */
SCU_CLOCK -> SYSCLKCR &= ~(1U << 16U);

/* Release the LEDTS reset */
SCU_RESET-> PRCLR1 |= (1U << 3);

/* Do not proceed until and unless the reset is confirmed as released */
while(1){
unsigned Status = SCU_RESET->PRSTAT1;
Status = (Status>>3) & 1;
if(!Status) break;
}

/* LEDTS configuration */
LEDTS0->GLOBCTL = 1U << 0; /* Enable Touchsense functionality */
LEDTS0->GLOBCTL |= 1U << 1; /* Enable LED functionality */
LEDTS0 ->GLOBCTL|= 1U << 13; /* Enable time slice interrupt */
NVIC_SetPriority(102, 60); /* Assign a priority of 60 to Node-102 */
NVIC_EnableIRQ(102); /* Enable IRQ-102 */

/* Other LEDTS configuration */

/* Finally */
while(1);/* All processing is now handled in the ISR */
}

void LEDTS0_0_IRQHandler(void)
{
  /* Confirm interrupt is genuine */
  /* Handle it – user application*/
}
```

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG