# C-Start and device initialization

## XMC1000

### About this document

### Scope and purpose

This application note provides an overview of the startup process and device initialization in the XMC1000 microcontroller family.

### Intended audience

This document is intended for engineers who have a good understanding of the XMC1000 microcontrollers.

## Table of contents

# 1 Introduction

The purpose of this user guide / application note 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.

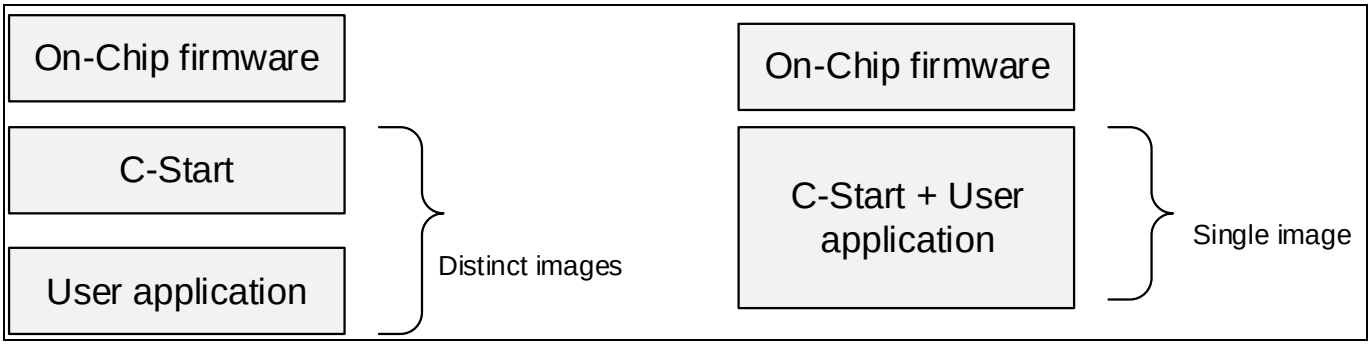


**Figure 1 C-Start tasks**

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.

# 2 C-Start tasks

This chapter elaborates upon the various tasks of C-Start. A Cortex-M0 CPU-based XMC1000 device is used in the illustrations that follow. Code fragments have been taken from the following files available with the DAVE™ distribution:

- startup_XMC1200.s
- system_XMC1200.c

## 2.1 Device booting

The following diagram indicates that after the reset (either Power-On-Reset or System reset) has been released, the CPU starts executing the startup software (SSW) stored in the ROM area of memory. The SSW execution stage is indicated by the pulling-up of P0.8. The role of the SSW is to evaluate the requested chip boot mode and take any necessary actions. As an example, if the chosen boot mode is ASC BSL mode, the SSW prepares to download the user application by first configuring the USIC peripheral for IO exchange and subsequently uploads the application into SRAM. The application is received over the UART IO lines.

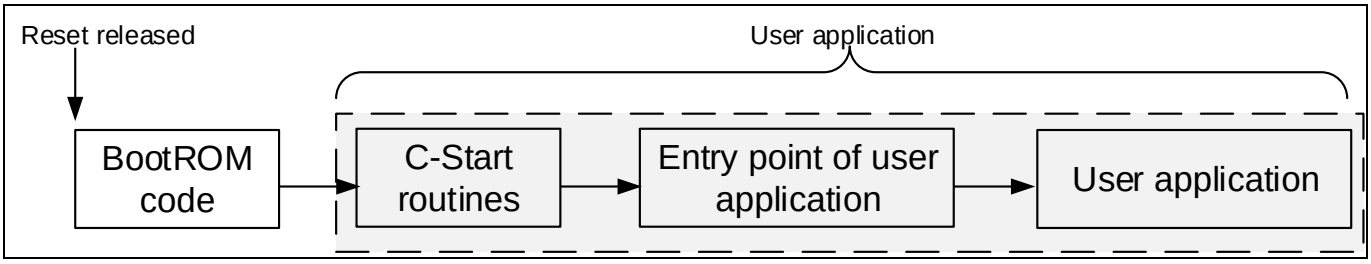


**Figure 2 Booting stages**

By default, the SSW runs with Main Clock (MCLK) frequency of 8 MHz, and all peripherals are disabled.

However, some user applications may require starting up with a different system frequency or configuration of enabled peripherals. For this purpose, two SSW-related data locations in flash are made available at 1000 $1010_H$ and 1000 $1014_H$ respectively. The code fragment below is an extract of the vector table written for the XMC1200 device.

```
__Vectors:
…
    .long   CLKVAL1_SSW              /* Clock configuration in SSW      */
    .long   CLKVAL2_SSW              /* Peripheral configuration in SSW */
```

CLKVAL1_SSW and CLKVAL2_SSW defined in the vector table point to the above mentioned flash addresses. The code fragment below shows the definition of both of them for the XMC1200 device.

```
#define CLKVAL1_SSW 0x00010400
#define CLKVAL2_SSW 0x80000000
```

These values can be changed directly in these definitions. These values will then be programmed to the respective addresses in flash when the application code is downloaded to the device.

In the SystemInit() function called by the reset handler, the MCLK frequency will be reinitialized to the maximum frequency, which is 32 MHz for XMC1100/1200/1300 devices and 48 MHz for the XMC1400 device. This is also the frequency at the point of user application entry.

If a user application is developed using DAVE™ CE, a different start up system frequency can be configured through the CLOCK_XMC1_0 APP (see section 5.2).

The User Mode with Debug Enabled (UMD) is a commonly deployed boot mode. On execution, the SSW reads the user application vector table, typically placed at the start of the flash area. It then extracts the start address of the C-Start routines and relinquishes control to the reset handler routine.

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

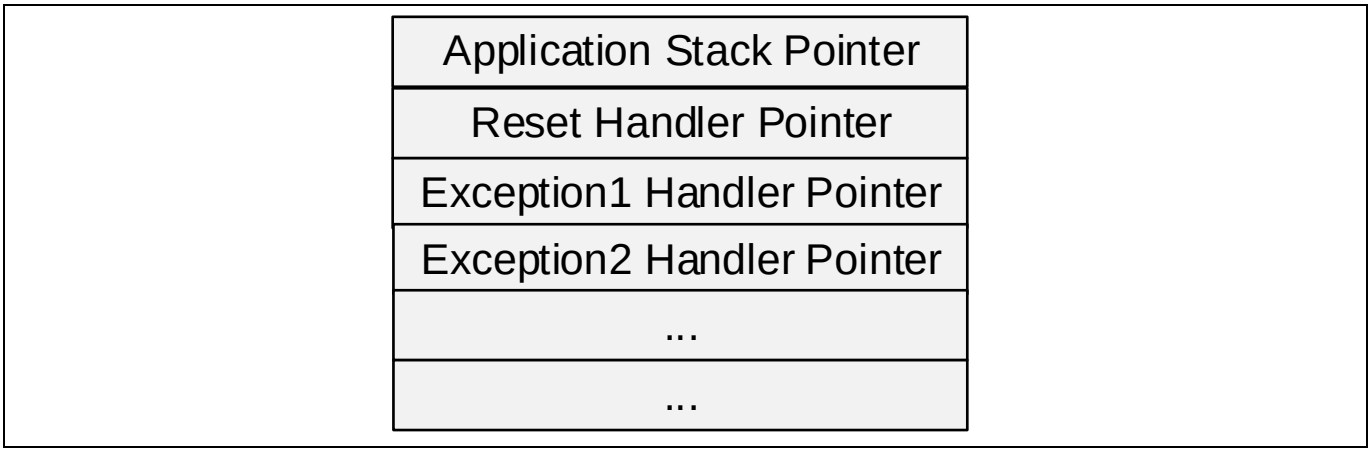


**Figure 3 Cortex-M vector table**

```
.syntax unified
.cpu cortex-m0

.section ".reset"

.align 2

.globl   __Vectors
.type    __Vectors, %object
__Vectors:
.long    __initial_sp               /* Top of Stack */
.long    Reset_Handler              /* Reset Handler */
.long    0                          /* Reserved */
Entry    HardFault_Handler          /* Hard Fault Handler */
/* Other vector table entries */
```

The code fragment listed above is an extract of the vector table written for the XMC1200 device for the GNU tool chain. The function pointer **Reset_Handler** is responsible for device initialization. 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 overwritten.*

## 2.2 Device initialization

The first stage of a typical reset handler written for an XMC1 device (specifically, the XMC1200) is shown below. Major functionality is highlighted in **bold.**

```
        .thumb_func
        .globl  Reset_Handler
        .type   Reset_Handler, %function

Reset_Handler:
/* Initialize interrupt veneer */
        ldr    r1, =eROData
        ldr    r2, =VeneerStart
        ldr    r3, =VeneerEnd
        bl   __copy_data

/* Clock tree, external memory setup etc may be done here */
        ldr  r0, =SystemInit
        blx  r0
```

It can be seen from the code fragment above 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 () 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 is executed, control is given to the next stage of the reset handler, known as 'Program loading'.

The job of a program loader (also known just as '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 typically to copy TEXT, RO-DATA and DATA from their load addresses (Load Memory Area or LMA) to their run addresses (Virtual Memory Area or 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 on in a subsequent chapter in this document.

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:

```
/* Initialize data */
       ldr    r1, =DataLoadAddr
       ldr    r2, =__data_start
       ldr    r3, =__data_end
       bl   __copy_data

/* RAM code */
       ldr    r1, =__ram_code_load
       ldr    r2, =__ram_code_start
       ldr    r3, =__ram_code_end
       bl   __copy_data

/* BSS section */
#ifndef __SKIP_BSS_CLEAR
       ldr    r1, =__bss_start
       ldr    r2, =__bss_end

       movs   r0, 0

       subs   r2, r1
       ble    .L_loop3_done

.L_loop3:
       subs   r2, #4
       str    r0, [r1, r2]
       bgt    .L_loop3
.L_loop3_done:
#endif /* __SKIP_BSS_CLEAR */

#ifndef __SKIP_LIBC_INIT_ARRAY
    ldr  r0, =__libc_init_array
    blx  r0
#endif
```

```
    ldr   r0, =main
    blx   r0

/* DATA COPY */
    .thumb_func
    .type __copy_data, %function
__copy_data:
       subs   r3, r2
       ble    .L_loop_done

.L_loop:
       subs   r3, #4
       ldr    r0, [r1,r3]
       str    r0, [r2,r3]
       bgt    .L_loop

.L_loop_done:
       bx  lr

       .pool
    .size   Reset_Handler,.-Reset_Handler
```

The program loader shown above 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 relinquished to the user application.

The vector table is remapped to locations in SRAM, with each vector allocated to a size of 4 bytes. Typically, the allocated size is not sufficient for an exception or interrupt. Therefore, a branch instruction is needed to jump to the actual handler at another location. This is achieved via veneers. Basically, a veneer acts as an intermediate target of the instruction and then sets the PC to the actual location. The code extract below is an example of a veneer code:

```
       .section ".XmcVeneerCode","ax",%progbits
       .align 1
       .globl HardFault_Veneer
   HardFault_Veneer:
       LDR R0, =HardFault_Handler
       MOV PC, R0
```

## 2.4 Giving control to the user application entry point

At this stage:

- The device initialization is complete
- Application data has been relocated from LMA to VMA

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

```
/* CEDE CONTROL TO ENTRY POINT OF USER APPLICATION */
 ldr  r0, =main
 blx  r0
```

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 implementing a self-looping program.

```
    .thumb_func
    .weak Default_handler
    .type Default_handler, %function
Default_handler:
    b  .
```

Users may in their application provide an alternative implementation of each 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 of interrupt handler definition in one of user's C files: */
void HardFault_Handler(void){}
```

# 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 the 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.

## 3.2 Memories on a typical XMC™ device

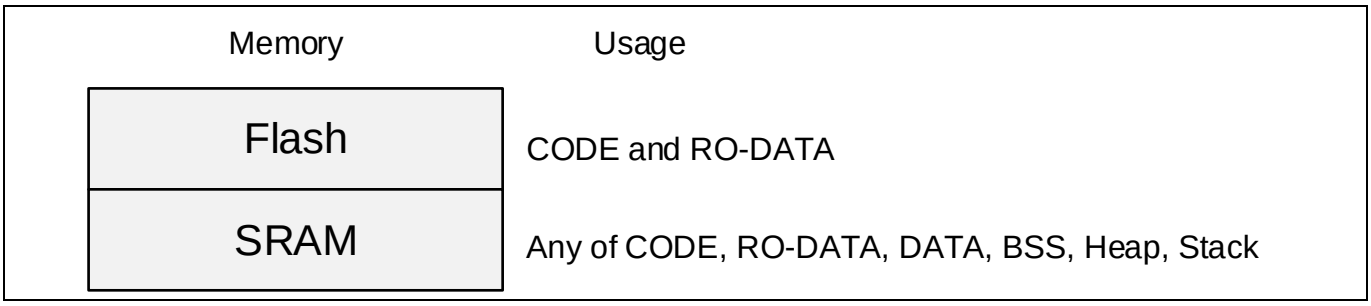


**Figure 4 XMC™ memories and usage**

## 3.3 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:

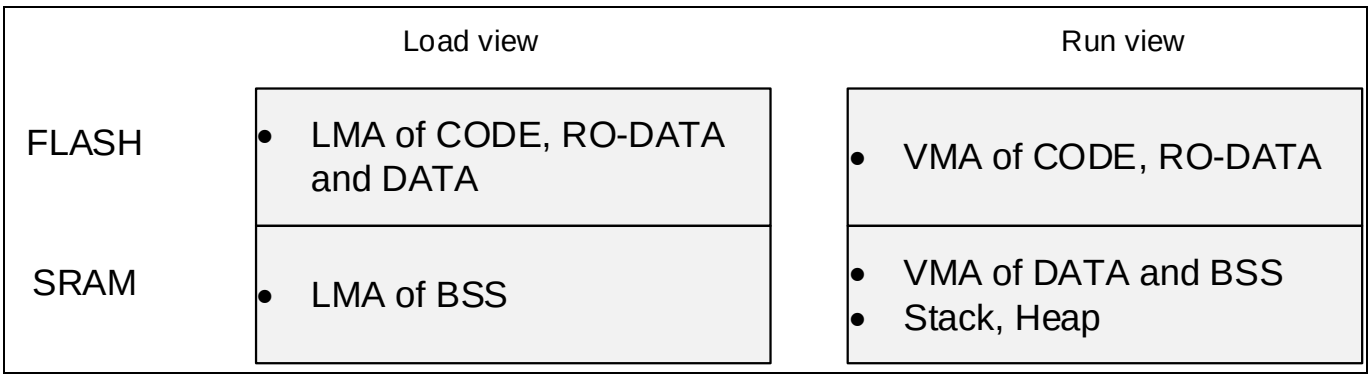


**Figure 5 LMA and VMA concept**

## 3.4 Typical program section assignment

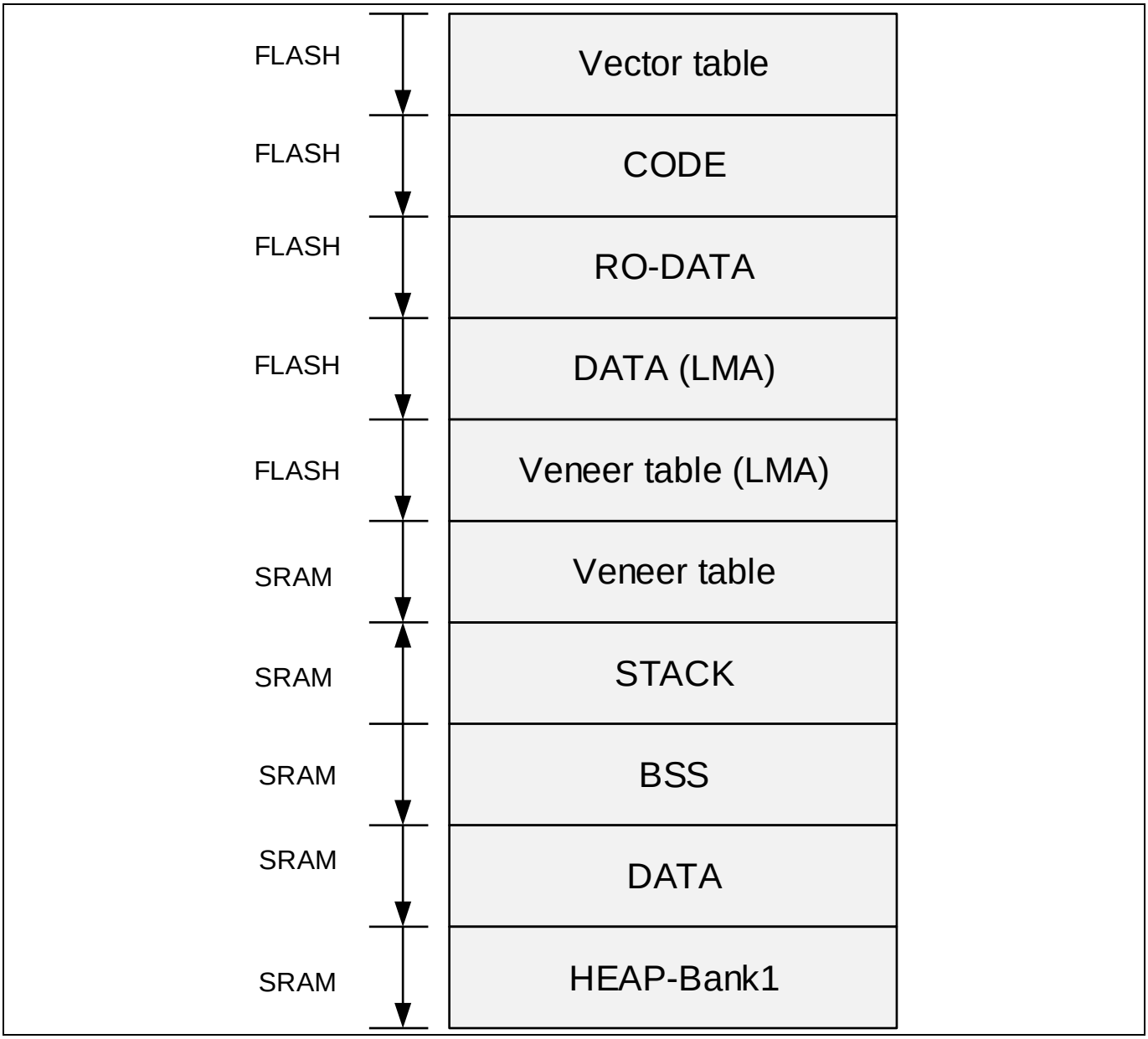


**Figure 6 Typical program section placement on an XMC1 device**

## 3.5 Elaboration of GNU linker scripts written for XMC1 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(Reset_Handler)
```

Linker scripts

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

```
MEMORY
```

(Defines various memory regions and their attributes)

```
{
  FLASH(RX)     : ORIGIN = 0x10001000, LENGTH = 0x32000
  SRAM(!RX)     : ORIGIN = 0x20000000, LENGTH = 0x4000
}

stack_size = 1024;
```

(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.

## Examples

### Example1: Positioning of startup code and standard .text

LMA = Flash, VMA = Flash

```
    .text : AT (ORIGIN(FLASH))
    {
        sText = .;
```

```
        *(.reset);
        *(.text .text.* .gnu.linkonce.t.*);
        …
    }>FLASH
```

### Example2: Positioning of .data

#### LMA = Flash, VMA = SRAM

```
    .data : AT(DataLoadAddr)
    {
        __data_start = .;
        * (.data);
        * (.data*);
        *(*.data);
        *(.gnu.linkonce.d*)
        . = ALIGN(4);
        __data_end = .;
    } > SRAM
```

# 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 SRAM.

## 4.2 XIP + time critical code in volatile memory

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

XMC1 devices have an SRAM block which serves this purpose. Such code is linked to the SRAM address space. During program loading, the program loader copies code from flash to the SRAM 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 SRAM

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

A dedicated section for time-critical code has been created in the linker script as shown below:

```
/* Define LMA of dedicated section */
    .ram_code : AT(DataLoadAddr + __data_size)
```

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

```
    {
        __ram_code_start = .;
/* functions with __attribute__ ((section (".ram_code"))) goes into ram_code
which is linked to SRAM */
        *(.ram_code)
        . = ALIGN(4);
        __ram_code_end = .;
    } > SRAM
    __ram_code_load = LOADADDR (.ram_code);
    __ram_code_size = __ram_code_end - __ram_code_start;
```

### 4.2.2 Example: assigning time-critical code to dedicated sections in source code

Time-critical code is assigned to the dedicated sections in the source code by adding the section attribute in the function declaration.

void Time_Critical_Routine(void) **__attribute__((section(".ram_code")));**

### 4.2.3 Example: program loader modification for loading time-critical code to SRAM

The copying of the dedicated section from LMA to VMA is done in the startup file as shown below:

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

```
/* RAM code */
       ldr    r1, =__ram_code_load
       ldr    r2, =__ram_code_start
       ldr    r3, =__ram_code_end
       bl   __copy_data

/* BSS CLEAR */

    .thumb_func
    .type __copy_data, %function
__copy_data:
       subs   r3, r2
       ble    .L_loop_done

.L_loop:
       subs   r3, #4
       ldr    r0, [r1,r3]
       str    r0, [r2,r3]
       bgt    .L_loop

.L_loop_done:
       bx  lr
```

## 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 SRAM addresses. Optionally, any constant data needed by the TEXT can also be linked to the SRAM 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 */

.text :
{
    sText = .;
    KEEP(*(.reset));
    …
} > FLASH
```

It is only the vector table and startup code that are linked to flash address space. The standard .text section is separated out and linked to the SRAM section below. The LMA of this section is in flash while the VMA is in SRAM.

In the following example, the TEXT section will be placed after the veneer code in the SRAM by the program loader in the startup file.

```
.VENEER_Code ABSOLUTE(0x2000000C): AT(eRODData)
{
    VeneerStart = .;
    KEEP(*(.XmcVeneerCode));
      *(.text .text.* .gnu.linkonce.t.*);
    . = ALIGN(4);
    VeneerEnd = .;
} > SRAM
```

# 5 Device initialization hints

## 5.1 Timing of device initialization

The purpose of this chapter is to elaborate upon device initialization topics. All example code shown in this chapter is implemented based on Infineon's XMC™ library (XMC™ Lib).

*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
- Peripheral suspend configuration
- 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 Configuring clock during startup software (SSW) execution

The MCLK is defined to 8 MHz by default in the startup code based on CLKVAL1_SSW and CLKVAL2_SSW values. It will be changed to the respective device's maximum frequency before control is relinquished to the user application entry point.

If the user is developing a DAVE™ CE project, the MCLK frequency for the XMC1400 device, also the MCLK source, can be changed in the UI of the CLOCK_XMC1_0 app, as shown in the following figures.

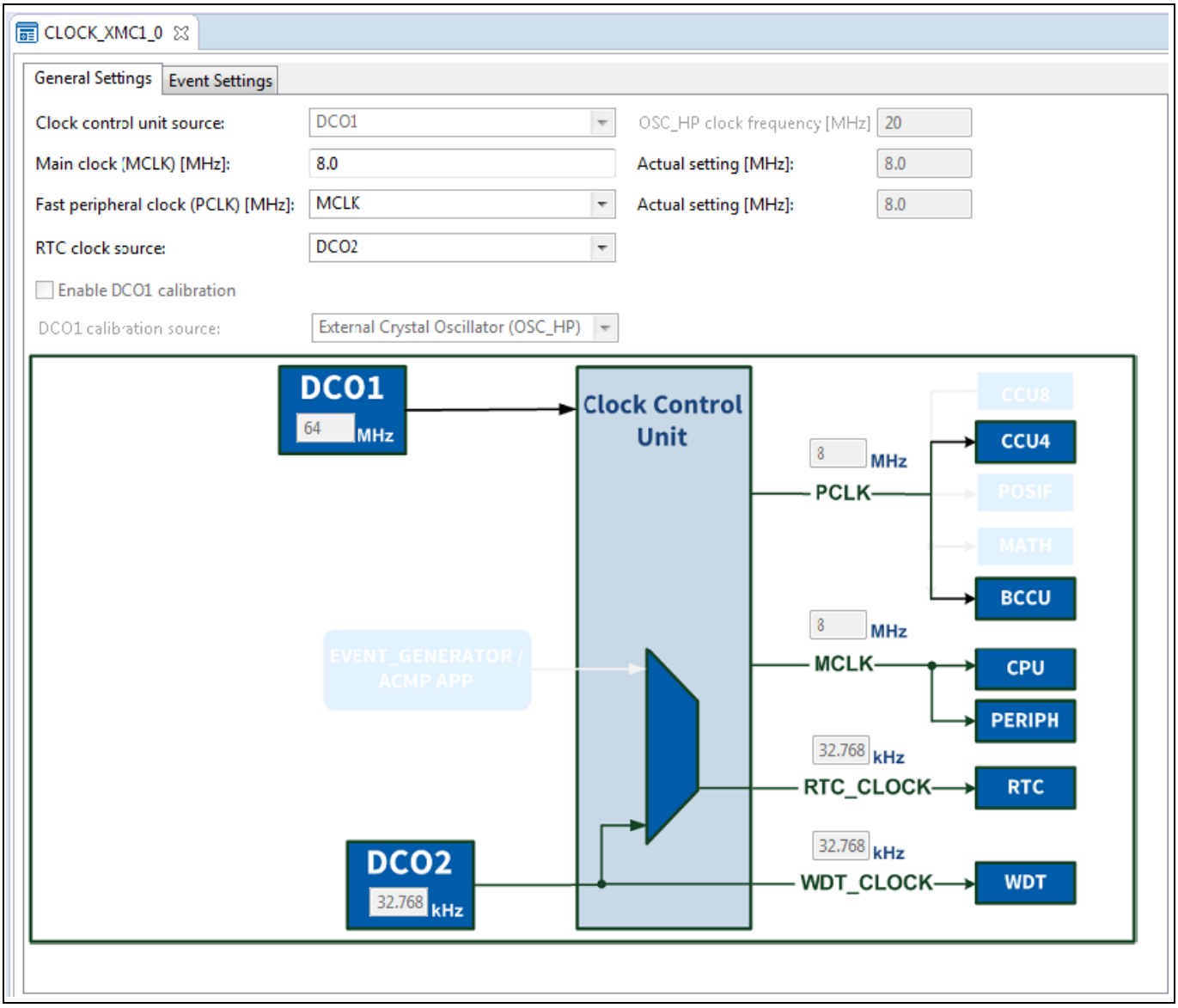


Figure 7 Changing XMC1100/1200/1300 MCLK frequency with CLOCK_XMC1_0 app

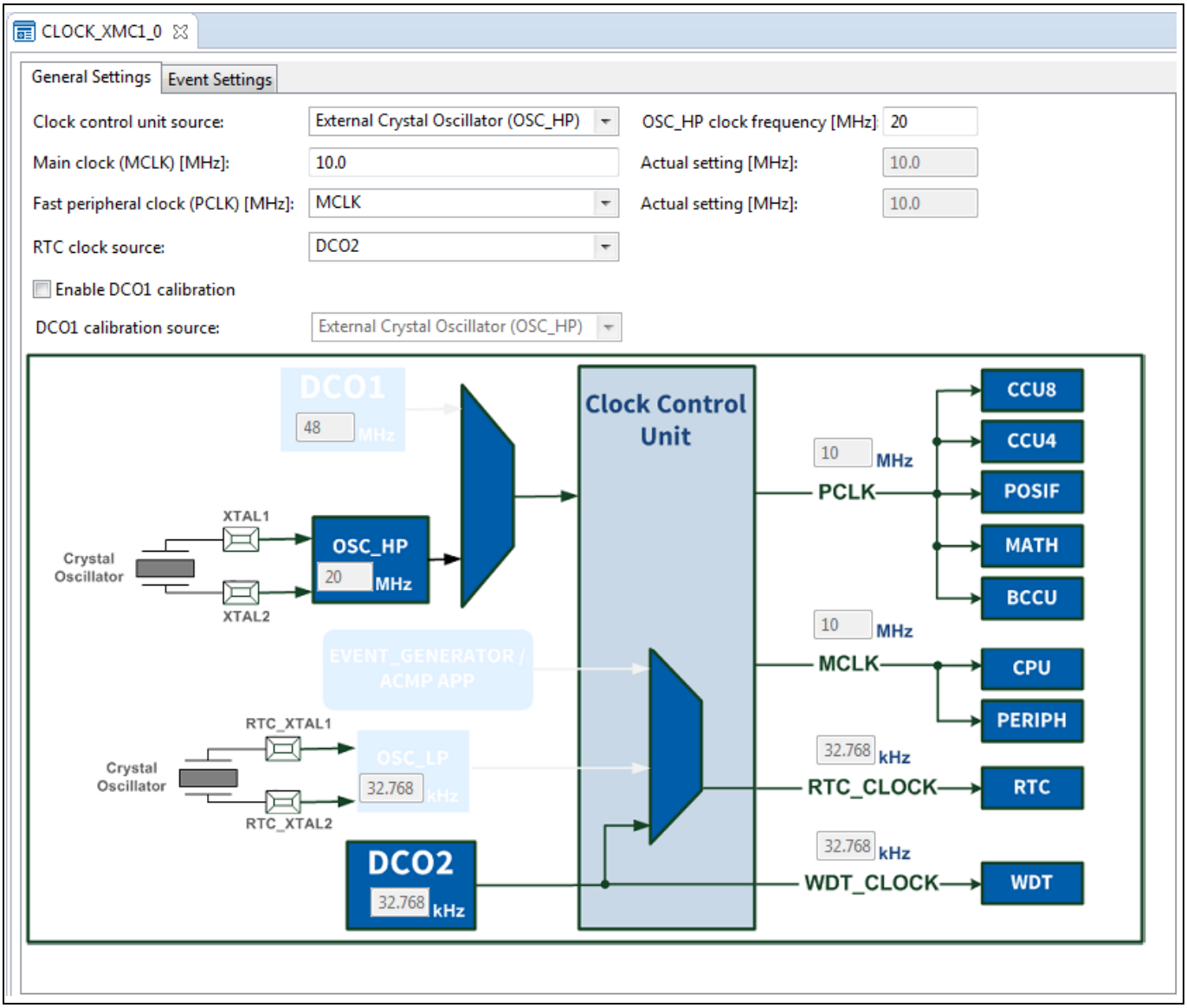


**Figure 8 Changing XMC1400 MCLK frequency with CLOCK_XMC1_0 app**

*Note: CLKVAL1_SSW and CLKVAL2_SSW are referred to as CLK_VAL1 and CLK_VAL2 respectively in the XMC1x00 Reference Manual (RM). Please refer to "Startup Software (SSW) Execution"section in the RM for more details.*

## 5.3 Controlling and handling reset

After system startup, it may be useful to check the cause of the previous reset. The register RSTSTAT holds this information. The status should be cleared upon reading to ensure a clear status at the next reset.

Below is an example code:

```
unsigned int status;
status = XMC_SCU_RESET_GetDeviceResetReason();    /* get the cause of reset */
XMC_SCU_RESET_ClearDeviceResetReason();           /* clear status field */
```

Additionally, critical events such as ECC error and loss of clock can be configured to trigger a reset. These event resets can be enabled or disabled via register RSTCON.

Below is an example code:

```
/* enable ECC and loss of clock reset */
XMC_SCU_RESET_EnableResetRequest(XMC_SCU_RESET_REQUEST_FLASH_ECC_ERROR);
XMC_SCU_RESET_EnableResetRequest(XMC_SCU_RESET_REQUEST_CLOCK_LOSS);
```

## 5.4 Configuring clock in user code

Configuring the clocks may cause a load change which may lead to clock blanking when the $V_{DDP}$ value drops below the threshold value, also known as VDROP event. It is therefore essential to check for the VDROP event, and wait accordingly for the $V_{DDP}$ to stabilize.

### XMC1100/1200/1300:

The clock for the XMC1100/1200/1300 devices can be simply configured via a single register, CLKCR. There are 4 parts or elements to be configured within this register:-

- Main Clock (MCLK) frequency
  The CPU and some of the peripherals are clocked by MCLK. MCLK has a range from 125 kHz to 32 MHz. MCLK can be adjusted via the divider bit fields, IDIV and FDIV.
- Peripheral Clock (PCLK) frequency
  Peripherals such as CCU8, POSIF, CCU4, MATH and BCCU are clocked by PCLK. PCLK can run at the same or double the frequency of MCLK. This option is configured via bit PCLKSEL.
- RTC Clock source
  RTC Clock source is configured via bit field RTCCLKSEL.
- Counter adjustment (CNTADJ)
  The CNTADJ value determines the length of the delay to let the $V_{DDP}$ stabilize in the event that the $V_{DDP}$ goes off the threshold value.

Here is some example code to configure the clock in the XMC1100/1200/1300 devices:

```
/* Data structure to configure standby clock as RTC clock source, PCLK =
2*MCLK, MCLK = 32MHz */
XMC_SCU_CLOCK_CONFIG_t CLOCK_XMC1_0_CONFIG =
{
  .pclk_src = XMC_SCU_CLOCK_PCLKSRC_DOUBLE_MCLK,
  .rtc_src = XMC_SCU_CLOCK_RTCCLKSRC_DCO2,
  .fdiv = 0U,
  .idiv = 1U
};
...
/* Clock initialization function */
XMC_SCU_CLOCK_Init(&CLOCK_XMC1_0_CONFIG);
```

### XMC1400:

The XMC1400 device additionally supports the use of an external crystal oscillator for accurate clock generation.

The clock for the XMC1400 device can be configured via the registers, CLKCR and CLKCR1. There are 4 parts or elements to be configured within these registers:-

- Main Clock (MCLK) frequency
  The CPU and some of the peripherals are clocked by MCLK. The MCLK can be source from either the on-chip oscillator DCO1 or an external clock via OSC_HP oscillator.
  The frequency of MCLK depends on the clock source:
  - With the on-chip oscillator DCO1, MCLK has a range from 188 kHz to 48 MHz.
  - While an external 4 MHz to 20 MHz crystal oscillator is supported with the OSC_HP.

  The MCLK source is configured via bit field DCLKSEL in the CLKCR1 register. MCLK can be further adjusted via the divider bit fields, IDIV and FDIV in CLKCR register.
- Peripheral Clock (PCLK) frequency
  Peripherals such as CCU8, POSIF, CCU4, MATH and BCCU are clocked by PCLK. PCLK can run at the same or double the frequency of MCLK. This option is configured via bit PCLKSEL.
- RTC Clock source
  RTC Clock source is configured via bit field RTCCLKSEL.
- Counter adjustment (CNTADJ)
  The CNTADJ value determines the length of the delay to let the $V_{DDP}$ stabilize in the event that the $V_{DDP}$ goes off the threshold value.

Here is some example code to configure the clock, which is sourced from an external oscillator crystal, in the XMC1400 device:

```
/* Data structure to enable OSC_HP and XTAL OSC watchdog, configure external
crystal as MCLK source, PCLK = 2*MCLK, MCLK = 20MHz */
XMC_SCU_CLOCK_CONFIG_t CLOCK_XMC1_0_CONFIG =
{
  .pclk_src = XMC_SCU_CLOCK_PCLKSRC_DOUBLE_MCLK,
  .rtc_src = XMC_SCU_CLOCK_RTCCLKSRC_DCO2,
  .fdiv = 0U,
  .idiv = 1U,
  .dclk_src = XMC_SCU_CLOCK_DCLKSRC_EXT_XTAL,
  .oschp_mode = XMC_SCU_CLOCK_OSCHP_MODE_OSC,
  .osclp_mode = XMC_SCU_CLOCK_OSCLP_MODE_DISABLED
};
…
/* Clock initialization function */
XMC_SCU_CLOCK_Init(&CLOCK_XMC1_0_CONFIG);
```

## 5.5 Controlling and initializing peripheral clocks

Clocks to peripherals are gated by default, unless already removed during SSW execution. Gating to peripheral clocks can be individually removed or asserted by setting the respective bits in CLKGATSET0 and CLKGATCLR0 registers. The status of a peripheral clock can be found by reading the CLKGATSTAT0 register.

Configuring the clocks may also cause a load change which may lead to clock blanking. It is therefore essential to check for the VDROP event, and wait for $V_{DDP}$ to stabilize.

Below is some example code:

```
XMC_SCU_CLOCK_UngatePeripheralClock(XMC_SCU_PERIPHERAL_CLOCK_LEDTS0);
```

After the gating to the peripheral clock has been removed, the specific peripheral initializations can then take place. Additionally, some peripherals such as the USIC have module enable bits which need to be set before initialization can begin proper. This information can be found in the Initialization and System Dependencies section of the module chapter in the XMC1x00 Reference Manual.

## 5.6 Peripheral initialization sequence

Some peripherals such as the LEDTS and the CCU4 start to run when the peripheral counter or global start bit is enabled. Their interrupts, especially those that occur at high frequencies, may interfere with the initialization of subsequent peripherals. In the worst case, the subsequent peripherals may not even be initialized. To avoid such situations, it is recommended that the enabling of such peripheral counters is carried out after all other peripheral initializations have been completed.

Below is some example code showing the initialization of the LEDTS0, CCU40 and USIC0:

```
/* LEDTS0 initialization instructions */
…
/* CCU40 initialization instructions */
…
/* USIC0 initialization instructions */
…
/* Start LEDTS-counter with CLK_PS=0x0080 */
XMC_LEDTS_StartCounter(XMC_LEDTS0, 0x0080);
/* Enable CCU40 global start bit based on active edge trigger*/
XMC_SCU_SetCcuTriggerHigh(XMC_SCU_CCU_TRIGGER_CCU40);
```

## 5.7 Configuring peripheral suspend

All peripherals except the PRNG have the peripheral suspend support feature. For these peripherals, the suspend mode is inactive by default, with the WDT being the only exception. The debug suspend behavior can be configured via a register bit or bit field within the respective peripherals. This is normally done during the peripheral initialization.

Below is an example code (not using XMC Lib) showing the debug suspend configuration for the LEDTS0 and CCU40:

```
LEDTS0->GLOBCTL |= (1 << 8); /* enable LEDTS-counter to suspend in debug */
CCU40->GCTRL |= (1 << 8); /*stop all running slices immediately in debug */
```

## 5.8 Configuring ports

Some peripherals, such as the USIC0 operating in the UART mode, require the use of port pin alternate functions, or the user application requires some general purpose input/output pin functions. These functions can be configured via the control registers in the ports.

Below is some example code showing the initialization of :

- P0.0 as a general purpose I/O

- P1.2 as the USIC0 transmit pin via alternate function 7
- P1.3 as the USIC receive pin

```
/* GPIO pin configuration data structure */
XMC_GPIO_CONFIG_t gpio_pin_config =
{
  .mode = XMC_GPIO_MODE_OUTPUT_PUSH_PULL,
  .output_level= XMC_GPIO_OUTPUT_LEVEL_HIGH
};
/* USIC0 transmit pin configuration data structure */
XMC_GPIO_CONFIG_t tx_pin_config =
{
  .mode = XMC_GPIO_MODE_OUTPUT_PUSH_PULL_ALT7,
};
/* USIC0 receive pin configuration data structure */
XMC_GPIO_CONFIG_t rx_pin_config =
{
  .input_hysteresis = XMC_GPIO_INPUT_HYSTERESIS_STANDARD,
  .mode = XMC_GPIO_MODE_INPUT_TRISTATE,
};
…
/* GPIO initialization function */
  XMC_GPIO_Init(P0_0, & gpio_pin_config);
  XMC_GPIO_Init(P1_2, &tx_pin_config);
  XMC_GPIO_Init(P1_3, &rx_pin_config);
```

## 5.9 Managing interrupts

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

- CPU level
- NVIC level
- Module 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.

### 5.9.1 Enabling of interrupts at CPU level

Interrupts can be enabled or disabled at CPU level with the intrinsic functions provided by CMSIS:

```
__disable_irq() /* disable all interrupts */
__enable_irq() /* enable all interrupts */
```

Interrupts are enabled by default. Hence there is no need to enable them at the start. However, these functions may be useful if the user application requires enabling or disabling of all interrupts.

### 5.9.2 Enabling of interrupts at NVIC and module level

**XMC1100/1200/1300:**

The enabling of interrupts at NVIC and module level is the same for these three devices.

The following code snippet depicts how LEDTS0 interrupts may be handled on a XMC1200 device. LEDTS0 service request is connected to Node-29 of NVIC.

```
/* Enable time frame interrupt */
XMC_LEDTS_EnableInterrupt(XMC_LEDTS0, XMC_LEDTS_INTERRUPT_TIMEFRAME);
NVIC_SetPriority(29, 0); /* Assign a priority of 0 (highest) to Node-29 */
NVIC_EnableIRQ(29); /* Enable IRQ-29 */
```

**XMC1400:**

On a XMC1400 device, each NVIC node can be assigned to one of three service request sources (Source A, B or C) or the logical OR of sources A and B.

Therefore, the enabling of interrupts requires an additional step to select the service request source for the NVIC node. This is done through the SCU register INTCRx (x = 0 or 1).

The following code snippet depicts how the CCU40 compare match while counting up interrrupt may be assigned to Node-8 of NVIC (Source B) on an XMC1400 device.

```
/* Enable compare match while counting up interrupt */
XMC_CCU4_SLICE_EnableEvent(CCU40_CC40,
XMC_CCU4_SLICE_IRQ_ID_COMPARE_MATCH_UP);
/* Select Source B for Node-8 */
XMC_SCU_SetInterruptControl(8, XMC_SCU_IRQCTRL_CCU40_SR0_IRQ8);
NVIC_SetPriority(8, 0); /* Assign a priority of 0 (highest) to Node-8 */
NVIC_EnableIRQ(8); /* Enable IRQ-8 */
```

### 5.9.3 Interrupt handler definition (overriding default handler)

C-Start defines the 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.

**XMC1100/1200/1300:**

For XMC1100/1200/1300 devices, the interrupt handler name is prefixed by the module name.

```
void LEDTS0_0_IRQHandler(void){}/*overrides the default interrupt handler for
Node-29*/
```

**XMC1400:**

For the XMC1400 device, the interrupt handler name is generic since the node may be assigned to different modules.

```
void IRQ8_Handler(void){}/*overrides the default interrupt handler for Node-
8*/
```

## 5.10 Putting it all together

The following example for the XMC1200 device pieces all of the information together.

```
/* System clock data structure */
XMC_SCU_CLOCK_CONFIG_t CLOCK_XMC1_0_CONFIG =
{
  .pclk_src = XMC_SCU_CLOCK_PCLKSRC_DOUBLE_MCLK,
  .rtc_src = XMC_SCU_CLOCK_RTCCLKSRC_DCO2,
  .fdiv = 0U,
  .idiv = 1U
};
/* GPIO pin configuration data structure */
XMC_GPIO_CONFIG_t gpio_pin_config =
{
  .mode = XMC_GPIO_MODE_OUTPUT_PUSH_PULL,
  .output_level= XMC_GPIO_OUTPUT_LEVEL_HIGH
};
/* USIC0 transmit pin configuration data structure */
XMC_GPIO_CONFIG_t tx_pin_config =
{
  .mode = XMC_GPIO_MODE_OUTPUT_PUSH_PULL_ALT7,
};
/* USIC0 receive pin configuration data structure */
XMC_GPIO_CONFIG_t rx_pin_config =
{
  .input_hysteresis = XMC_GPIO_INPUT_HYSTERESIS_STANDARD,
  .mode = XMC_GPIO_MODE_INPUT_TRISTATE,
};

int main(void)
{
unsigned int status;

/* Check reset status and perform reset configuration */
status = XMC_SCU_RESET_GetDeviceResetReason();   /* get the cause of reset */
/* Perform necessary tasks here in case of a certain reset */
XMC_SCU_RESET_ClearDeviceResetReason();          /* clear status field */
/* enable ECC and loss of clock reset */
```

```
XMC_SCU_RESET_EnableResetRequest(XMC_SCU_RESET_REQUEST_FLASH_ECC_ERROR);
XMC_SCU_RESET_EnableResetRequest(XMC_SCU_RESET_REQUEST_CLOCK_LOSS);

/* System clock configuration */
XMC_SCU_CLOCK_Init(&CLOCK_XMC1_0_CONFIG);

/* LEDTS0 initialization */
XMC_SCU_CLOCK_UngatePeripheralClock(XMC_SCU_PERIPHERAL_CLOCK_LEDTS0);
/* Other LEDTS0 configurations should be done here */
NVIC_SetPriority(29, 0); /* Assign a priority of 0 (highest) to Node-29 */
NVIC_EnableIRQ(29); /* Enable IRQ-29 */

/* CCU40 initialization */
XMC_SCU_CLOCK_UngatePeripheralClock(XMC_SCU_PERIPHERAL_CLOCK_CCU40);
/* Other CCU40 configurations should be done here */
NVIC_SetPriority(21, 64); /*Assign a priority of 64 (level 1) to Node-21*/
NVIC_EnableIRQ(21); /* Enable IRQ-21 */

/* USIC0 initialization */
XMC_SCU_CLOCK_UngatePeripheralClock(XMC_SCU_PERIPHERAL_CLOCK_USIC0);
/* Other USIC0 configurations should be done here */
NVIC_SetPriority(9, 128); /*Assign a priority of 128 (level 2) to Node-9*/
NVIC_EnableIRQ(9); /* Enable IRQ-9 */

/* GPIO initialization function */
  XMC_GPIO_Init(P0_0, & gpio_pin_config);
  XMC_GPIO_Init(P1_2, &tx_pin_config);
  XMC_GPIO_Init(P1_3, &rx_pin_config);

/* Start LEDTS-counter with CLK_PS=0x0080 */
XMC_LEDTS_StartCounter(XMC_LEDTS0, 0x0080);
/* Enable CCU40 global start bit based on active edge trigger*/
XMC_SCU_SetCcuTriggerHigh(XMC_SCU_CCU_TRIGGER_CCU40);

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

/* Interrupt handler definitions */
void LEDTS0_0_IRQHandler(void)
```

```
{
  /* Confirm interrupt is genuine */
  /* Handle it - user application*/
}

void CCU40_0_IRQHandler(void)
{
  /* Confirm interrupt is genuine */
  /* Handle it - user application*/
}

void USIC0_0_IRQHandler(void)
{
  /* Confirm interrupt is genuine */
  /* Handle it - user application*/
}
```

# 6 References

[1] XMC1000 Reference Manual at http://www.infineon.com/xmc1000 Tab: Documents

## Revision history

### Major changes since the last revision

| Page or reference | Description of change |
|---|---|
| -- | First Release |
| | |
| | |

**Edition 2016-02-22**

**Published by**

**Infineon Technologies AG**

**81726 Munich, Germany**



**Do you have a question about this document?**

**Email:** erratum@infineon.com

**Document reference**
**AP32326**

**IMPORTANT NOTICE**

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. 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) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

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

**WARNINGS**

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.