XMC1000

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 2013-04

Microcontrollers

Edition 2013-04

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2013 Infineon Technologies AG
All Rights Reserved.

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, 2013-04	

Trademarks of Infineon Technologies AG

AURIX™, C166™, CanPAK™, CIPOS™, CIPURSE™, EconoPACK™, CoolMOS™, CoolSET™, CORECONTROL™, CROSSAVE™, DAVE™, EasyPIM™, EconoBRIDGE™, EconoDUAL™, EconoPIM™, EiceDRIVER™, eupec™, FCOS™, HITFET™, HybridPACK™, I²RF™, ISOFACE™, IsoPACK™, MIPAQ™, ModSTACK™, my-d™, NovalithIC™, OptiMOS™, ORIGA™, PRIMARION™, PrimePACK™, PrimeSTACK™, PRO-SIL™, PROFET™, RASIC™, ReverSave™, SatRIC™, SIEGET™, SINDRION™, SIPMOS™, SmartLEWIS™, SOLID FLASH™, TEMPFET™, thinQ!™, TRENCHSTOP™, TriCore™.

Other Trademarks

Advance Design System™ (ADS) of Agilent Technologies, AMBA™, ARM™, MULTI-ICE™, KEIL™, PRIMECELL™, REALVIEW™, THUMB™, µVision™ of ARM Limited, UK. AUTOSAR™ is licensed by AUTOSAR development partnership. Bluetooth™ of Bluetooth SIG Inc. CAT-iq™ of DECT Forum. COLOSSUS™, FirstGPS™ of Trimble Navigation Ltd. EMV™ of EMVCo, LLC (Visa Holdings Inc.). EPCOS™ of Epcos AG. FLEXGO™ of Microsoft Corporation. FlexRay™ is licensed by FlexRay Consortium. HYPERTERMINAL™ of Hilgraeve Incorporated. IEC™ of Commission Electrotechnique Internationale. IrDA™ of Infrared Data Association Corporation. ISO™ of INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. MATLAB™ of MathWorks, Inc. MAXIM™ of Maxim Integrated Products, Inc. MICROTEC™, NUCLEUS™ of Mentor Graphics Corporation. Mifare™ of NXP. MIPI™ of MIPI Alliance, Inc. MIPS™ of MIPS Technologies, Inc., USA. muRata™ of MURATA MANUFACTURING CO., MICROWAVE OFFICE™ (MWO) of Applied Wave Research Inc., OmniVision™ of OmniVision Technologies, Inc. Openwave™ Openwave Systems Inc. RED HAT™ Red Hat, Inc. RFMD™ RF Micro Devices, Inc. SIRIUS™ of Sirius Satellite Radio Inc. SOLARIS™ of Sun Microsystems, Inc. SPANSION™ of Spansion LLC Ltd. Symbian™ of Symbian Software Limited. TAIYO YUDEN™ of Taiyo Yuden Co. TEAKLITE™ of CEVA, Inc. TEKTRONIX™ of Tektronix Inc. TOKO™ of TOKO KABUSHIKI KAISHA TA. UNIX™ of X/Open Company Limited. VERILOG™, PALLADIUM™ of Cadence Design Systems, Inc. VLYNQ™ of Texas Instruments Incorporated. VXWORKS™, WIND RIVER™ of WIND RIVER SYSTEMS, INC. ZETEX™ of Diodes Zetex Limited.

Last Trademarks Update 2011-02-24

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.

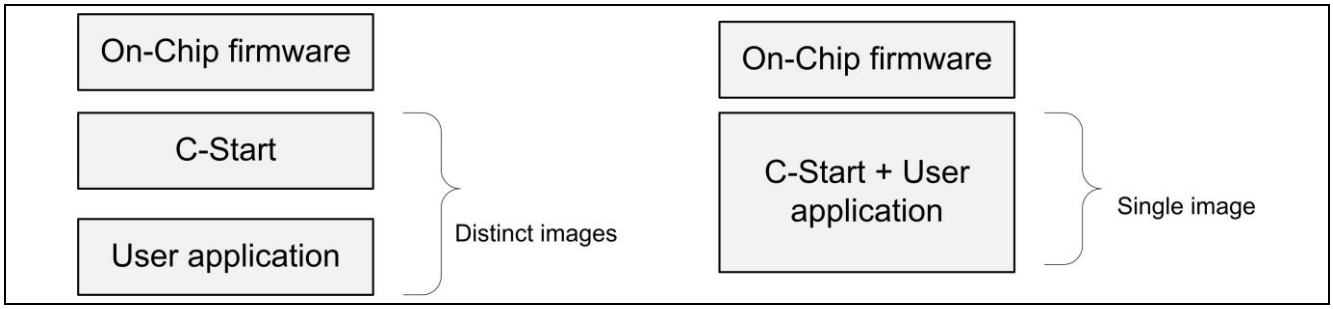

Figure 1 C-Start packaging

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 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 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.

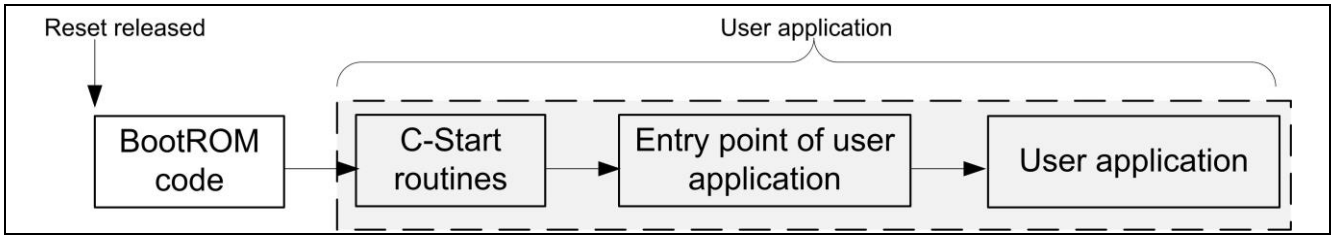

Figure 2 Booting stages

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

```
__Xmc1200_interrupt_vector_cortex_m:
…
.long   CLKVAL1_SSW /* Clock Configuration in SSW */
.long   CLKVAL2_SSW /* Peripheral Configuration in SSW */
```

CLKVAL1_SSW and CLKVAL2_SSW defined in the vector table points to the abovementioned Flash addresses respectively. The code fragment shows the definition of these two data.

```
#ifdef DAVE_CE
#include <XMC1200_SCU.inc>
#include "../Dave/Generated/inc/DAVESupport/Device_Data.h"
#else
#define CLKVAL1_SSW 0x80000000
#define CLKVAL2_SSW 0x80000000
#endif
```

CLKVAL1_SSW and CLKVAL2_SSW are both defined with values 8000000_H by default. These values can be changed directly at these definitions. These values will then be programmed to the respective addresses in Flash when the application code is downloaded to the device. If the user is developing a DAVE CE project, CLKVAL1_SSW and CLKVAL2_SSW will instead be defined in a separate file which is generated by DAVE3. The generated file will hold values according to the configurations made by the user in the app GUI.

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 Flash area. It then extracts the start address of the C-Start routines and cedes 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 of this table contains the start address of the reset handler.

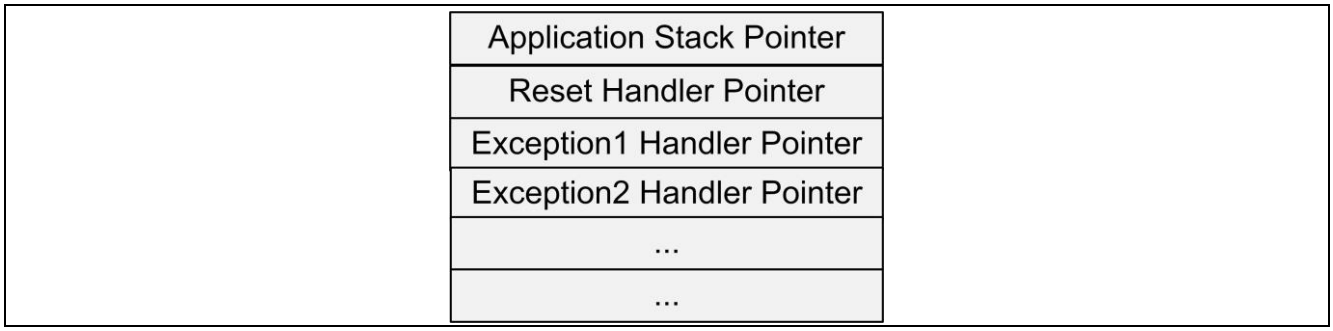

Figure 3 Cortex-M vector table

```
.syntax unified

.section ".Xmc1200.reset"
.globl  __Xmc1200_interrupt_vector_cortex_m
.type   __Xmc1200_interrupt_vector_cortex_m, %object

__Xmc1200_interrupt_vector_cortex_m:
.long   __Xmc1200_stack             /* Top of Stack */
.long   __Xmc1200_reset_cortex_m    /* Reset Handler */

Entry   NMI_Handler                 /* NMI Handler */
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 __Xmc1200_reset_cortex_m is the reset handler and 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

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

```
    .thumb_func
    .globl  __Xmc1200_reset_cortex_m
    .type   __Xmc1200_reset_cortex_m, %function
__Xmc1200_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 R0, =__Xmc1200_stack
    MOV SP, R0

    /* 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       __Xmc1200_Program_Loader

    .pool
    .cantunwind
    .fnend
    .size   __Xmc1200_reset_cortex_m,.-__Xmc1200_reset_cortex_m
```

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(), 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 ‘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 or LMA) to run address (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 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:-

```
   .section .Xmc1200.postreset,"x",%progbits
   __Xmc1200_Program_Loader:
   .fnstart
   /* Memories are accessible now*/

   /* DATA COPY */
   /* R0 = Start address, R1 = Destination address, R2 = Size */
   LDR R0, =DataLoadAddr   /* Obtained from linker script */
   LDR R1, =__Xmc1200_sData /* Obtained from linker script */
   LDR R2, =__Xmc1200_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
   ADDS R0,#4
   ADDS R1,#4
   B COPYLOOP

SKIPCOPY:
   /* BSS CLEAR */
   LDR R0, =__Xmc1200_sBSS /* Start of BSS obtained from linker script */
   LDR R1, = Xmc1200_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
   ADDS R0,#4
   B CLEARLOOP

SKIPCLEAR:
   /* VENEER COPY */
   /* R0 = Start address, R1 = Destination address, R2 = Size */
   LDR R0, =VeneerLoadAddr
   LDR R1, =VeneerStart
   LDR R2, =VeneerSize

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

   /* The proverbial loop from the schooldays */
VENEERCOPYLOOP:
   LDR R3,[R0]
   STR R3,[R1]
   SUBS R2,#1
   BEQ SKIPVENEERCOPY
   ADDS R0,#4
   ADDS R1,#4
   B VENEERCOPYLOOP
```

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 ceded to the user application.

The vector table is remapped to locations on 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
.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 with workarounds, and errata optionally executed,
- Application data has been relocated from LMA to VMA

The execution environment has now 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

   /* Reset stack pointer before zipping off to user application,
      Optional */
   LDR R0,=__Xmc1200_stack
   MOV SP,R0

   MOVS R0,#0
   MOVS R1,#0

   /* CEDE CONTROL TO ENTRY POINT OF USER APPLICATION */
   LDR R2,=main
   MOV PC,R2
```

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.

```
  .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 of interrupt handler definition 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 tool chain. 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

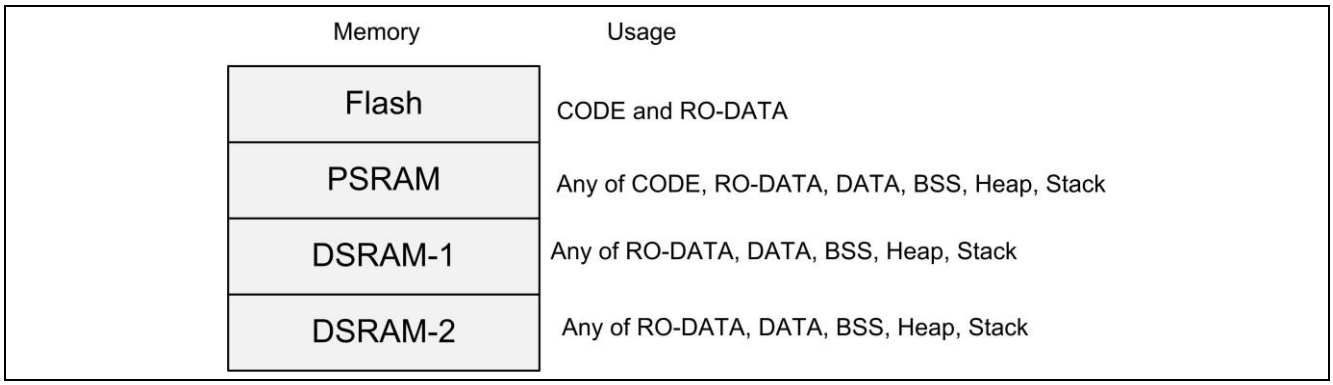

Figure 4 XMC memories and usage

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:-

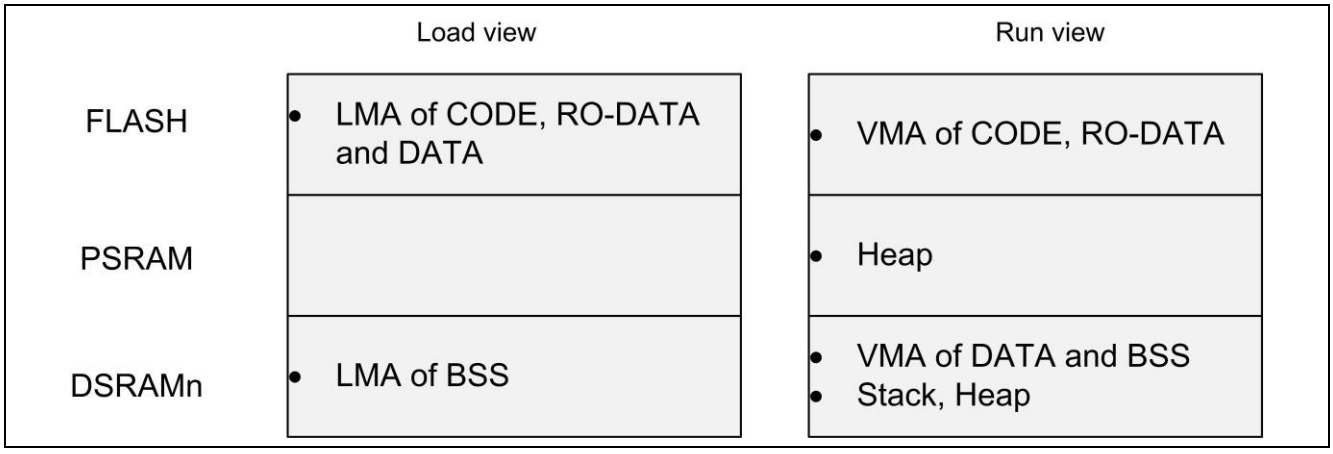

Figure 5 LMA and VMA concept

3.4 Typical program section assignment

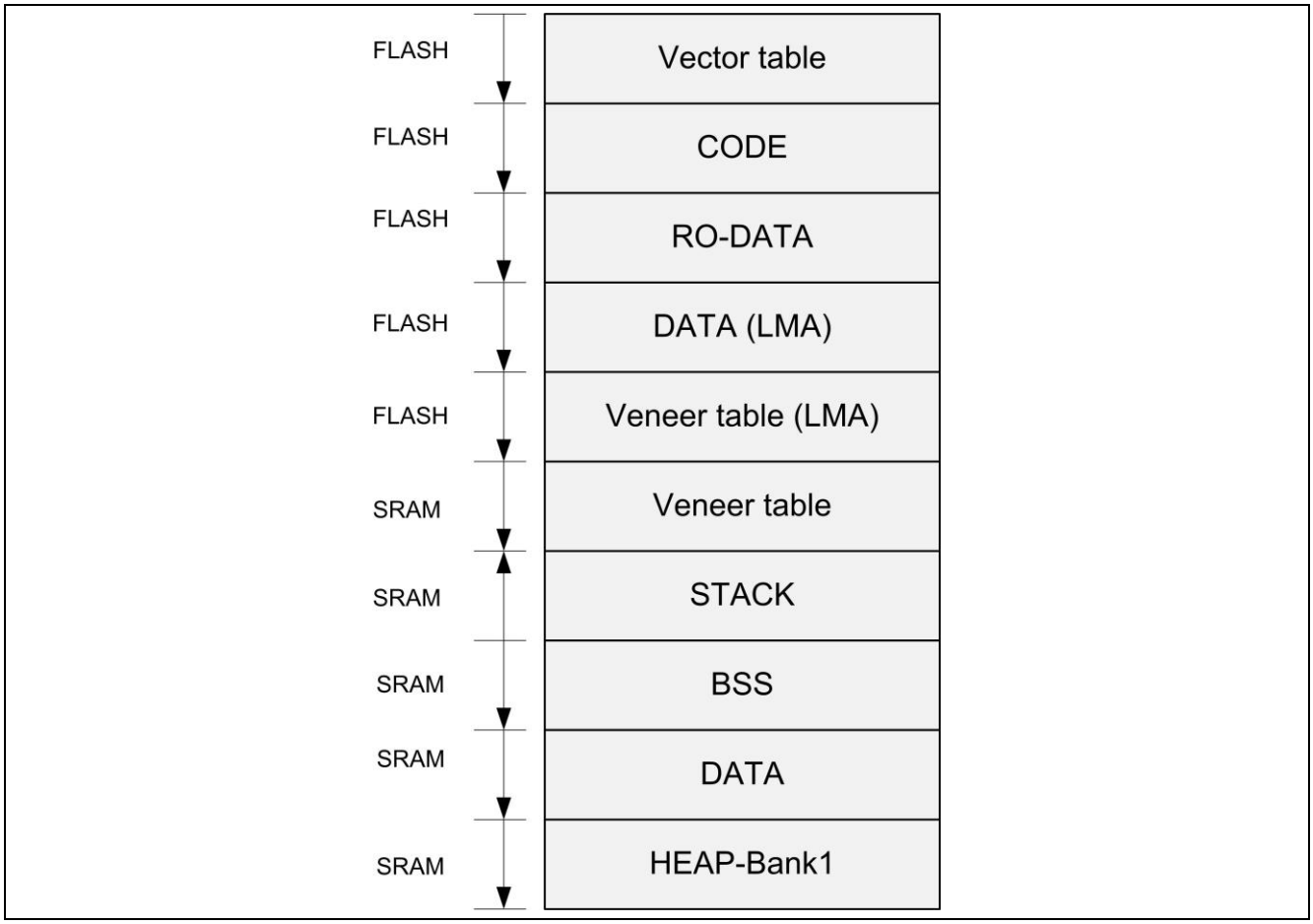

Figure 6 Typical program section placement on a XMC1 device

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(__Xmc1200_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(RX)     : ORIGIN = 0x10001000, LENGTH = 0x32000
  SRAM(!RX)     : ORIGIN = 0x20000000, LENGTH = 0x4000
}
```

```
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.
```

Examples

Example1: Positioning of startup code and standard .text

LMA = Flash, VMA = Flash

```
    .text : AT (ORIGIN(FLASH))
    {
        sText = .;
        *(.Xmc1200.reset);
        *(.Xmc1200.postreset);
        *(.XmcStartup);
        *(.text .text.* .gnu.linkonce.t.*);
  }>FLASH
```

Example2: Positioning of .data

LMA = Flash, , VMA = SRAM

```
    .data ABSOLUTE(ALIGN(16)): AT(eRODData)
    {
        __Xmc1200_sData = .;
        * (.data);
        * (.data*);
        *(*.data);
        *(.gnu.linkonce.d*)
        __Xmc1200_eData = ALIGN(4);
    } > SRAM
```

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 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 a 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

```
/* Define LMA of dedicated section */
sIRAMCodeLoad = eRODData + __Xmc1200_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 SRAM)
   . = ALIGN(4);
   eIRAMCode = ABSOLUTE(.);
  } > SRAM
  IRAMCodeSize      = eIRAMCode - ORIGIN(SRAM);
```

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

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

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 SRAM 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 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 */

 Startup : AT(ORIGIN(FLASH))
    {
     StartText = ABSOLUTE(.);
     *(.Xmc1200.reset);
     *(.Xmc1200.postreset);
     *(.XmcStartup);
     . = ALIGN(4);
     eStartup = ABSOLUTE(.);
    } > FLASH

Note that only the vector table and startup code are linked to the flash address
space. Standard .text section is separated out and linked to SRAM section below.
LMA of this section is in flash while the VMA is in SRAM.
     StartupSize = eStartup - StartText;
     sUserTextLoad = ORIGIN(FLASH)+ StartupSize;
.text ABSOLUTE(ORIGIN(SRAM)): AT(sUserTextLoad)
    {
      sUserTextRun = ABSOLUTE(.);
      *(.text .text.* .gnu.linkonce.t.*);
      . = ALIGN(4);
      eUserTextRun = ABSOLUTE(.);
   } > SRAM
      UserTextSize = eUserTextRun - sUserTextRun;
```

4.3.2 Example: Program loader modification

```
   /* Copy the TEXT from flash to SRAM */
   /* 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 SRAM */
   /* 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 SRAM */
   /* R0 = Start address, R1 = Destination address, R2 = Size */
   LDR R0, =DataLoad
   LDR R1, =__Xmc1200_sData
   LDR R2, =__Xmc1200_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 Device initialization hints

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
- 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

CLKVAL1_SSW and CLKVAL2_SSW are both defined with values 0x8000000 by default in the startup code. This means that SSW will ignore the values defined here and instead use the default values programmed in the SSW code. Values for CLKVAL1_SSW and CLKVAL2_SSW can be changed directly at the definition, like in the following example:-

```
#ifdef DAVE_CE
#include <XMC1200_SCU.inc>
#include "../Dave/Generated/inc/DAVESupport/Device_Data.h"
#else
#define CLKVAL1_SSW 0x00000200 /* PCLK = MCLK = 16MHz */
#define CLKVAL2_SSW 0x00000060 /* remove clock gating to LEDTS0 & LEDTS1 */
#endif
```

These values will be programmed to the respective addresses in Flash when the application code is downloaded to the device. If the user is developing a DAVE CE project, the values can be changed in the UI of the CLK002 app, as shown in the following figure.

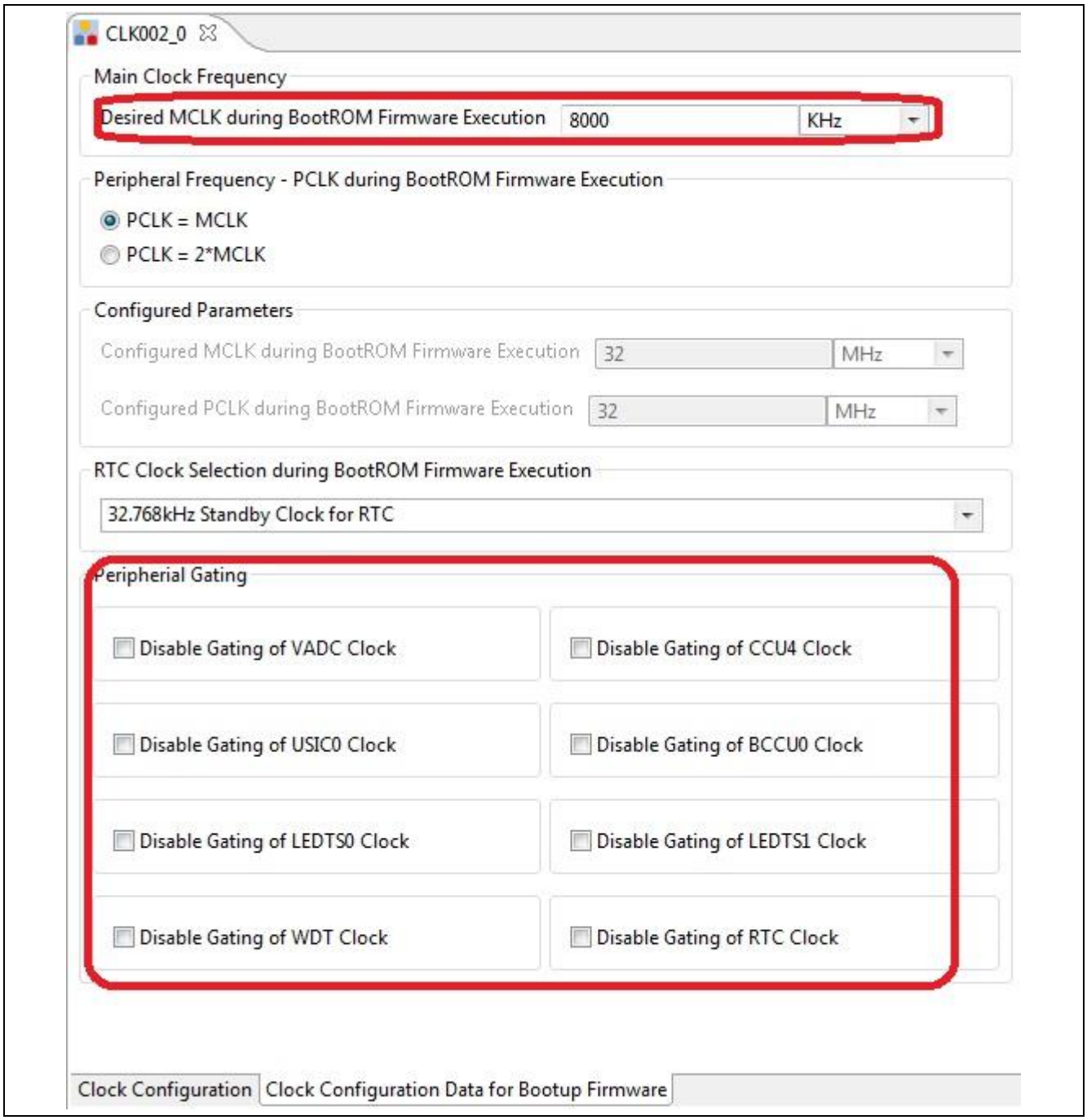

Figure 7 Changing CLKVAL1_SSW and CLKVAL2_SSW values with CLK002 app

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 also be cleared upon reading to ensure a clear status at the occurrence of the next reset.

Below is an example code:-

```
unsigned int status;
```

```
status = SCU_RESET->RSTSTAT & 0x000003FF; /* get the cause of reset */
SCU_RESET->RSTCLR = 1U; /* 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:

```
SCU_RESET->RSTCON = 0x00000003UL; /* enable ECC & loss of clock reset */
```

5.4 Configuring clock in user code

Clock for the XMC1000 family of 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 125kHz to 32MHz. 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.

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.

Here is an example code to configure clock:

```
/* CLKCR has protected bits. Open access */
SCU_GENERAL->PASSWORD = 0x000000C0UL;
/* CNTADJ = 1024 clock cycles, Standby clock as RTC clock source,
PCLK = 2*MCLK, MCLK = 32MHz */
SCU_CLK->CLKCR = 0x3FF10100UL;
while((SCU_CLK->CLKCR)&0x40000000UL); /* Wait for VDDP to stabilize */
SCU_GENERAL->PASSWORD = 0x000000C3UL; /* Close access to protected bits */
```

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. Status of a peripheral clock can be found out by reading 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 accordingly for the V_{DDP} to stabilize.

Below is an example code:-

```
SCU_GENERAL->PASSWORD = 0x000000C0UL; /* CLKCR has protected bits. Open access */
SCU_CLOCK->CLKGATSET0 = (1U << 9U); /* To enable gating to WDT clock */
while((SCU_CLK->CLKCR)&0x40000000UL); /* Wait for VDDP to stabilize */
SCU_CLOCK ->CLKGATCLR0 = (1U << 5U); /* To remove gating to LEDTS0 clock */
while((SCU_CLK->CLKCR)&0x40000000UL); /* Wait for VDDP to stabilize */
SCU_GENERAL->PASSWORD = 0x000000C3UL; /* Close access to protected bits */
```

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 get 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 an example code showing the initialization of the LEDTS0, CCU40 and USIC0:

```
/* LEDTS0 initialization instructions */
…
/* CCU40 initialization instructions */
…
/* USIC0 initialization instructions */
…
LEDTS0->GLOBCTL|=0x00800000; /* Start LEDTS-counter with CLK_PS=0x0080 */
SCU_GENERAL->CCUCON = 0x00000001; /* Enable CCU40 global start bit */
```

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 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 Managing interrupts

Modules generate events which potentially can 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.

5.8.1 Enabling of interrupts at CPU level

Interrupts can be enabled or disabled at CPU level with 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.

5.8.2 Enabling of interrupts at NVIC and module level

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.

```
LEDTS0->GLOBCTL |= 1U << 14; /* Enable time frame interrupt */
NVIC_SetPriority(29, 0); /* Assign a priority of 0 (highest) to Node-29 */
NVIC_EnableIRQ(29); /* Enable IRQ-29 */
```

5.8.3 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.

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

5.9 Putting it all together

The following example pieces all of the information together.

```
int main(void)
{
unsigned int status;

/* Check reset status and perform reset configuration */
status = SCU_RESET->RSTSTAT & 0x000003FF; /* get the cause of reset */
/* Perform necessary tasks here in case of a certain reset */
SCU_RESET->RSTCLR = 1U; /* clear status field */
SCU_RESET->RSTCON = 0x00000003UL; /* enable ECC and loss of clock reset */

/* System clock configuration */
SCU_GENERAL->PASSWORD = 0x000000C0UL; /* CLKCR has protected bits. Open access */
SCU_CLK->CLKCR = 0x3FF10100UL; /* CNTADJ = 1024 clock cycles, Standby clock as RTC
clock source, PCLK = 2*MCLK, MCLK = 32MHz */
while((SCU_CLK->CLKCR)&0x40000000UL); /* Wait for VDDP to stabilize */
SCU_GENERAL->PASSWORD = 0x000000C3UL; /* Close access to protected bits */

/* LEDTS0 initialization */
SCU_GENERAL->PASSWORD = 0x000000C0UL; /* CLKCR has protected bits. Open access */
SCU_CLOCK ->CLKGATCLR0 = (1U << 5U); /* To remove gating to LEDTS0 clock */
while((SCU_CLK->CLKCR)&0x40000000UL); /* Wait for VDDP to stabilize */
SCU_GENERAL->PASSWORD = 0x000000C3UL; /* Close access to protected bits */
/* 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 */
SCU_GENERAL->PASSWORD = 0x000000C0UL; /* CLKCR has protected bits. Open access */
SCU_CLOCK ->CLKGATCLR0 = (1U << 2U); /* To remove gating to CCU40 clock */
```

```
while((SCU_CLK->CLKCR)&0x40000000UL); /* Wait for VDDP to stabilize */
SCU_GENERAL->PASSWORD = 0x000000C3UL; /* Close access to protected bits */
/* 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 */
SCU_GENERAL->PASSWORD = 0x000000C0UL; /* CLKCR has protected bits. Open access */
SCU_CLOCK ->CLKGATCLR0 = (1U << 3U); /* To remove gating to USIC0 clock */
while((SCU_CLK->CLKCR)&0x40000000UL); /* Wait for VDDP to stabilize */
SCU_GENERAL->PASSWORD = 0x000000C3UL; /* Close access to protected bits */
/* 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 */

LEDTS0->GLOBCTL|=0x00800000; /* Start LEDTS-counter with CLK_PS=0x0080 */
SCU_GENERAL->CCUCON = 0x00000001; /* Enable CCU40 global start bit */

/* 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*/
}
```

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

Published by Infineon Technologies AG