Software Update of XMC1000 Microcontrollers Using an ASC Interface

XMC1000

About this document

Scope and purpose

This Application Note describes how to update the application program in XMC1000 device flash using a Host PC. The communication interface to Host PC is determined by application, and could be ASC, SPI and USB. The DAVE example projects are provided with Application Note to demonstrate how to implement the remotely controlled flash update system. In the demo codes ASC (UART) interface is used to communicate with PC. The applicable products are XMC1000 Microcontrollers family. The example codes are tested on XMC1300 Boot Kit.

Intended audience

This application note is intended to customers who want to develop a remote control system to update the flash codes on XMC1000 microcontroller family, like FW (firmware) update and application code update.

Table of Contents

About this document **1**
Table of Contents **2**
1 Introduction **3**
1.1 Tool-chains 4
1.2 Example Programs 4
2 Concept of Demonstrator **5**
2.1 Flash Partitioning 5
2.2 Bootstrap Program 6
2.2.1 Modification of Linker Description (ld) File 7
2.3 Application Programs 8
2.3.1 Example of Application Programs 9
2.3.2 Modifications in Linker Description (ld) File 9
2.3.3 Copy of Flash Loader Program to SRAM 10
3 Flash Loader Program **12**
3.1 Initialization of ASC Module 12
3.2 Flash Loader Procedure 12
3.3 Example of Flash Loader Program 13
3.4 DAVE4 Project Settings 13
3.4.1 Modification of DAVETH linker descript (ld) File 14
3.4.2 Modification of DAVETH startup.s File 14
3.5 Flash Memory Organization 14
4 HOST PC Program Example **17**
4.1 Communication Protocol 20
4.1.1 Mode 0: Program Flash Page 21
4.1.2 Mode 1: Erase Flash Sector 22
4.2 Response Code to the HOST 22
5 Usage of Demonstrator **23**
5.1 Hardware Setup 23
5.2 Demonstrator File Structure 23
5.3 Run the Demonstrator 24
6 Reference Documents **28**
Revision History **29**

1 Introduction

This application note describes how to update the user software (Application Program) residing in the Flash memory of the microcontroller through a connected PC Host. The software update happens during normal operation of the microcontroller (during run-time). The used communication interface between the microcontroller and the PC Host is ASC (UART) using a USB VCOM CDC adapter. The used concept is applicable to other communication options like SPI, USB, Ethernet, etc.

For safety reason the concept used in this application note assumes the existence of two full independent Application Programs, Application Program 1 and Application Program 2. A Bootstrap Program finds out which of the two Application Programs is the most recent version and executes it. In case of a update, the currently not used Application Program is erased and replaced by the update. This concept ensures that in case of data or power loss during the update no harm happens as the currently used Application Program is not touched.

The update procedure is managed by the Flash Loader that is stored in flash. Upon update request from the PC Host the Flash Loader is copied from flash into the SRAM (by the currently used Application Program). After copying is finished, the Flash Loader takes over the control. It communicates via a USIC interface (UART module) with the PC according to a defined protocol, obtains the hex file, erases and programs the Flash. The hex file transferred from the PC host to the microcontroller is not encrypted. After successful update of the Application Program the Flash Loader issues a software reset. The Boot Loader will then start the new Application Program.

In order to complete the software update we need following types of programs:

- Bootstrap Program:
  Program that is executed after reset and that determines which of the 2 Application Program should be executed afterwards
- Flash Loader:
  Program that includes Flash driver routines to erase and program the Flash plus the communication and protocol routines to connect and communicate with the PC Host
- Application Program:
  Program that provides the intended application uses case running on the MCU.
- PC Host:
  PC that is used to connect to the MCU to update the MCU software and the Application Program

The SW package that is provided with this application note is structured in following separate projects. The projects are developed with DAVE v4 for XMC 1300:

1. Boot Loader project
   Includes all the required adaptions of the linker script
2. Flash Loader project
   Includes the required adaptions of the linker script and the startup files
3. Project for the Application Program 1
   Simple use case for demonstration purpose (Blinky), it includes all required linker script adaptations
4. Project for the Application Program 2
   Simple use case for demonstration purpose (another Blinky), it includes all required linker script adaptations

MS Visual C++ projects for the PC Host

- PC program to communicate with the MCU target via a USB CDC VCOM channel

All projects are fully tested and ready to use and explore.

1.1 Tool-chains

The demo programs for XMC1000 device are developed with the following tool-chain:

- DAV4 development platform v4.2.6

1.2 Example Programs

The HOST PC program is developed with Microsoft Visual C++ 2010. The example source codes are found in the following folders:

- .\ SRAMCode, contains the Flash Loader developed using DAVE GCC compiler.
- .\ Bootstrap, contains the bootstrap program developed using DAVE GCC compiler.
- .\ Application1\Blinky1, contains the Application program 1 developed using DAVE GCC compiler.
- .\ Application1\Blinky2, contains the Application program 2 developed using DAVE GCC compiler.
- .\XMC1x_Load\, holds the example HOST PC program that demonstrates the whole process of flash update using a Host PC. The project files can be compiled with Microsoft Visual C++2010.

Chapter 5 describes in detail how to use the example program to download your own program into Flash and run it.

2 Concept of Demonstrator

2.1 Flash Partitioning

Flash partitioning is the first step needed to be done for the implementation of flash update using a Host PC, because the flash sections for the different program storage must be defined before starting to write the application codes. In this application note we use the flash partition in Figure 1.

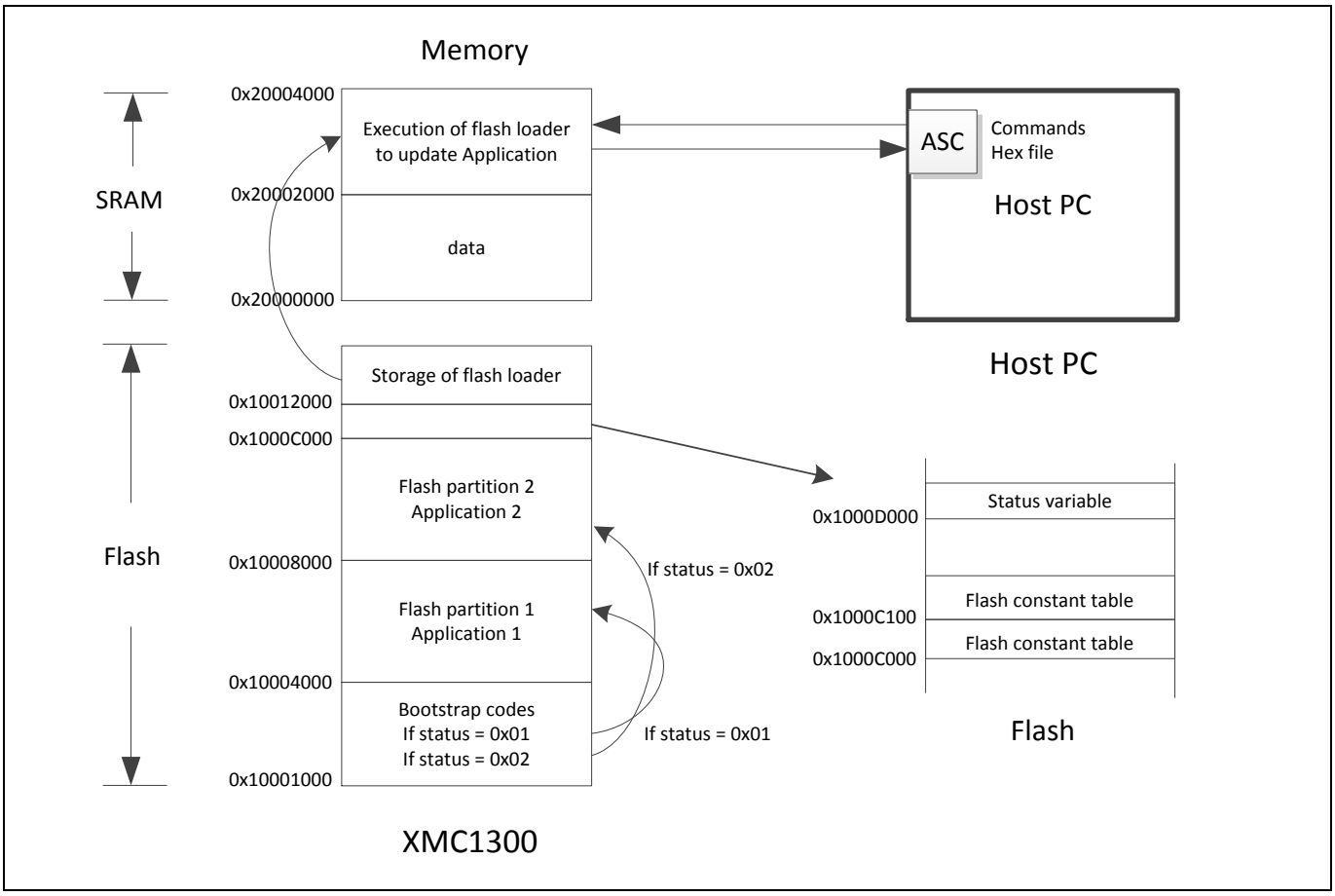


Figure 1 Flash partition of demonstration for flash update using a Host PC

In the Figure 1 we partition the flash into 4 parts:

- Bootstrap codes: 0x10001000 – 0x10003FFF (12 KB)
- Application 1: 0x10004000 – 0x10007FFF (16 KB)
- Application 2: 0x10008000 – 0x1000CFFF (16KB)
- Storage for flash loader: 0x10012000 – 0x10013FFF (8 KB)

Flash loader is just stored in flash. If the flash update operation is required, the flash loader codes are first copied from flash to SRAM from 0x20002000 to 0x20004000, then execute from 0x20002000. In this case, the first part of SRAM (0x20000000 – 0x20001FFF) is used for data.

2.2 Bootstrap Program

Bootstrap program is the first executed program after reset. The program executes from 0x10001000, and is responsible for management of application programs. Figure 2 shows the flow of bootstrap program.

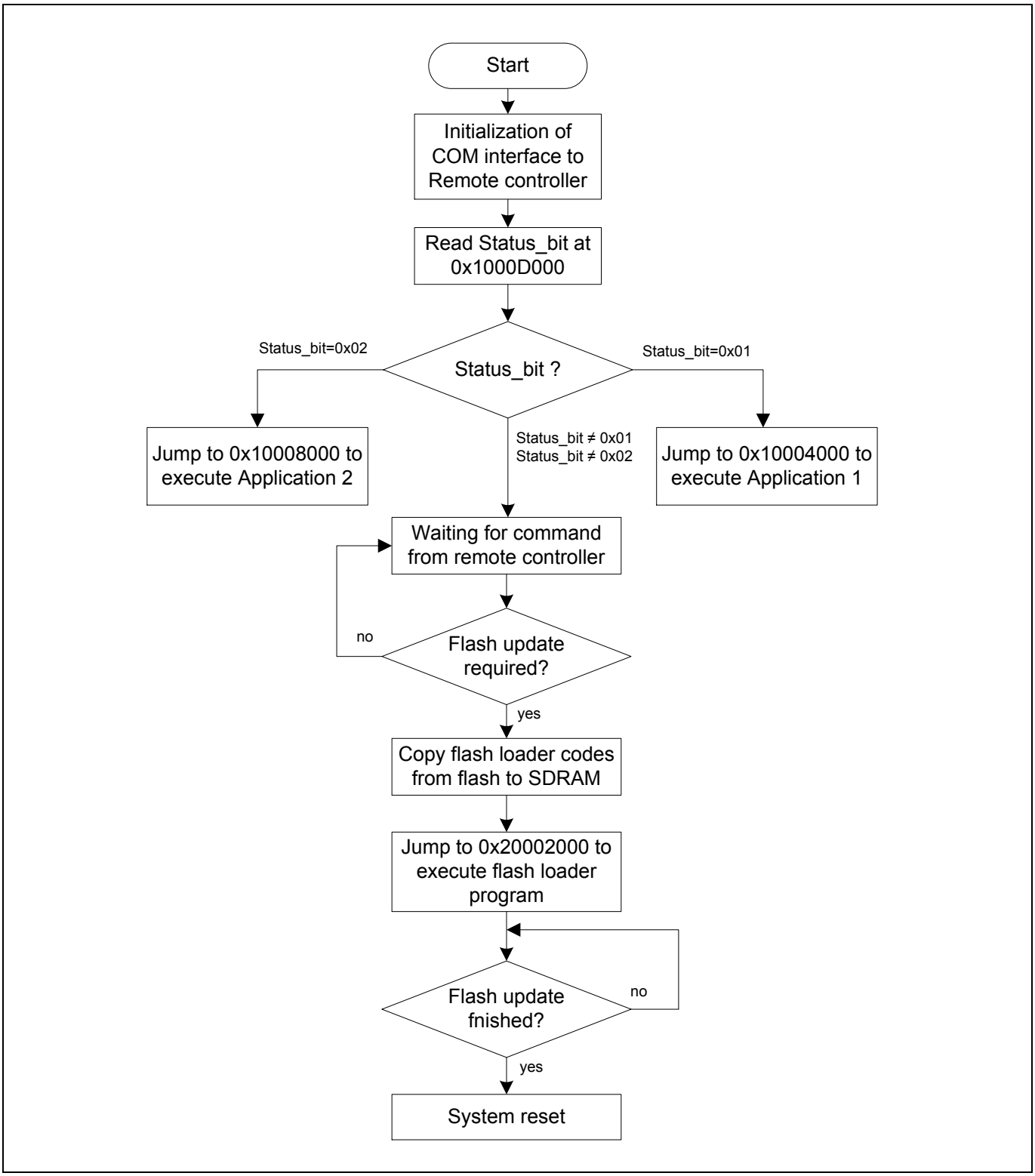


Figure 2 Program flow of bootstrap codes

At the start of bootstrap program the communication (COM) interface to PC must be initialized and is ready to communicate with the PC. If the ASC (UART) interface is used for communication with PC, the used ASC module in XMC1000 device must be initialized. After initialization the status bit saved at 0x1000D000 will be verified. If the status bit is equal to 0x01, this means that the Applicaton1 will be executed now. Then the program counter will be set with 0x10004000, where the Application 1 is located and stored. The program jumps to Application1 to execute. If the status bit is equal to 0x02, the program jumps to Application 2 located at 0x10008000 to execute.

If the status bit is neither 0x01 nor 0x02, the program is waiting for a command from PC. As soon as the flash update command is received on the device side, the flash loader codes stored in flash will be copied to SRAM and executed from 0x20002000. The flash loader program takes over control of XMC1000 device and communicates with PC to finish the application update in the flash. After the application codes are completely updated, the status bit at 0x1000D000 will be also updated to indicate the new updated application. Finally a system rest will be done to reset the system.

2.2.1 Modification of Linker Description (ld) File

Two important flash sector tables are defined in Bootstrap project. The sector tables are used for flash programming for applications. Table “XMC1000_FLASH1_SectorTable” contains the flash areas from 0x10004000 to 0x10008000, while the table “XMC1000_FLASH2_SectorTable” contains the flash areas from 0x10008000 to 0x1000D000. These two tables are used in flash loader running from SRAM, but saved in flash at 0x1000C000 and 0x1000C100. To locate the constant table in a dedicated address we need to define a special section in linker description file like:

```
  IRAM_Code_1 : AT (0x1000C000)
  {
   sIRAMCode = ABSOLUTE(0x1000C000);
   KEEP(* (.IRAMCode1)) ;
   . = ALIGN(4);
   eIRAMCode        = ABSOLUTE(0x1000C000);
  } > FLASH_1

  IRAM_Code_2 : AT (0x1000C100)
  {
   sIRAMCode = ABSOLUTE(0x1000C100);
   KEEP(* (.IRAMCode2)) ;
   . = ALIGN(4);
   eIRAMCode        = ABSOLUTE(0x1000C100);

  } > FLASH_2
```

where FLASH_1 and FLASH_2 are defined in Memory:

```
MEMORY
{
    FLASH(RX) :   ORIGIN = 0x10001000, LENGTH = 0x3000
    FLASH_1(RX) : ORIGIN = 0x1000C000, LENGTH = 0x100
    FLASH_2(RX) : ORIGIN = 0x1000C100, LENGTH = 0x100
    SRAM(!RX)   : ORIGIN = 0x20000000, LENGTH = 0x4000
}
```

In the main function we can define

```
extern const TSectorTableEntry XMC1000_FLASH1_SectorTable[] __attribute__ ((section
(".IRAMCode1")))  ;

extern const TSectorTableEntry XMC1000_FLASH2_SectorTable[] __attribute__ ((section
(".IRAMCode2")))  ;
```

With above modification in ld file and the definition in main function the both tables are located in the dedicated address.

2.3 Application Programs

Application program is user specific, which is developed according to the application requirements. But, in the application codes the initialization of communication interface is mandatory to enable the PC to access the device during the running of application codes. To not disturb the normal execution of application programs the interrupt is used to access the device by PC. Figure 3 shows the major program flow of application codes.

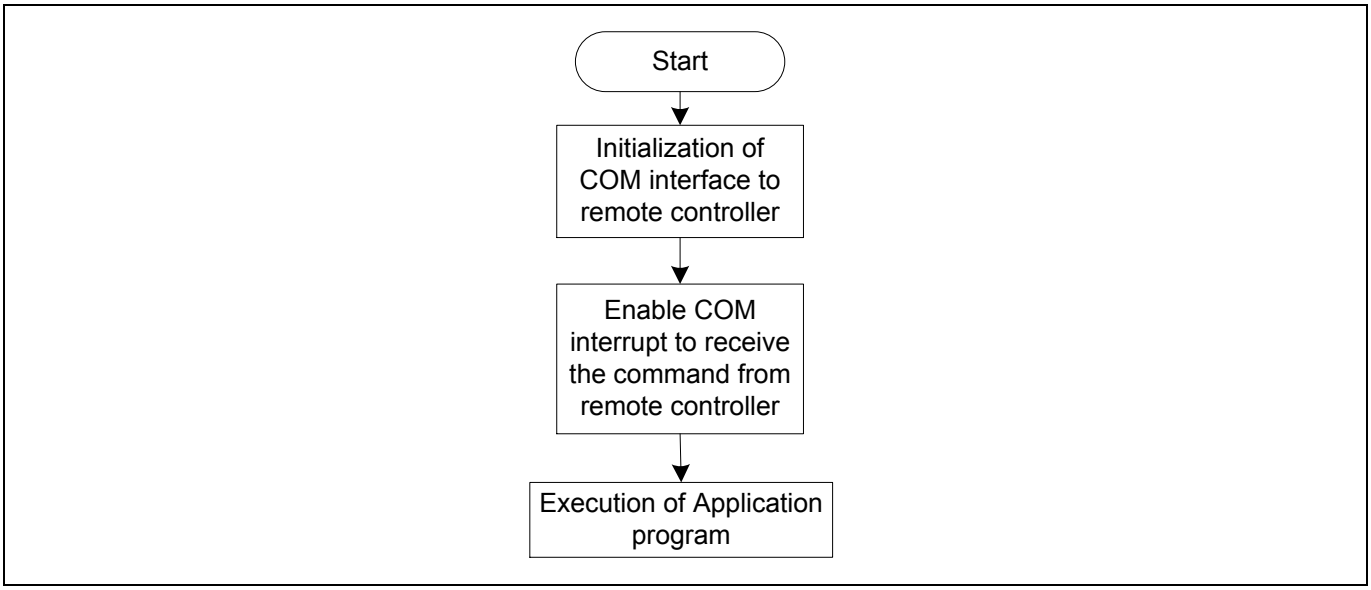


Figure 3 Major program flow of Application codes

In the interrupt handler routine, two operations are done. Firstly all interrupts should be disabled. Secondly the flash loader codes are copied from flash to SRAM and executed here to complete the flash update operation. The general flow of interrupt handler is showed on Figure 4.

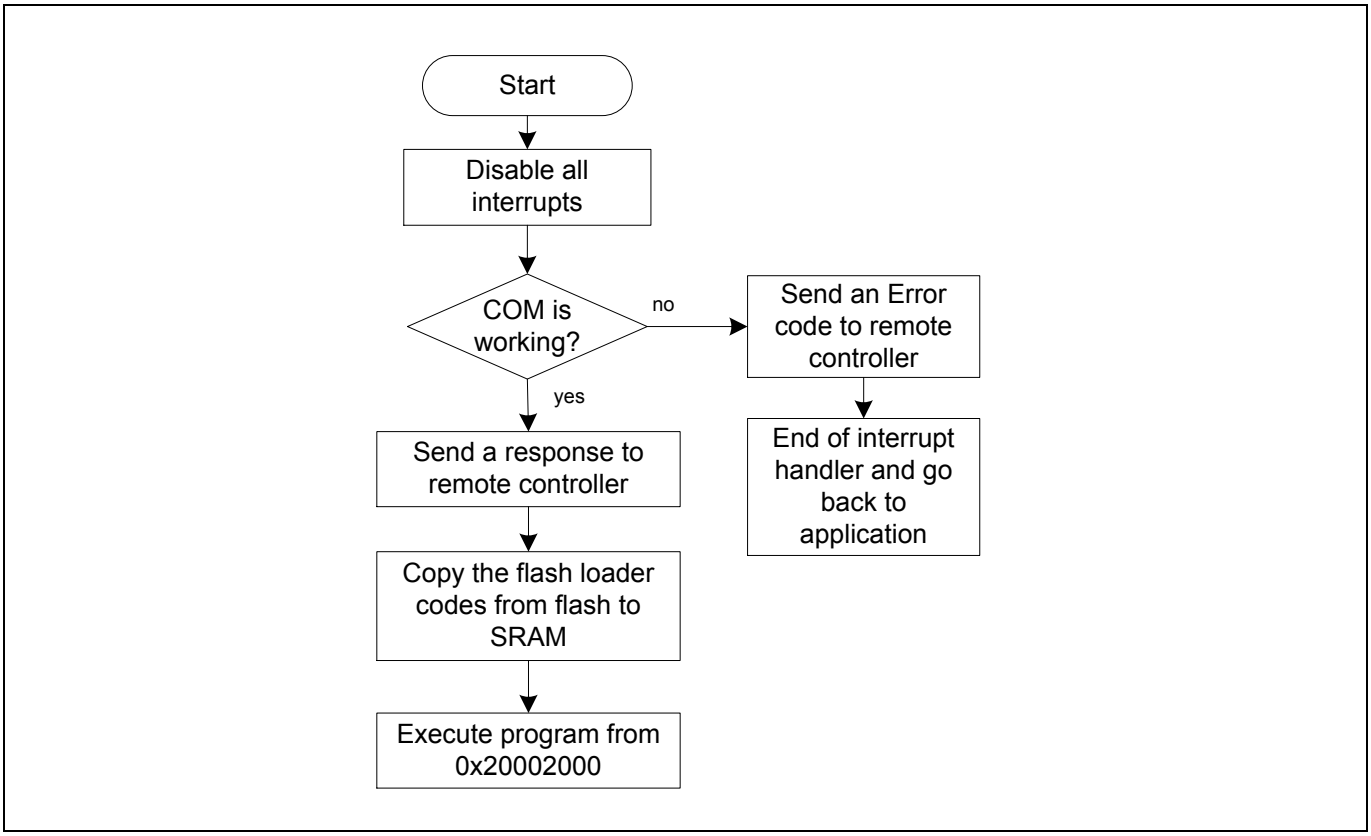


Figure 4 Program flow of interrupt handler in Application codes

2.3.1 Example of Application Programs

The demo codes in this AppNote contain two application programs, where a system timer is used to toggle the LEDs on XMC1300 boot kit that are connected with ports P0.0, P0.1, P0.6, P0.7, P0.8 and P0.9 with a frequency depending on the analog value of P2.5. The LEDs that are connected to these ports will toggle respectively. The difference is that Application 1 toggles all 6 LEDs, while Application 2 just toggles the last three LEDs. To make sure that the application codes can be located in the defined flash partition and the flash update is working, the linker description (ld) file in application project needs to be modified. Furthermore, a SRAM copy routine must be also included in the interrupt handler.

2.3.2 Modifications in Linker Description (ld) File

The programs are developed with DAVE4 v4.2.6. To make the applications to be located in different partitions, the memories in linker descript file (linker_script.ld) needs to be changed like:

For Application 1:

```
MEMORY
{
    FLASH(RX) : ORIGIN = 0x10004000, LENGTH = 0x4000
    SRAM(!RX) : ORIGIN = 0x20000000, LENGTH = 0x4000

}
```

where flash starts at 0x10004000 with length of 0x4000. The original linker descript file defines the flash to start from 0x10001000. The start address is the flash address where the application will be located and executed. This address can be changed according to application.

For Application 2:

```
MEMORY
{
    FLASH(RX) : ORIGIN = 0x10008000, LENGTH = 0x4000
    SRAM(!RX) : ORIGIN = 0x20000000, LENGTH = 0x4000

}
```

where flash starts at 0x10008000 with length of 0x4000.

Note: if we want to change the start address and the size of applications, we need to modify the flash partition and the sector tables of XMC1000_FLASH1_SectorTable and XMC1000_FLASH2_SectorTable in Device_Memroy.h. For example, if the flash partition1 is extended to 32Kb, XMC1000_FLASH1_SectorTable is given by:

```
const TSectorTableEntry XMC1000_FLASH1_SectorTable[] =
{
        {0x10004000, 0x1000},    /*4 Kb*/
        {0x10005000, 0x1000},
        {0x10006000, 0x1000},
        {0x10007000, 0x1000},
        {0x10008000, 0x1000},
        {0x10009000, 0x1000},
        {0x1000A000, 0x1000},
        {0x1000B000, 0x1000},
        {0,0}
};
```

2.3.3 Copy of Flash Loader Program to SRAM

Here is an example of interrupt handler routine that copies the flash loader program from flash to SRAM to execute. The flash loader program is stored at 0x100012000. The codes will be copied to SRAM at 0x20002000 to execute.

```
unsigned char* RamAddr = (unsigned char *) (0x20002000);//SRAM address
unsigned char* FlasAddrSys = (unsigned char *) (0x10012000);//flash loader code

/* USIC0 Interrupt Handler */
void USIC0_0_IRQHandler(void)
{
    int i;

    SysTick->CTRL  &= 0xFFFFFFFC;  // disable system timer imterrupt

    // setup communication with PC
    i = (USIC0_CH1->OUTR & 0xFF);
    if (i == 0x5D)
    {
        SendByte(0x5D);  // communication is OK, send 0x5D back to to PC
        NVIC_DisableIRQ(USIC0_0_IRQn);  //disable interrupt

        for (i=0; i<TABLE_SIZE; i++)  //copy the codes from Flash to SRAM
         {
           *RamAddr = *FlasAddrSys;
```

```
            RamAddr++;
            FlasAddrSys++;
          }
          RunRAM(); // jump to SRAM to execute the flash loader program

      }
      else
           SendByte(BSL_MODE_ERROR);

}

void RunRAM(void)
{
  __asm
  (
    "LDR r0, =0x20002001;"    // execute program from SRAM; at 0y20002000
    "BLX r0;"
  );
}
```

Note: Cortex-M0 has thumb instruction with 16 bits width. So, the 0x2000200**1** should be loaded to register R0 instead of 0x20002000.

3 Flash Loader Program

In the demo of this application note we use ASC (UART) on the XMC1000 device to communicate with PC to complete the flash update. The mechanism of failure handling is built in the flash loader program. If the flash programming is failed due to power loss or break of communication between PC and device, the previous application codes are still the default code and the application system is not impacted. A new download process can be restarted again.

3.1 Initialization of ASC Module

The communication between PC and the target device is established via the ASC interface. Figure 5 shows a hardware setup for this application. On the target device side, the channel 1 of USIC0 (U0C1) is used as ASC. Ports P1.3 and P1.2 are used as RxD and TxD, respectively.

- receive pin RxD at pin P1.3 (USIC0_CH1.DX0A)
- transmit pin TxD at pin P1.2 (USIC0_CH1.DOUT0)

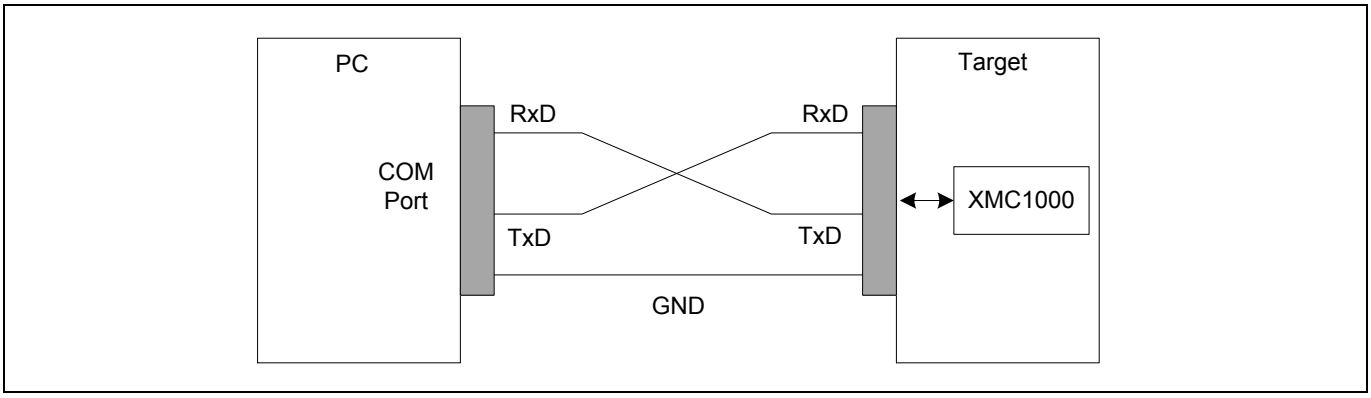


Figure 5 Connection between PC and XMC1000

The ASC interface must be initialized at the beginning of main program in both bootstrap and application codes in order for the device to be ready to communicate with PC. Note that we do not use interrupt in bootstrap program to receive the message from PC. The interrupt handler is just used in application programs.

3.2 Flash Loader Procedure

The flash loader procedure is shown below in Figure 6. In this example we configure the ASC with a fixed baudrate of 19200 bit/s. Before entering the flash loader the ASC communication is already set up, which is completed in the application codes. In the flash loader program the HOST starts by transmitting a header data block to device to tell device what should be done. In order to send the data from PC to device we have defined different header data blocks. The communication protocols are described in Chapter 4 of application note. If the header data is correctly received by device, the flash loader responds with a data of 0x55 to represent the successful receive. At the same time the data block is evaluated by flash loader program to check which command is sent from PC. In case of flash erase command the flash erase routine will be called to execute flash erase operation. If the program flash operation is required, the flash programming routine is called. After flash programming is completed, the flash loader sends a data of 0x55 to Host PC to indicate the successful flash programming, and updates the status bit of application program. Before leaving the flash loader program a system reset will be done. Then, the device starts again from ROM codes. The application program being just updated will be executed after system reset.

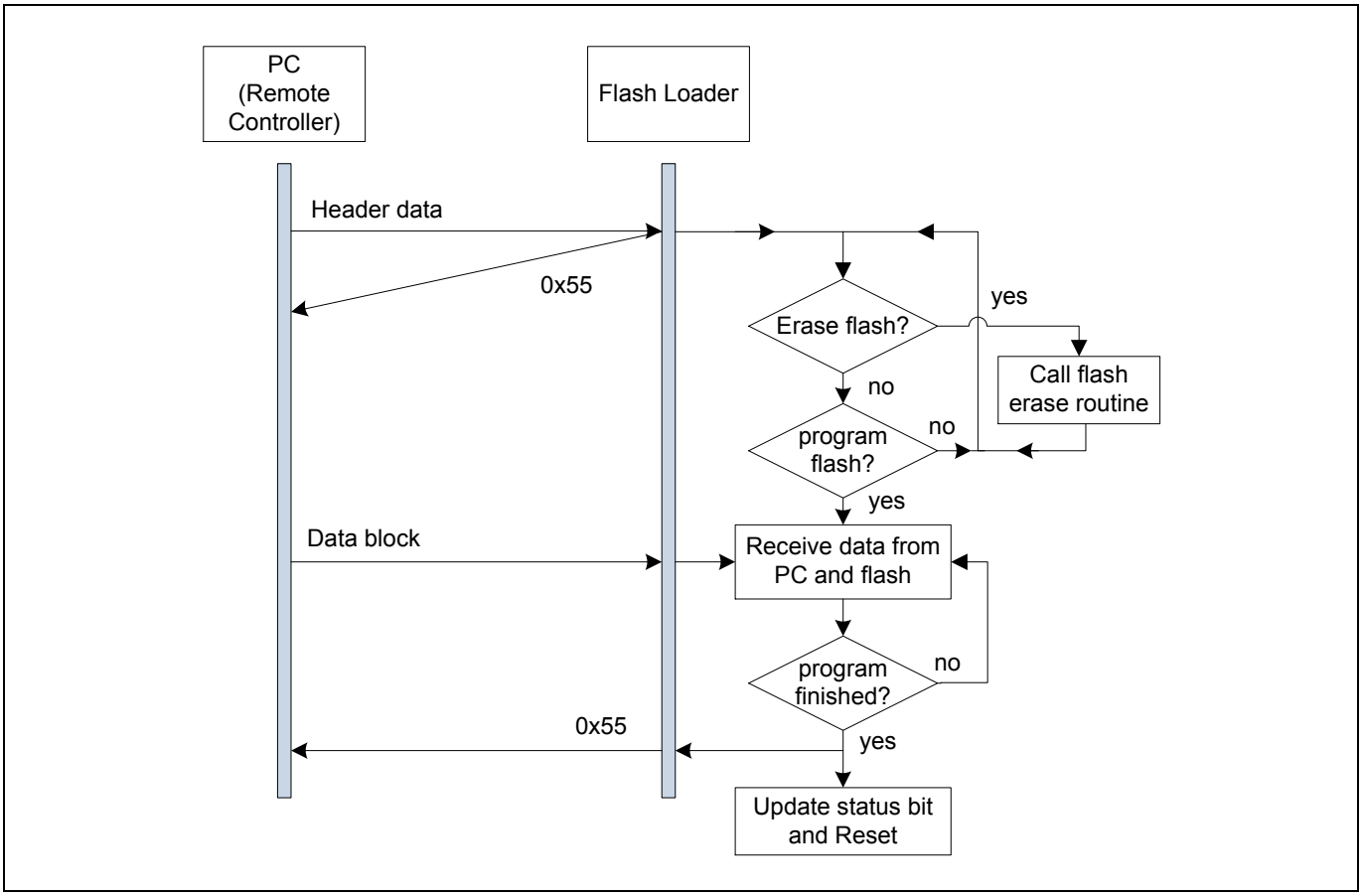


Figure 6 Flash loader procedure for Flash programming

3.3 Example of Flash Loader Program

An example of flash loader program developed with DAVE is provided in this application note. The Flash Loader implements the Flash routines and establishes the communication between PC and the target device. Flash loader routines provide the following features:

- Erase Flash sectors
- Erase, Program and verify the programmed Flash pages

If the communication module with PC is fixed, the flash loader program can be reused for all applications and independent from application codes. Below we give the DAVE4 project settings.

3.4 DAVE4 Project Settings

The Flash loader DAVE™ project is available in the .\SRAMCode\ folder. The project can be imported into the DAVE™ IDE with the following steps:

- Open the DAVE™ IDE
- Import the Infineon DAVE™ project
- Select root directory as .\SRAMCode
- Finish the import

Note: the DAVE generated hex file of flash loader is located at 0x20002000. In order to store the hex file in flash, we need to modify the first line of hex file to the flash address.

3.4.1 Modification of DAVE™ linker descript (ld) File

The Flash Loader program must be located in the SRAM starting at 0x20002000, because the Flash Loader program can only run from SRAM. Therefore the default linker script file generated from DAVE4 cannot be used in the Flash loader project, because the default linker script file locates the codes in Flash starting at 0x10001000. The linker script file that locates the codes into SRAM is provided in linker_script.ld. In comparison with the default ld file the changes are the memory definition:

```
MEMORY
{
    FLASH(RX) : ORIGIN = 0x20002000, LENGTH = 0x2000
    SRAM(!RX) : ORIGIN = 0x20000000, LENGTH = 0x2000
}
```

Here we keep using the names of “Flash” and “SRAM” in the memory definition in order to avoid changes of rest part of ld file. But, all memories are in SRAM. We divide the SRAM into two parts, one for codes (0x20002000-0x20003FFF) and another for data (0x20000000-0x20001FFF). The XMC1000 memory organization is described in the “Flash Memory Organization”.

3.4.2 Modification of DAVE™ startup.s File

It is important to note that all clock setting functions in the startup_XMC1x00.S file used in SRAMCode project, must be removed so that the clock settings made in the Application programs can be kept without modification. Otherwise, the communication with PC will be broken. For example, the following instructions in the DAVE™ startup_XMC1300.S file **must be removed**:

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

ldr  r0, =SystemInit
blx  r0
```

These instructions must be removed because the functions SystemInit() will change the clock settings, which will change the ASC baud rate and destroy the ASC communication between the Host PC and board after control handover from Application code to the flash loader program. If the baud rate is changed, the ASC communication between PC and board will be broken and the Flash programming will not work anymore. Furthermore, interrupt veneer is already in SRAM and needs not to be copied again to SRAM.

The startup.S files provided in the SRAMCode project has been modified and the system init functions are removed.

3.5 Flash Memory Organization

The embedded Flash module in the XMC1000 family includes 200 KB (maximum) of Flash memory for code or constant data.

Flash memory is characterized by its sector architecture and page structure. The offset address of each sector is relative to the base address of its bank which is given in Table 1. Some device types (see the XMC1000 Data Sheet) can have less Flash memory. For such devices, the higher numbered physical sectors are not available.

Table 1 Flash Memory Map

Range Description	Size	Start Address
Program Flash	200 Kbytes	0x10001000

- Flash erasure is sector-wise.
- Sectors are subdivided into pages.
- Flash memory programming is page-wise.
- A Flash page contains 256 bytes.
- Table 2 lists the logical sector structure in the XMC1000 family of products.

Table 2 Sector Structure of XMC1000 Flash

Sector	Address Range	Size
1	0x10001000 – 0x10001FFF	4 KB
2	0x10002000 – 0x10002FFF	4 KB
3	0x10003000 – 0x10003FFF	4 KB
4	0x10004000 – 0x10004FFF	4 KB
5	0x10005000 – 0x10005FFF	4 KB
6	0x10006000 – 0x10006FFF	4 KB
7	0x10007000 – 0x10007FFF	4 KB
8	0x10008000 – 0x10008FFF	4 KB
9	0x10009000 – 0x10009FFF	4 KB
10	0x1000A000 – 0x1000AFFF	4 KB
11	0x1000B000 – 0x1000BFFF	4 KB
12	0x1000C000 – 0x1000CFFF	4 KB
13	0x1000D000 – 0x1000DFFF	4 KB
14	0x1000E000 – 0x1000EFFF	4 KB
15	0x1000F000 – 0x1000FFFF	4 KB
16	0x10010000 – 0x10010FFF	4 KB
17	0x10011000 – 0x10011FFF	4 KB
18	0x10012000 – 0x10012FFF	4 KB
19	0x10013000 – 0x10013FFF	4 KB
20	0x10014000 – 0x10014FFF	4 KB
21	0x10015000 – 0x10015FFF	4 KB
22	0x10016000 – 0x10016FFF	4 KB
23	0x10017000 – 0x10017FFF	4 KB
24	0x10018000 – 0x10018FFF	4 KB
25	0x10019000 – 0x10019FFF	4 KB
26	0x1001A000 – 0x1001AFFF	4 KB
27	0x1001B000 – 0x1001BFFF	4 KB
28	0x1001C000 – 0x1001CFFF	4 KB

Sector	Address Range	Size
29	0x1001D000 – 0x1001DFFF	4 KB
30	0x1001E000 – 0x1001EFFF	4 KB
31	0x1001F000 – 0x1001FFFF	4 KB
32	0x10020000 – 0x10020FFF	4 KB
33	0x10021000 – 0x10021FFF	4 KB
34	0x10022000 – 0x10022FFF	4 KB
35	0x10023000 – 0x10023FFF	4 KB
36	0x10024000 – 0x10024FFF	4 KB
37	0x10025000 – 0x10025FFF	4 KB
38	0x10026000 – 0x10026FFF	4 KB
39	0x10027000 – 0x10027FFF	4 KB
40	0x10028000 – 0x10028FFF	4 KB
41	0x10029000 – 0x10029FFF	4 KB
42	0x1002A000 – 0x1002AFFF	4 KB
43	0x1002B000 – 0x1002BFFF	4 KB
44	0x1002C000 – 0x1002CFFF	4 KB
45	0x1002D000 – 0x1002DFFF	4 KB
46	0x1002E000 – 0x1002EFFF	4 KB
47	0x1002F000 – 0x1002FFFF	4 KB
48	0x10030000 – 0x10031FFF	4 KB
49	0x10032000 – 0x10032FFF	4 KB

4 HOST PC Program Example

The XMC1000 Host PC program is developed in C++. The file **XMC1x_load_API.cpp** contains the API for direct communication with the flash loader. The API includes the functions listed in Table 3:

Table 3 API Functions

API Function	Description
Init_uart	Initialize PC COM interface
bl_send_header	Send header block via ASC interface
bl_send_data	Send data block via ASC interface
bl_send_EOT	Send EOT block via ASC interface
bl_erase_flash	Erase Flash sectors
bl_download_flash	Download code to Flash
Make_flash_image	Create a Flash image from HEX file

The main program (XMC1x_Load.cpp) initializes ASC and sends Application Hex file to the target device.

The user must specify the HEX file to be downloaded. Two example HEX files (blinky1.hex, blinky2.hex) are provided. The application code is first downloaded to Flash and the Bootstrap program decides which application will be executed after reset based on the application status bit.

- The Flash erase procedure is implemented in the function bl_erase_flash() shown in Figure 7.
- The Flash programming procedure is implemented in bl_download_flash() shown in Figure 8.

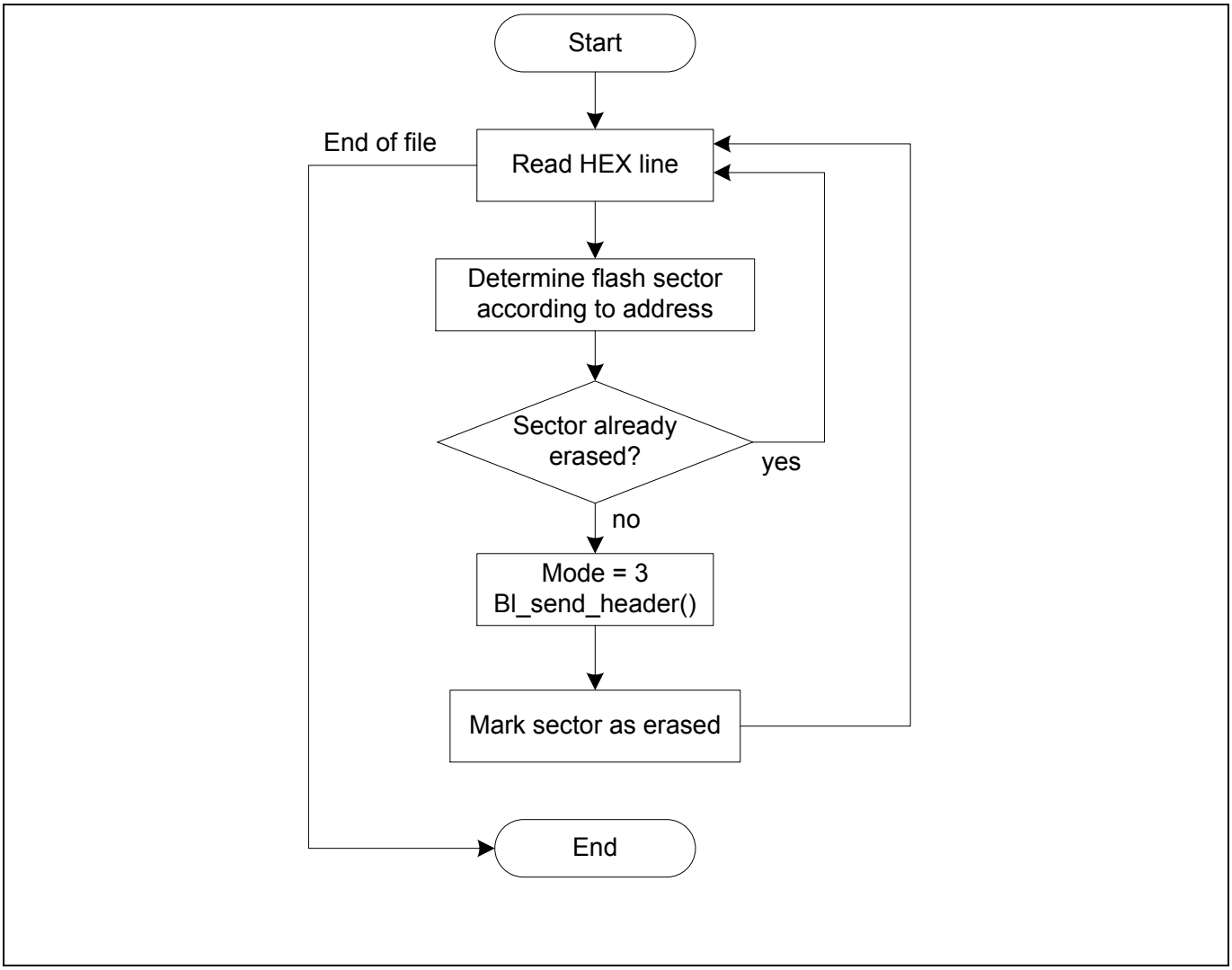


Figure 7 Flash erase procedure implemented in bl_erase_flash()

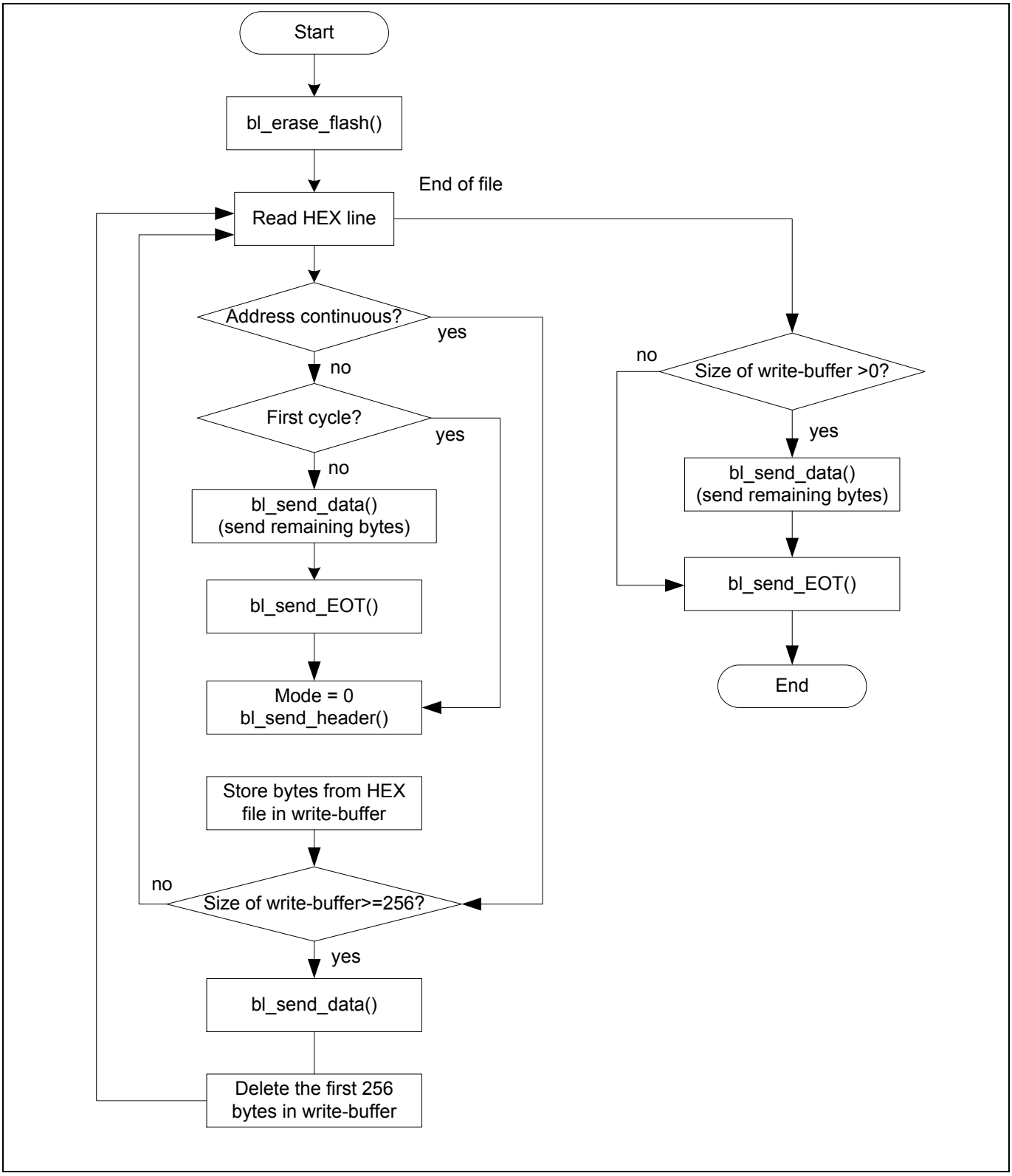


Figure 8 Flash programming procedure implemented in bl_download_flash()

4.1 Communication Protocol

The Flash loader program establishes a communication structure to receive commands from the HOST PC.

The HOST sends commands via transfer blocks. Three types of blocks are defined:

Header Block

Byte 0	Byte1	Bytes 2 … 14	Byte 15
Block Type (0x00)	Mode	Mode-specific content	Checksum

The header block has a length of 16 bytes.

Data Block

Byte 0	Byte1	Bytes 2 … 257	Bytes 258 … 262	Byte 263
Block Type (0x01)	Verification option	256 data bytes	Not used	Checksum

The data block has a length of 264 bytes.

EOT Block

Byte 0	Bytes 1 … 14	Byte 15
Block Type (0x02)	Not used	Checksum

The EOT block has a length of 16 bytes.

The action required by the HOST is indicated in the Mode byte of the header block.

The Flash loader program waits to receive a valid header block and performs the corresponding action. The correct reception of a block is judged by its checksum, which is calculated as the XOR sum of all block bytes excluding the block type byte and the checksum byte itself.

In ASC mode, all block bytes are sent at once via the UART interface. The different modes specify the Flash routines that will be executed by the flash loader. The modes and their corresponding communication protocol are described as follows.

4.1.1 Mode 0: Program Flash Page

Header Block

Byte 0	Byte1	Bytes 2 … 5	Byte 6 … 14	Byte 15
Block Type (0x00)	Mode (0x00)	Page Address	Not Used	Checksum

- Page Address (32bit)
  - Address of the Flash page to be programmed. The address must be 256-byte-aligned and in a valid range (see chapter 3), otherwise an address error will occur. Byte 2 indicates the highest byte, and byte 5 indicates the lowest byte.

After reception of the header block, the device sends either 0x55 as acknowledgement or an error code for an invalid block. The loader enters a loop waiting to receive the subsequent data blocks in the format shown below.

The loop is terminated by sending an EOT block to the target device.

Data Block

Byte 0	Byte1	Bytes 2 … 257	Bytes 258 … 262	Byte 263
Block Type (0x01)	Verification option	256 data bytes	Not used	Checksum

- Verification Option
  - Set this byte to 0x01 to request a verification of the programmed page bytes.
  - If set to 0x00, no verification is performed.
- Code bytes
  - Page content.
  - After each received data block, the device either sends 0x55 to the PC as acknowledgement, or it sends an error code.

EOT Block

Byte 0	Bytes 1 … 14	Byte 15
Block Type (0x02)	Not used	Checksum

After each received EOT block, the device sends either 0x55 to the PC as acknowledgement, or it sends an error code.

4.1.2 Mode 1: Erase Flash Sector

Header Block

Byte 0	Byte1	Bytes 2 … 5	Byte 6 … 14	Bytes 10 … 14	Byte 15
Block Type (0x00)	Mode (0x03)	Sector Address	Sector Size	Not Used	Checksum

- Sector Address (32bit)
  - Address of the Flash sector to be erased. The address must be a valid sector address, otherwise an address error will occur.
  - Byte 2 indicates the highest address byte.
  - Byte 5 indicates the lowest address byte.
- Sector Size (32bit)
  - Size of the Flash sector to be erased. The size must be a valid sector size.
  - Byte 6 indicates the highest address byte.
  - Byte 9 indicates the lowest address byte.
  - The device sends either 0x55 to the PC as acknowledgement, or it sends an error code.

Note: in example of this application note the sector address is fixed to partitions whose section address is storied in flash. So, no section address is transmitted from PC. Here, Byte2 contains the flash partition number.

4.2 Response Code to the HOST

The Flash loader program will let the HOST know whether a block has been successfully received and whether the requested Flash routine has been successfully executed by sending out a response code listed in Table 1.

Table 4 Table 1 Response Codes

Response Code	Description
0x55	Acknowledgement, no error
0xFF	Invalid block type
0xFE	Invalid mode
0xFD	Checksum error
0xFC	Invalid address
0xFB	Error during Flash erasing
0xFA	Error during Flash programming
0xF9	Verification error
0xF8	Flash partition error

5 Usage of Demonstrator

The example programs have been tested on Infineon XMC1300 boot kit. The user can use the example program to download user application codes (hex file format) into Flash. Here we give a description how to do that.

5.1 Hardware Setup

Connect the XMC1x00 boot kit to the PC host via a USB cable. Before opening the PC program, the following steps need to be done in XMC1300 kit using one of flash loader tools like Memtool and XMCFlasher:

1. Erase all flash
2. Load Bootstrap.hex file under .\Bootstrap\Debug into flash
3. Load SRAMCode.hex under .\SRAMCode\Debug into flash

With Memtool the BMI must be configured firstly as ASC Bootstrap Load Mode (ASC_BSL() . After the flash programming is finished, BMI needs to be configured back to User Mode (Debug) SWD_0. With XMCLoader BMI should be configured as User Mode (Debug) SWD_0.

Note: do not rebuild the SRAMCode project. If you want to rebuild the SRAMCode project in DAVE IDE, you need to modify the first line of SRAMCode.hex file in order that the address points to 0x10012000, where the flash loader is storied. Replace the first line in SRAMCode.hex file like:

:02000004**2000**DA → :02000004**1001**E9

5.2 Demonstrator File Structure

Figure 9 shows the file structure in the example programs.

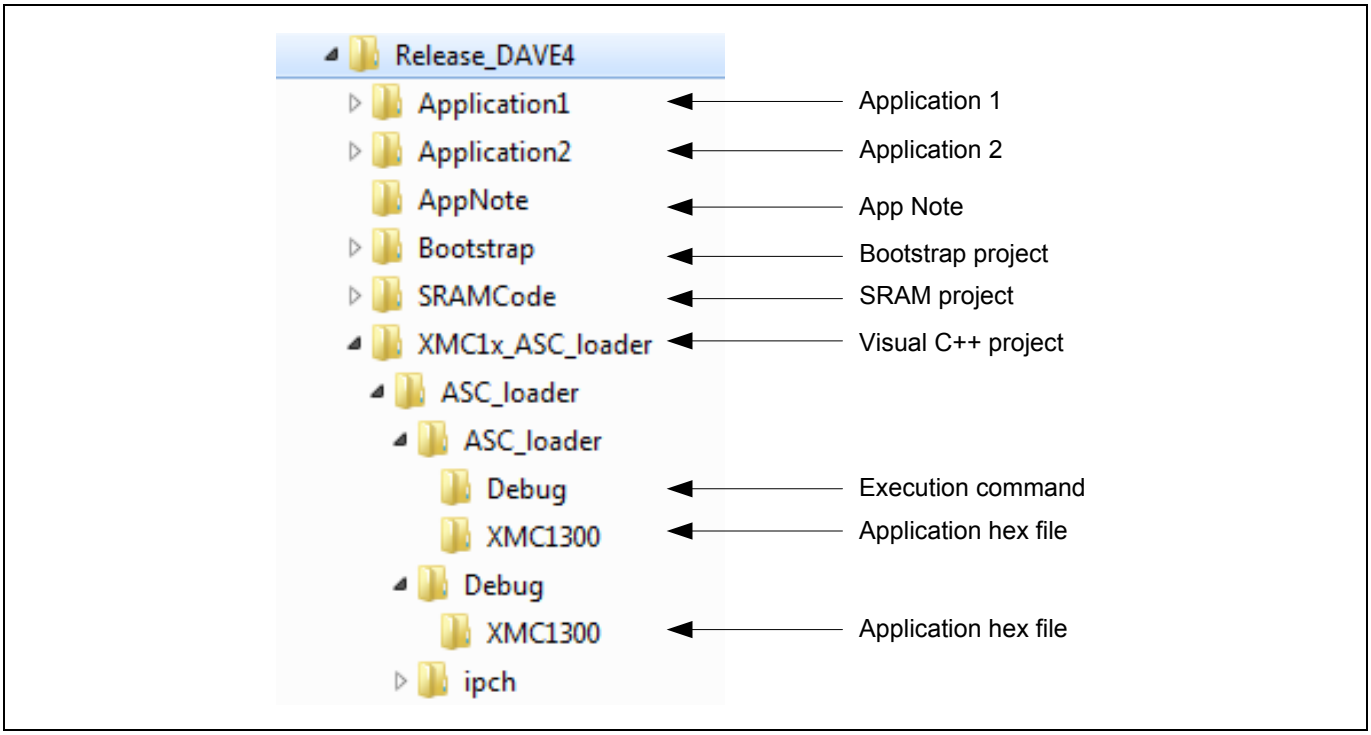


Figure 9 File structure of example programs

5.3 Run the Demonstrator

Before starting the demonstrator, the hex file that needs to be downloaded into Flash and copied into the folders .\ XMC1x_Load \Debug\XMC1300 and .\ XMC1x_Load \ XMC1x_Load \XMC1300 shown in Figure 10:

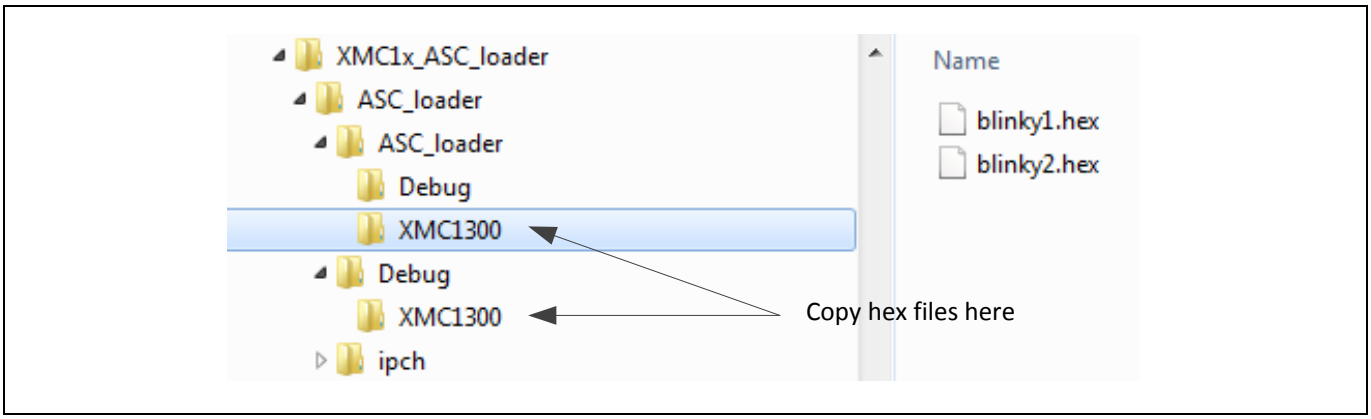


Figure 10 Location of object hex files to be flashed

There are two ways to start the demonstrator.

1. Double click the file ASC_loader.exe under .\ XMC1x_ASC_loader \Debug:

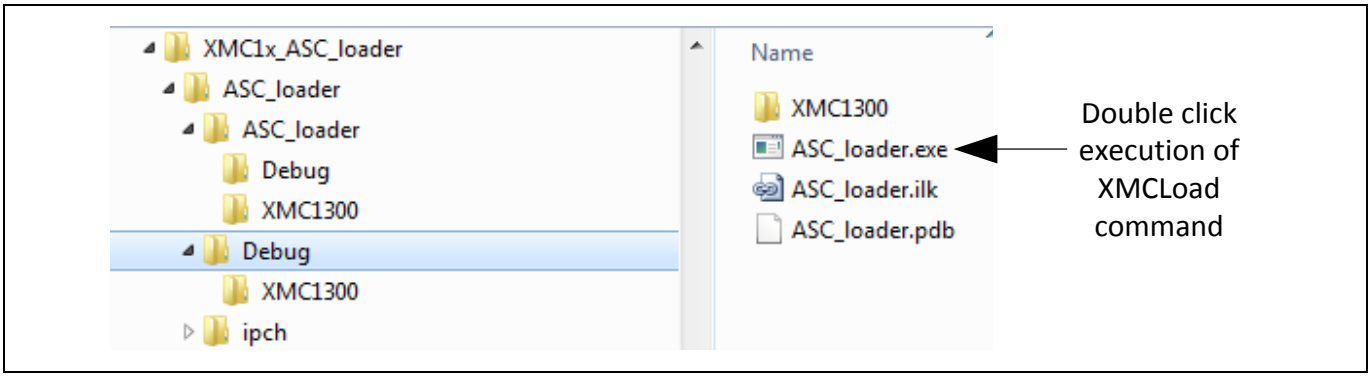


Figure 11 Direct start of demonstrator example

2. Double click the file XMCLoad.sln file in the folder .\XMC1x_Load to open the Microsoft Visual C++ project. The project in this Device Guide is developed using Microsoft Visual C++ 2010.

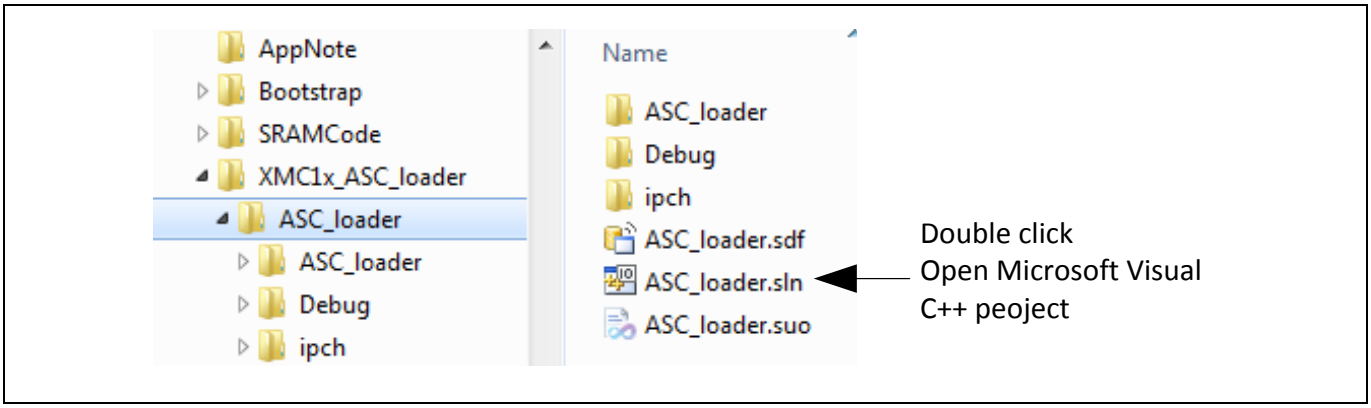


Figure 12 Start using Microsoft Visual project

In Microsoft Visual project workbench the project can be started from the “F5” key.

On starting the demonstrator the following window is displayed:

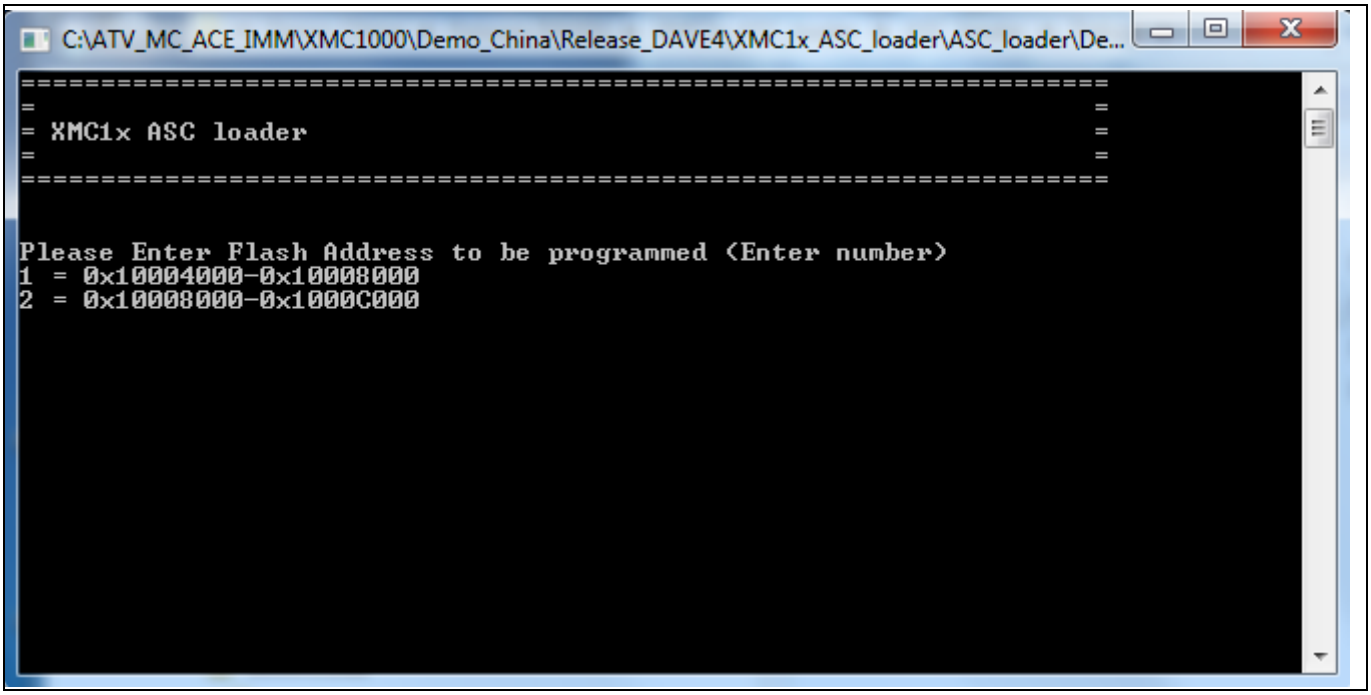


Figure 13 Start Window from Visual Project

Follow the instructions in the window to finish the Flash programming.

Note: The hex file name that will be programmed into Flash must be given completely with the file extension; e.g. blinky1.hex. Otherwise, the program does not know the file name. The Flash loader program accepts only hex file format.

After the hex file is programmed into Flash, a system reset is done to go back to Bootstrap program. The application program being just updated will be executed.

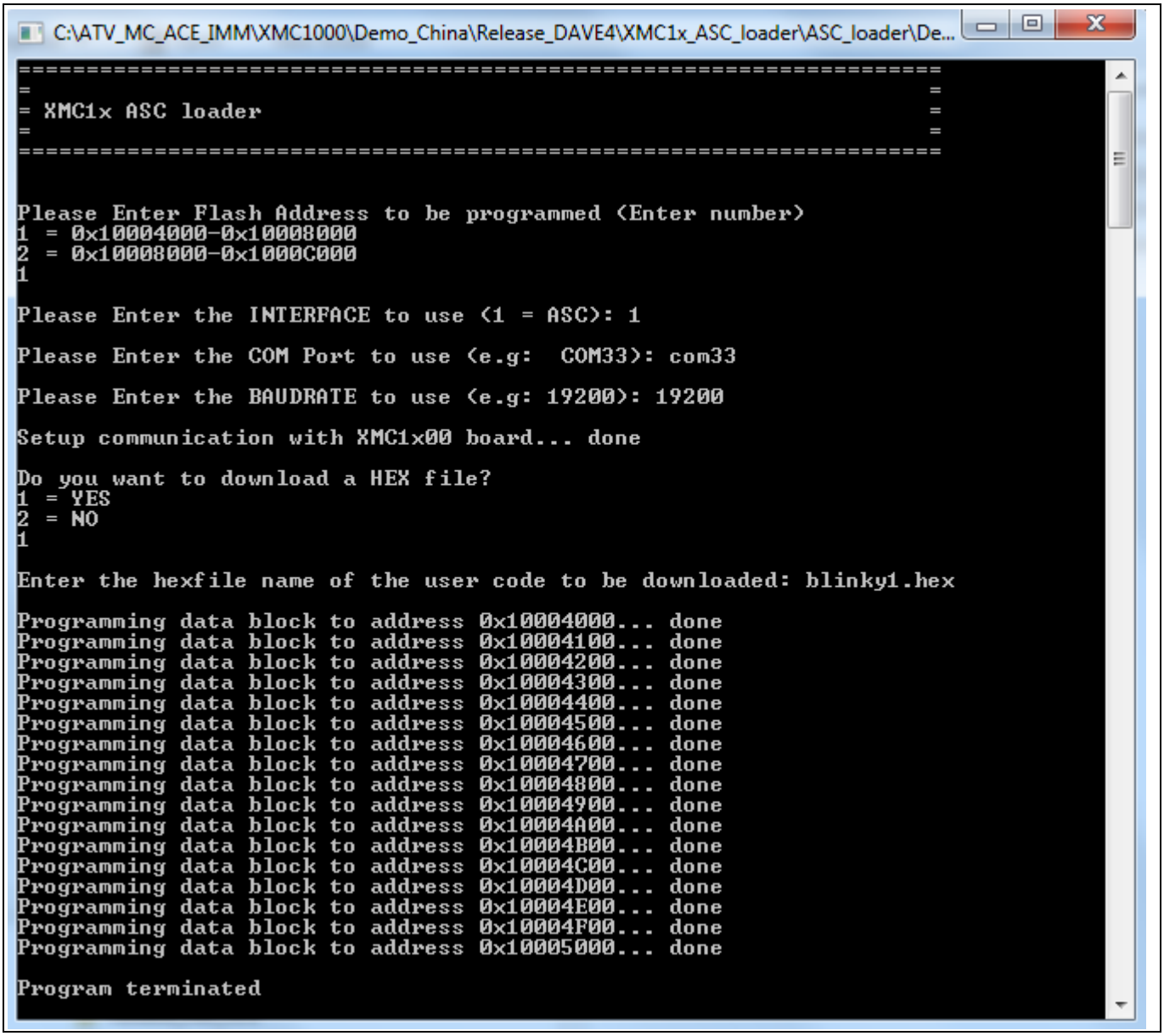


Figure 14 Window GUI illustrates the flash programing with Application 1.

Note: If we want to update Application 2 just after Application1 is successfully downloaded in flash, we need to do a power reset of on-board-debugger (OBD) in XMC1300 kit, because the debugger probe (XMC4200) in OBD must be reset for the next VCOM communication with PC. However, if the USIC module in XMC1300 device is used directly, this power reset is no necessary.

The mechanism to protect the currently used flash partition from update is built in the demonstrator example programs. The Flash Loader running in device checks firstly if the selected flash partition from Host PC is currently used. If yes, an error message is sent to Host PC to require a new choice of flash partition. If the partition is correctly chosen, the programming process goes ahead as shown in Figure 15.

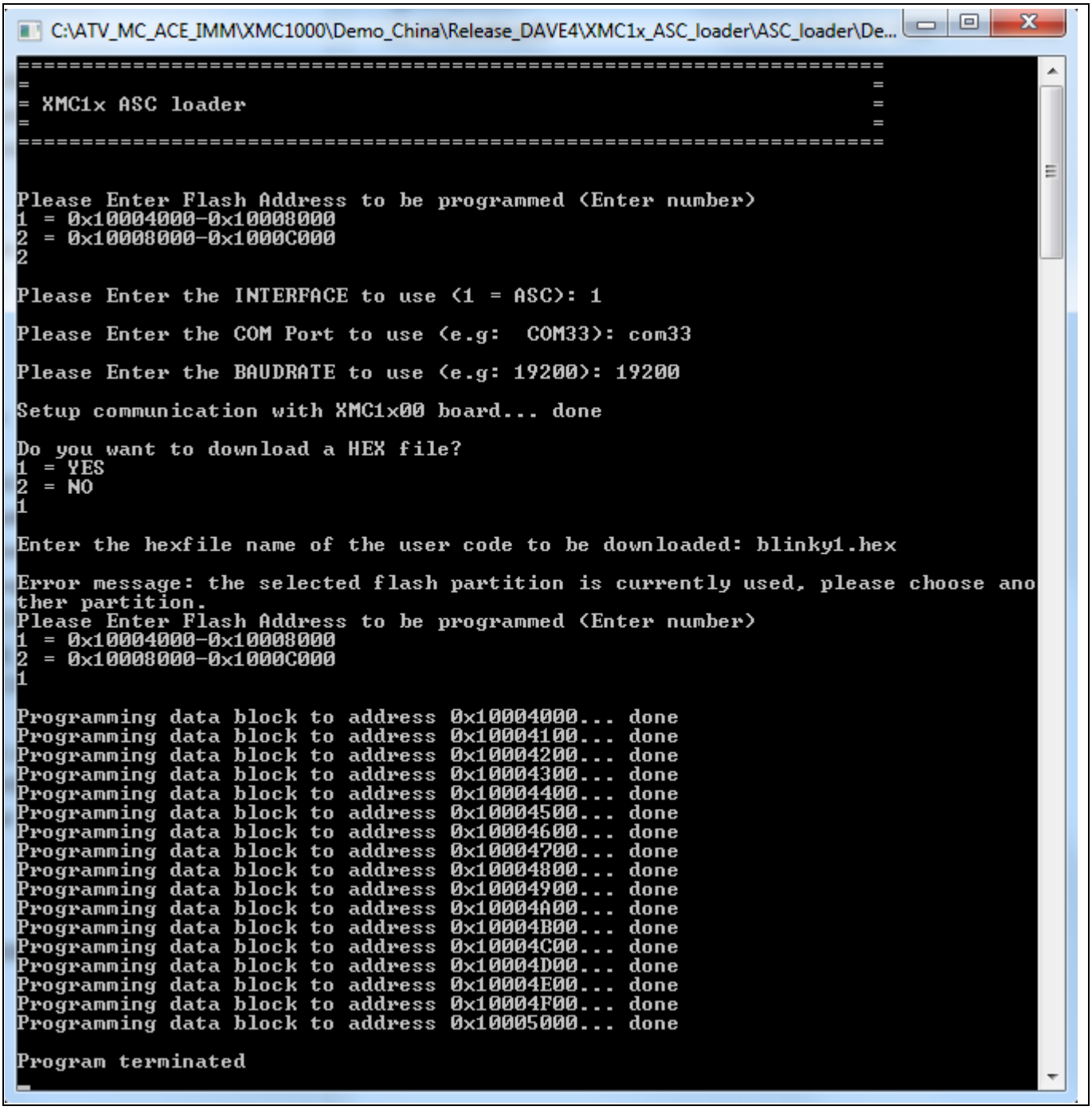


Figure 15 Protection of currently used flash partition

6 Reference Documents

Table 5 References

Document	Description	Location
XMC1x00 User's Manual	User's Manual for XMC1x00 device	http://www.infineon.com/XMC1000/RM
XMC1000 - ASC Bootstrap Loader	Application note for XMC1000	http://www.infineon.com/xmc1000/App

Revision History

Current Version is 1.0, 2016-05-28

Page or Reference	Description of change
V1.0, 2016-05	
	Initial Version

Trademarks of Infineon Technologies AG
AURIX™, C166™, CanPAK™, CIPOS™, CoolGaN™, CoolMOS™, CoolSET™, CoolSiC™, CORECONTROL™, CROSSAVE™, DAVE™, DI-POL™, DrBlade™, EasyPIM™, EconoBRIDGE™, EconoDUAL™, EconoPACK™, EconoPIM™, EiceDRIVER™, eupec™, FCOS™, HITFET™, HybridPACK™, Infineon™, ISOFACE™, IsoPACK™, i-Wafer™, MIPAQ™, ModSTACK™, my-d™, NovalithIC™, OmniTune™, OPTIGA™, OptiMOS™, ORIGA™, POWERCODE™, PRIMARION™, PrimePACK™, PrimeSTACK™, PROFET™, PRO-SIL™, RASIC™, REAL3™, ReverSave™, SatRIC™, SIEGET™, SIPMOS™, SmartLEWIS™, SOLID FLASH™, SPOC™, TEMPFET™, thinQ!™, TRENCHSTOP™, TriCore™.

Trademarks updated August 2015

Other Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2016-05

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2016 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference
AP32337

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.