Software update of XMC1000 microcontrollers using a SPI 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 the host PC is determined by the application, and could be ASC, SPI or IIC. DAVE™ example projects are provided with this application note to demonstrate how to implement the remotely controlled flash update system. In the demo codes, a SPI (Serial Peripheral Interface)/SSC (Synchronous Serial Channel) interface is used to communicate with the PC through a gateway between SPI and ASC, where the gateway is implemented in a XMC1300 kit. The applicable products are the XMC1000 microcontroller family. The example codes are tested on the XMC1300 boot kit.

Intended audience

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

Table of contents

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 takes place during normal operation of the microcontroller (during run-time). The communication interface used between the microcontroller and the PC host is SPI (Serial Peripheral Interface), which is implemented using the SSC (Synchronous Serial Channel) communication protocol in the USIC module of the XMC1000 microcontroller family. As a normal PC has no SPI interface, a communication gateway is required to change the data format from SPI (target board) to ASC (host PC). On the gateway board an ASC interface is connected to the PC host using a USB VCOM CDC adapter. This concept is applicable to other communication options such as USB, Ethernet, etc.

For safety reasons, the concept used in this application note assumes the existence of two fully independent application programs, application program 1 and application program 2. A Bootstrap program establishes which of the two application programs is the most recent version and executes it. When updating, the currently unused application program is erased and replaced by the update. This concept ensures that, in the case of data or power loss during the update, no corruption occurs as the currently used application program is not touched.

The update procedure is managed by the flash loader that is stored in flash. Upon receiving an update request from the PC host, the flash loader is copied from flash into the SRAM (by the currently used application program). After the copying is finished, the flash loader takes control and communicates via a SPI interface (USIC module) with the PC through the gateway board according to a defined protocol, obtains the hex file then erases and programs the flash. The hex file transferred from the PC host to the microcontroller is not encrypted. After a 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 the following types of programs are required:

- Bootstrap program:
  Program that is executed after reset that determines which of the 2 application programs should be executed afterwards
- Flash loader:
  Program that includes flash driver routines to erase and program the flash as well as the communication and protocol routines to connect and communicate with the PC host via the gateway
- Application program:
  Program that provides the intended application use case running on the MCU.
- PC host:
  PC that is used to connect to the MCU via the gateway to update the MCU software and the application program

The SW package provided with this application note is structured into the following separate projects, developed with DAVE™ v4 for XMC1300:

1. Boot loader project
   Includes all the required adaptations of the linker script
2. Flash loader project
   Includes the required adaptations of the linker script and the startup files
3. Project for the application program 1
   Simple use case for demonstration purposes (Blinky), it includes all required linker script adaptations
4. Project for the application program 2

Simple use case for demonstration purposes (another Blinky), it includes all required linker script adaptations

5. Project for the gateway
   Gateway program executed on XMC1300 kit

MS Visual C++ projects for the PC host

- PC program to communicate with the gateway 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 the XMC1000 device are developed with the following tool-chain:

- DAVE™ V4 development platform v4.3.2

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.
- .\Gateway_XMC13, contains the gateway program executed on XMC1300 kit

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

2 Concept of the demonstrator

2.1 Flash partitioning

Flash partitioning is the first required step for the implementation of a flash update using a host PC, as the flash sections for the different program storage must be defined before starting to write the application code. In this application note we use the flash partition shown in Figure 1.

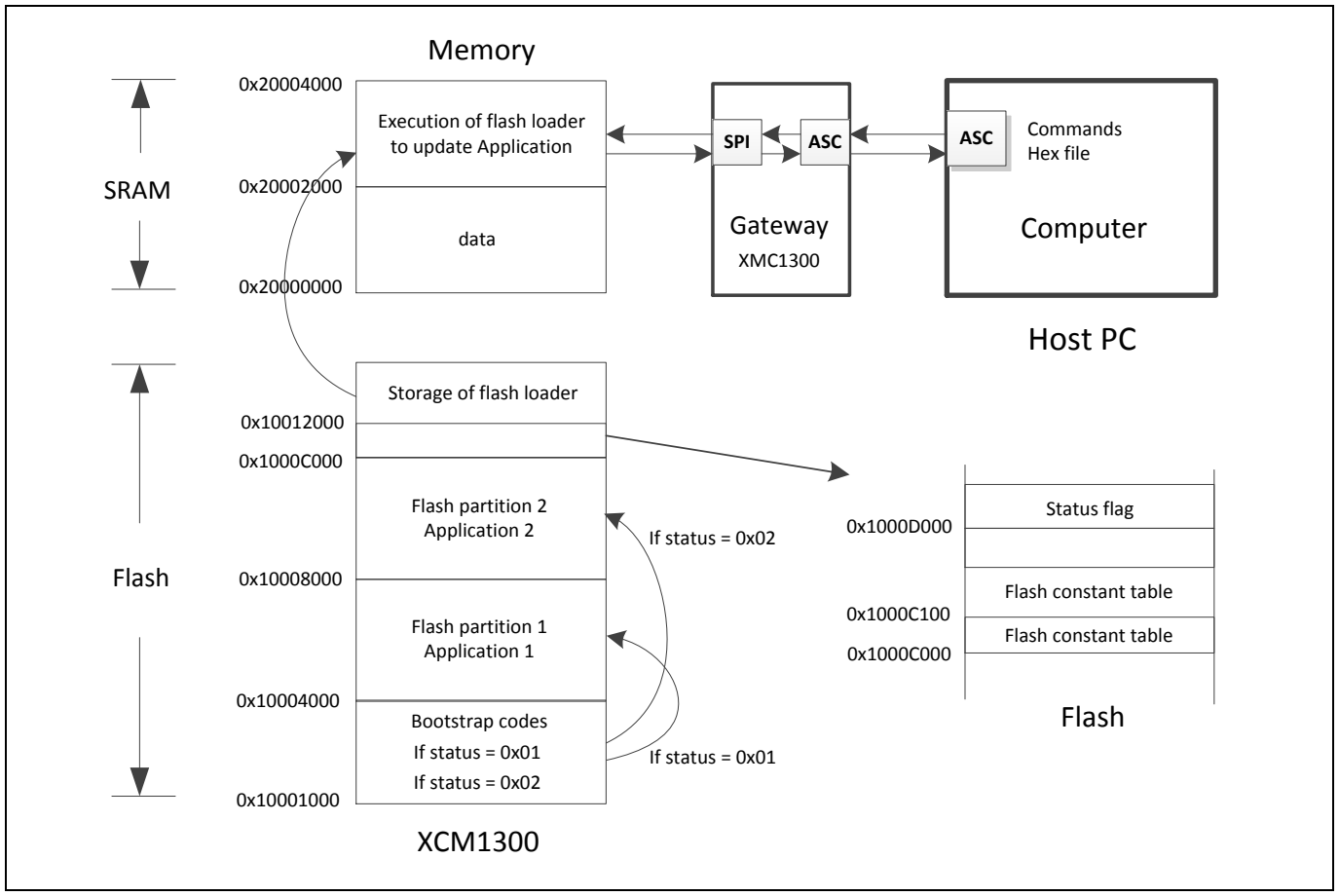

Figure 1 Flash partition of demonstration for flash update using a host PC

In Figure 1 we partition the flash into 4 parts:

- Bootstrap codes: 0x10001000 – 0x10003FFF (12 KB)
- Application 1: 0x10004000 – 0x10007FFF (16 KB)
- Application 2: 0x10008000 – 0x1000BFFF (16KB)
- Storage for flash loader: 0x10012000 – 0x10013FFF (8 KB)

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

Note: A gateway is required in this application example to exchange the data between the ASC and SPI interfaces, as the host PC only has an ASC interface. The SPI data from the target board (XMC1300 kit) needs to

be converted to ASC data in the gateway, and then transferred to the PC. In this example, we are using a second XMC1300 kit as a gateway.

2.2 Bootstrap program

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

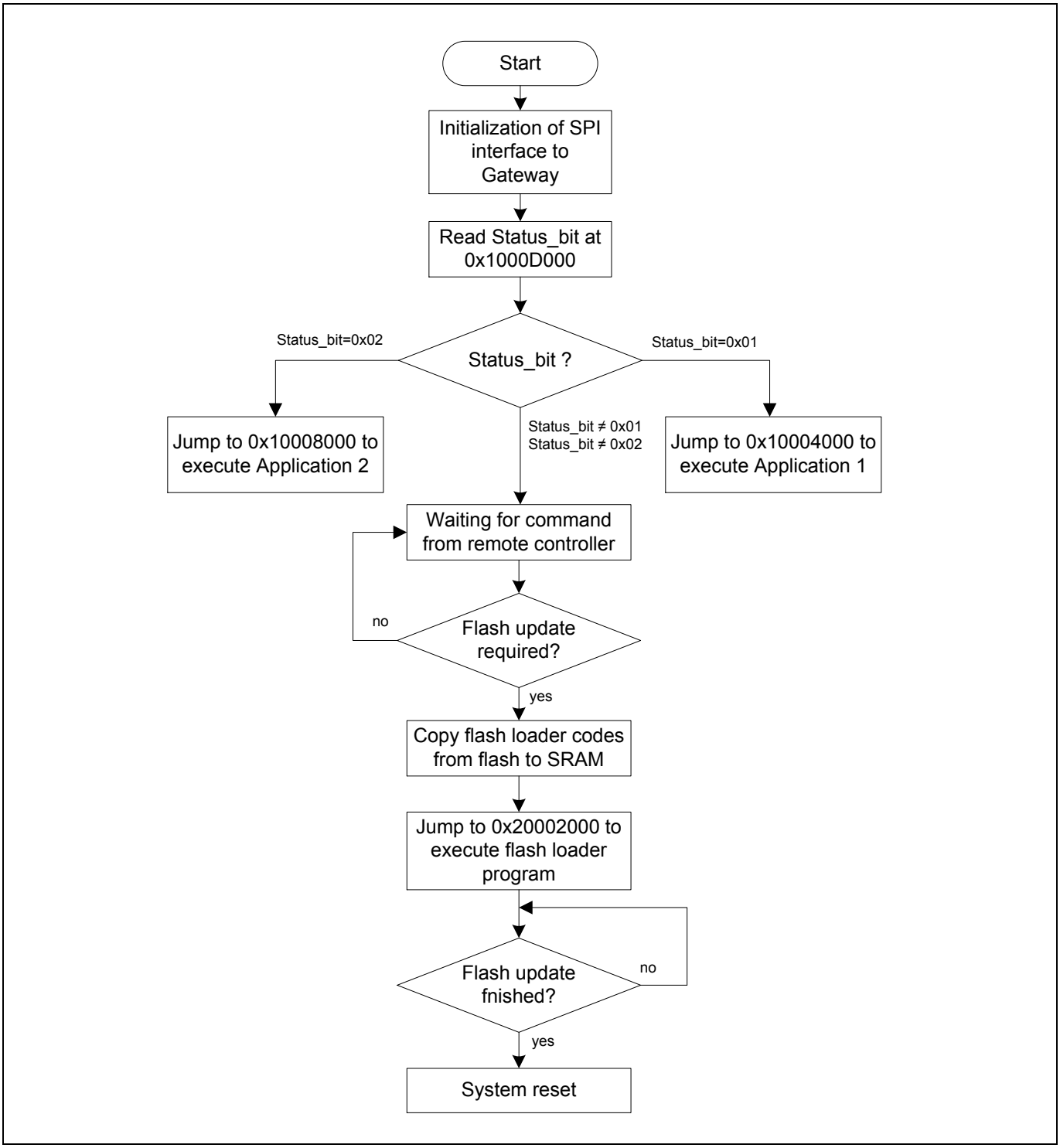

Figure 2 Program flow of bootstrap codes

At the start of bootstrap program the communication interface (SPI) to the gateway must be initialized and be ready to communicate with the gateway. After initialization, the status bit saved at 0x1000D000 will be verified. If the status flag is equal to 0x01, this means that application 1 will be executed. Next, the program counter will be loaded with 0x10004000, where application 1 is located and stored. The program then jumps to application 1 to execute. If the status flag is equal to 0x02, then the program jumps to application 2 located at 0x10008000 to execute.

If the status flag is neither 0x01 nor 0x02, then the program is waiting for a command from the 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 control of the XMC1000 device and communicates with the PC to finish the application update in the flash. After the application codes are completely updated, the status flag at 0x1000D000 will also be updated to indicate the new updated application. Finally a system reset will be performed to reset the system.

2.2.1 Modification of linker description (ld) file

Two important flash sector tables are defined in the Bootstrap project. These 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 the flash loader running from SRAM, but are saved in flash at 0x1000C000 and 0x1000C100. To locate the constant table at a dedicated address we need to define a special section in the linker description file similar to this:

```
 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 the above modifications in the ld file and the definition in the main function, both tables are located at the dedicated address.

2.3 Application programs

Application programs are user specific and are developed according to the application requirements. However, in the application programs, initialization of the communication interface is mandatory to enable the PC to access the device during the running of application codes. So as not to disturb the normal execution of application programs, an interrupt is used to access the device from the PC. Figure 3 shows the major program flow for application codes.

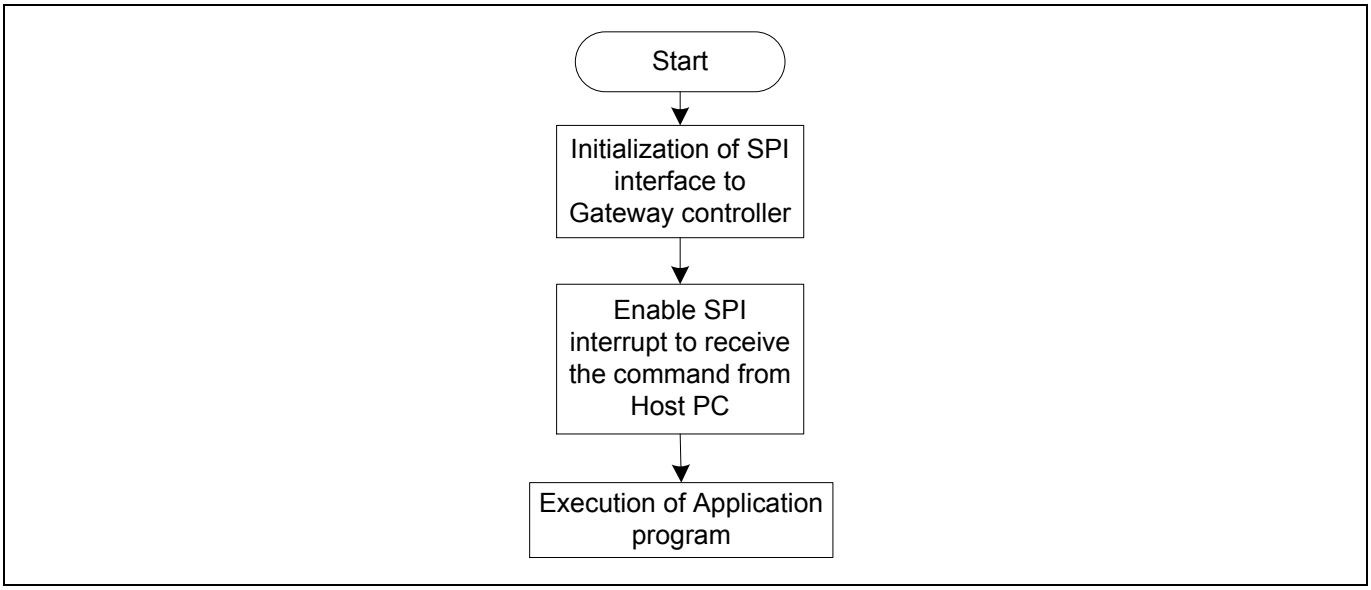

Figure 3 Major program flow of application codes

In the interrupt handler routine all interrupts should firstly be disabled. Then, the SPI interrupt status flag will be checked if the status flag is valid. If flag is equal to “1” indicating a valid status flag, the interrupt routine continues execution. Otherwise, the interrupt routine will be terminated. In the case of a valid status flag, the status flag will be cleared, and the data will be read. If the received data is equal to “0x5D” that means that a software update is required and the flash loader codes are copied from flash to SRAM and executed there to complete the flash update operation. The general flow of the interrupt handler is shown in Figure 4.

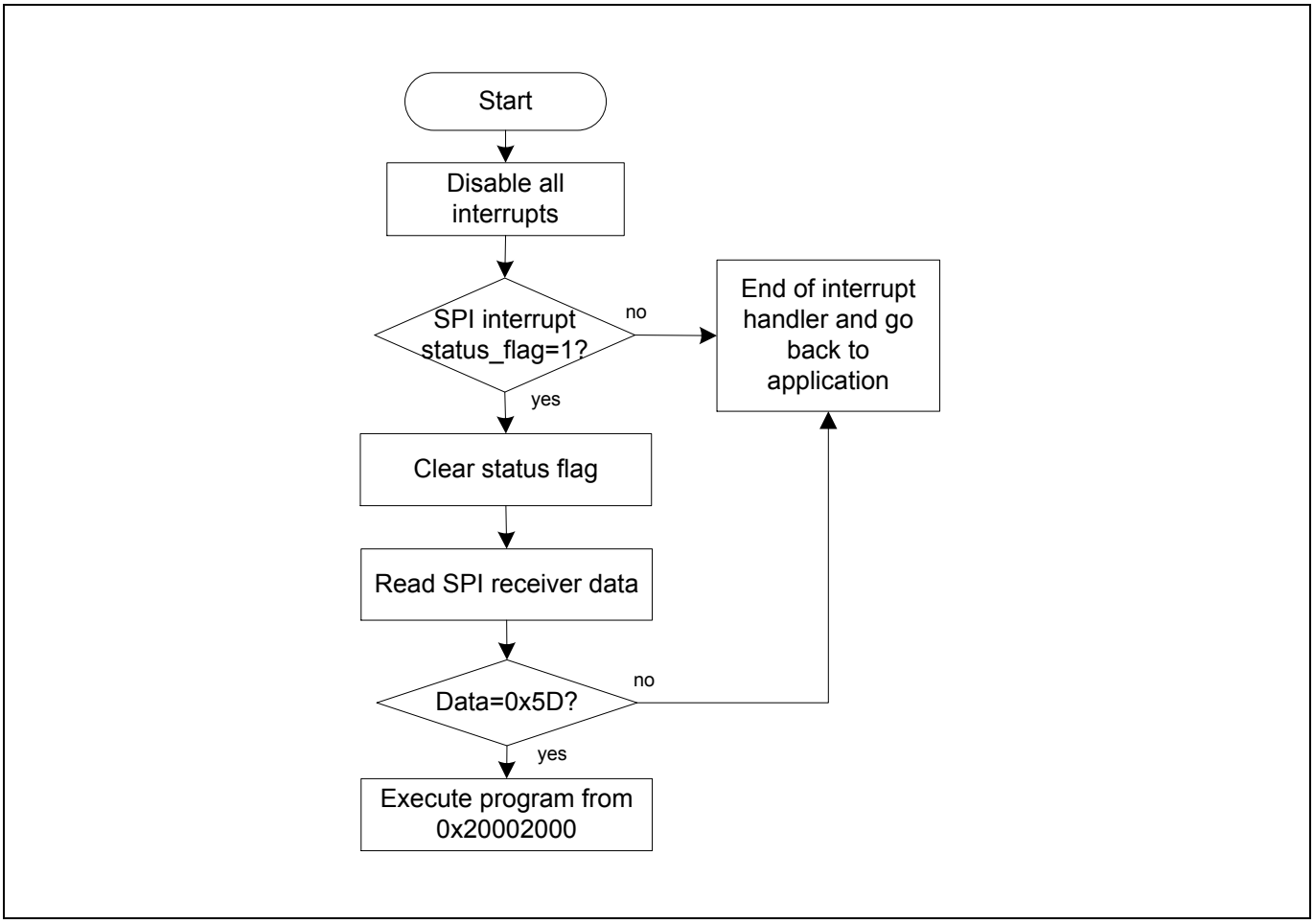

Figure 4 Program flow of the interrupt handler in application codes

2.3.1 Example of application programs

The demo codes in this application note contain two application programs. A system timer is used to toggle the LEDs on the 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. Note that application 1 toggles all 6 LEDs, while application 2 toggles just 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 the application project needs to be modified. Furthermore, an 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 DAVE™ V4 v4.3.2. To ensure the applications will be located in different partitions, the memories in linker descript file (linker_script.ld) need to be changed in this way:

For application 1:

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

}

where flash starts at 0x10004000 with a 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 the application requirements.

For application 2:

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

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

Note: if we want to change the start address and the size of the applications, we need to modify the flash partition and the sector tables of XMC1000_FLASH1_SectorTable and XMC1000_FLASH2_SectorTable in Device_Memory.h of the Bootstrap program. For example, if flash partition1 is extended to 32kb, the 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 Copying the flash loader program to SRAM

Here is an example of an 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 IRQ9_Handler(void)
{
       int i;

       //disable interrupt
       XMC_SPI_CH_DisableEvent(SPI_SLA_CH, XMC_SPI_CH_EVENT_ALTERNATIVE_RECEIVE);
       XMC_SPI_CH_DisableEvent(SPI_SLA_CH, XMC_SPI_CH_EVENT_STANDARD_RECEIVE);

       SysTick->CTRL  &= 0xFFFFFFFC;
       NVIC_DisableIRQ(IRQ9_IRQn);

      //check interrupt status flag
      if((((XMC_SPI_CH_GetStatusFlag(SPI_SLA_CH) &
                  MC_SPI_CH_STATUS_FLAG_ALTERNATIVE_RECEIVE_INDICATION)>>15)| \
              ((XMC_SPI_CH_GetStatusFlag(SPI_SLA_CH) &
              XMC_SPI_CH_STATUS_FLAG_RECEIVE_INDICATION)>>14))== 1U)
       {
              /* Clear flag */
              XMC_SPI_CH_ClearStatusFlag(SPI_SLA_CH,
                           XMC_SPI_CH_STATUS_FLAG_ALTERNATIVE_RECEIVE_INDICATION);
              XMC_SPI_CH_ClearStatusFlag(SPI_SLA_CH,
                           XMC_SPI_CH_STATUS_FLAG_RECEIVE_INDICATION);

              /* Read received data */
              RxData = XMC_SPI_CH_GetReceivedData(SPI_SLA_CH);

              if(RxData == 0x5D)
              {
                     for (i=0; i<TABLE_SIZE; i++)  //copy flash loader into SRAM
                     {
                            *RamAddr = *FlasAddrSys;
                            RamAddr++;
                            FlasAddrSys++;
                     }
                     RunRAM();  //run the program from SRAM

              }
       }

}

void RunRAM(void)
{
  __asm
  (
    "LDR r0, =0x20002001;"
    "BLX r0;"
  );

}
```

Note: Cortex-M0 has 16-bit thumb instructions, so 0x2000200**1** should be loaded to register R0 instead of 0x20002000.

3 Flash loader program

In the demo in this application note we use the SPI interface on the XMC1000 device to communicate with the PC to complete the flash update. The mechanism for failure handling is built into the flash loader program. If the flash programming fails due to power loss or break of communication between the PC and device, the previous application codes remain the default code and the application system is not impacted. A new download process can be started.

3.1 Initialization of SPI module

The communication between the PC and the target device is established via a gateway controller realized on an XMC1300 kit. The target kit is connected with the gateway via an SPI/SSC interface as shown in Figure 5. On both the target and gateway device sides, channel 0 of USIC0 (U0C0) is configured as the SPI/SSC. Ports P0.8, P0.9, P1.0 and P1.1 are configured as SCLK (shift clock), SEL (slave select), MOSI (master output and slave input) and MISO (master input and slave output) for the SPI bus, respectively.

- Shift clock pin SCLK at pin P0.8 (USIC0_CH0.DX1B)
- Slave select pin SEL at pin P0.9 (USIC0_CH0.DX2B)
- Master output and slave input MOSI at pin P1.0 (USIC0_CH0.DX0C)
- Master input and slave outputs MISO at pin P1.1 (USIC0_CH0.ALT7)

In this application note we use the XMC1300 to implement the gateway functionality. Channel 1 of USIC0 (U0C1) in the XMC1300 is configured as the ASC module. Ports P1.3 and P1.2 are configured as RxD (receiver data) and TxD (transmitter data), respectively.

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

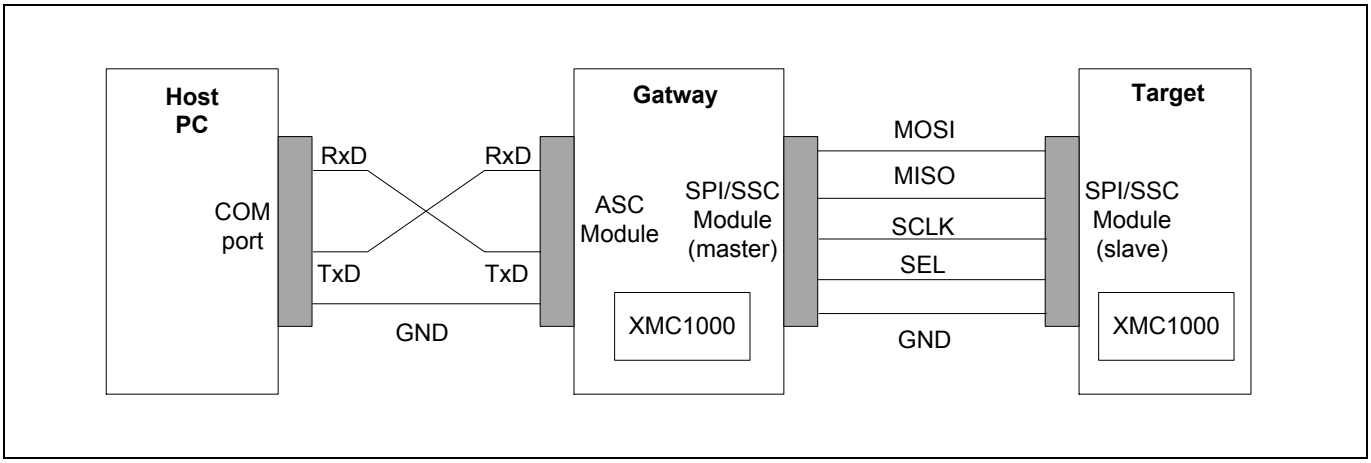

Figure 5 Connection between PC and XMC1000

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

The ASC and SPI modules must be initialized in the gateway controller. The detailed implementation of the gateway will be described in section 4.

3.2 Flash loader procedure

The flash loader procedure is shown below in Figure 6. Before entering the flash loader, the SPI (slave) communication is already set up, which is completed in the application code. In the flash loader program, the host starts by transmitting a header data block to inform the device what needs to be done. In order to send the data from the PC to the device we have defined different header data blocks. The communication protocols are described in Chapter 5 of this application note.

If the header data is correctly received by the device, the flash loader responds with 0x55 to represent a successful receive. At the same time, the data block is evaluated by the flash loader program to check which command has been sent from the PC. If it is a flash erase command, the flash erase routine will be called to execute a 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 0x55 to the host PC to indicate successful flash programming, and updates the status bit of the application program. Before leaving the flash loader program a system reset will be performed. Then, the device starts again from the ROM codes. The application program that was just updated will be executed after a system reset.

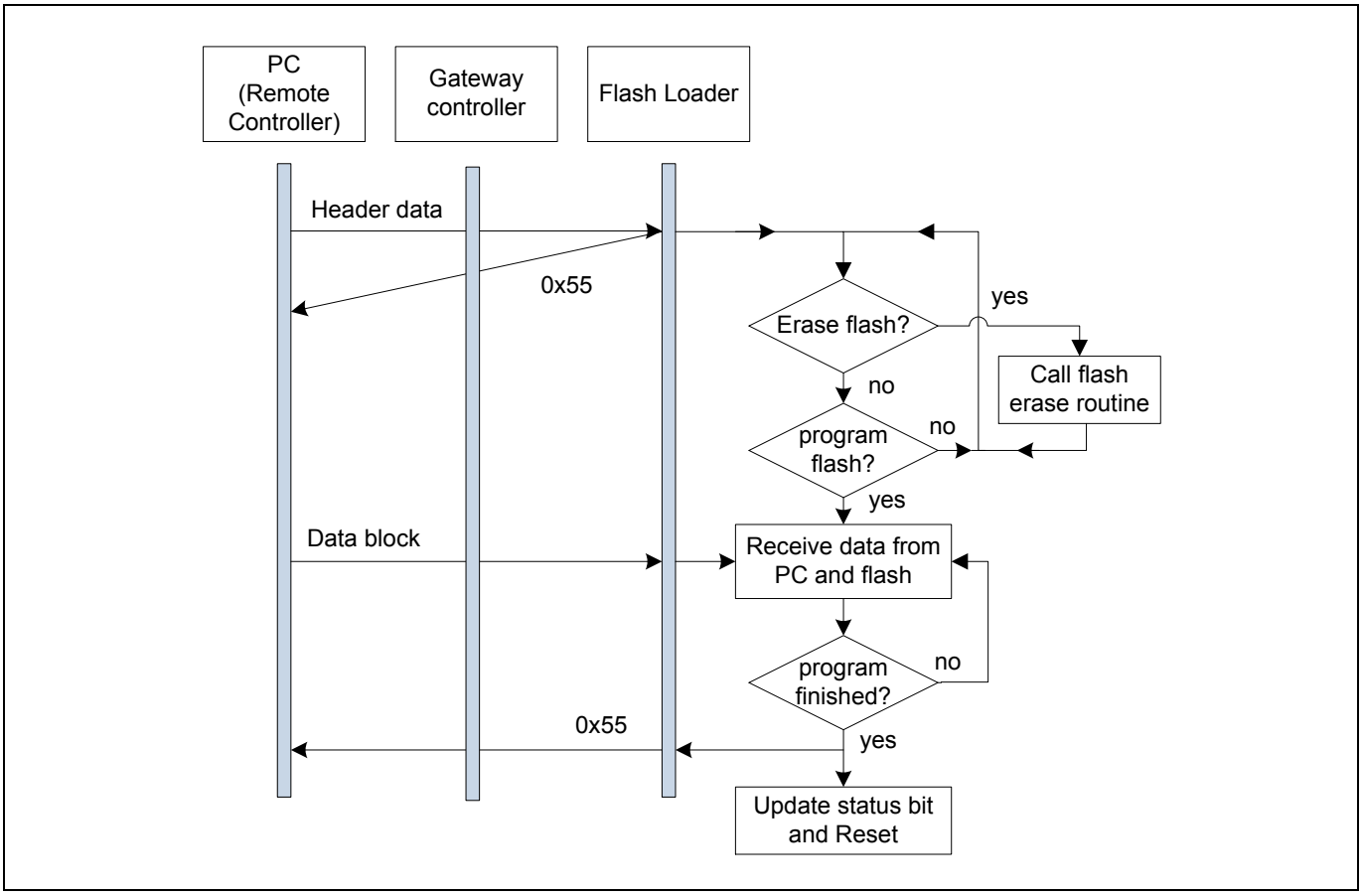

Figure 6 Flash loader procedure for flash programming

3.3 Example of flash loader program

An example of a flash loader program developed with DAVE™ is provided in this application note. The flash loader implements the flash routines and establishes communication between the 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 the PC is the same, the flash loader program can be reused for all applications, independent from the application codes. Below we give the DAVE™ v4 project settings.

3.4 DAVE™ v4 project settings

The flash loader DAVE™ v4 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 the hex file to the flash address.

3.4.1 Modification of DAVE™ linker descript (ld) file

The flash loader program must be located in SRAM starting at 0x20002000, as the Flash loader program can only run from SRAM. Therefore, the default linker script file generated from DAVE™ V4 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 in the memory definition:

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

Here we continue using the names “Flash” and “SRAM” in the memory definition in order to avoid changes to the rest of the ld file. However, all memory locations in this case 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 Section 3.5, “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 the SRAM code project must be removed so that the clock settings made in the application programs can be retained without modification. Otherwise, the communication with the gateway controller 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
```

Flash **loader program**

```
ldr     r3, =VeneerEnd
bl  __copy_data

ldr  r0, =SystemInit
blx  r0
```

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

The startup.S files provided in the SRAM code project have 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 kB	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

Flash **loader program**

Sector	Address range	Size
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
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

Flash **loader program**

Sector	Address range	Size
47	0x1002F000 – 0x1002FFFF	4 kB
48	0x10030000 – 0x10031FFF	4 kB
49	0x10032000 – 0x10032FFF	4 kB

4 Gateway implementation

In principle, the gateway functionality can be implemented on any microcontroller that has SPI/SSC and ASC modules. As shown in Figure 5 we implement the gateway using the XMC1300 kit in this application note. The gateway has the following tasks:

- Setup ASC communication with PC
- Setup SPI communication with target kit
- Data format transfer between PC and target kit

SPI in the gateway is configured as **Master**. Below is the main function of the gateway as implemented in this application note:

```
int main(void)
{
  int i;

  SystemCoreClockUpdate();//system clock update
  ASC_Init();  //ASC initialization

  //SPI master init
  SPI_Master_Init(SPI_MAS_CH);

  //set up ASC communication with PC
  while(((USIC0_CH1->TRBSR & (0x01UL << 3) ) >> 3)) {};
     TxData = (USIC0_CH1->OUTR & 0xFF);

  if (TxData == 0x5D)
  {
       SendByte(TxData);  //send ACK to PC
       Write_Slave(SPI_MAS_CH,TxData); //send ACK to XMC1000 to initialize the software
update
       RxData = Read_Slave(SPI_MAS_CH); //dummy read
  }
  else
         SendByte(BSL_MODE_ERROR);
  //end


  while(1)
  {
       while(((USIC0_CH1->TRBSR & (0x01UL << 3) ) >> 3))   {};  //gateway ASC receiver
         {
              TxData = (USIC0_CH1->OUTR & 0xFF);
              Write_Slave(SPI_MAS_CH,TxData);  //SPI gateway: transfer to XMC1000 SPI slave
              RxData = Read_Slave(SPI_MAS_CH); //dummy read

              Write_Slave(SPI_MAS_CH,TxData);  //SPI gateway: transfer to XMC1000 SPI slave
              RxData = Read_Slave(SPI_MAS_CH); //dummy read

              if((RxData != 0xAA) && (RxData != 0xBB) && (RxData != 0xCC))  //0xAA is defined
as ACK from slave and do not transfered to PC
              {
                 SendByte(RxData); //just the response codes defined in AppNote are transfered
to PC
                 XMC_GPIO_ToggleOutput(LED0); //toggle LED
```

```
            }

            else if(RxData == 0xCC)
            {
               for(i=0; i<500000; i++){;}  //delay to wait for flash erase finished
               Write_Slave(SPI_MAS_CH,TxData);  //SPI gateway: transfer to XMC1000 SPI slave
               RxData = Read_Slave(SPI_MAS_CH); //dummy read
               SendByte(RxData); //send ACK to PC
            }

            else if(RxData == 0xBB)
            {
               for(i=0; i<5000; i++){;} //delay to wait for flash programming finished
               Write_Slave(SPI_MAS_CH,TxData);  //SPI gateway: transfer to XMC1000 SPI slave
               RxData = Read_Slave(SPI_MAS_CH); //dummy read
               SendByte(RxData); //send ACK to PC
            }
         }

   }

}
```

Note: The following points must be noted by the SPI gateway implementation:

1. The SPI communication protocol is a fully master-controlled communication protocol, the slave has no control capability. The data communication is performed synchronously by a master writing operation. Here, the gateway is configured as the master, while the target kit is configured as the slave. Thus, all communication is initialized by gateway.
2. During writing of data to the slave, the master reads data synchronously from the slave. However, the data read from the slave is the previous data saved in the transmit data buffer (TBUF). To obtain the correct ACK from the flash loader program the slave needs to be read twice.
3. The SPI interface on the target board is the slave, where the flash program is running. As an SPI slave cannot initiate the sending of any information to the master, all ACK information from the target board must be read by the master. To ensure that the master reads the correct ACK response from the slave, two delays (implemented as “for” loops) are inserted into the code. One delay is used to wait for the flash erase complete ACK response. The second delay is for flash programming. The Flag “0xCC” indicates that the flash erase operation is ongoing, while the flag “0xBB” indicates that the flash programming is ongoing.

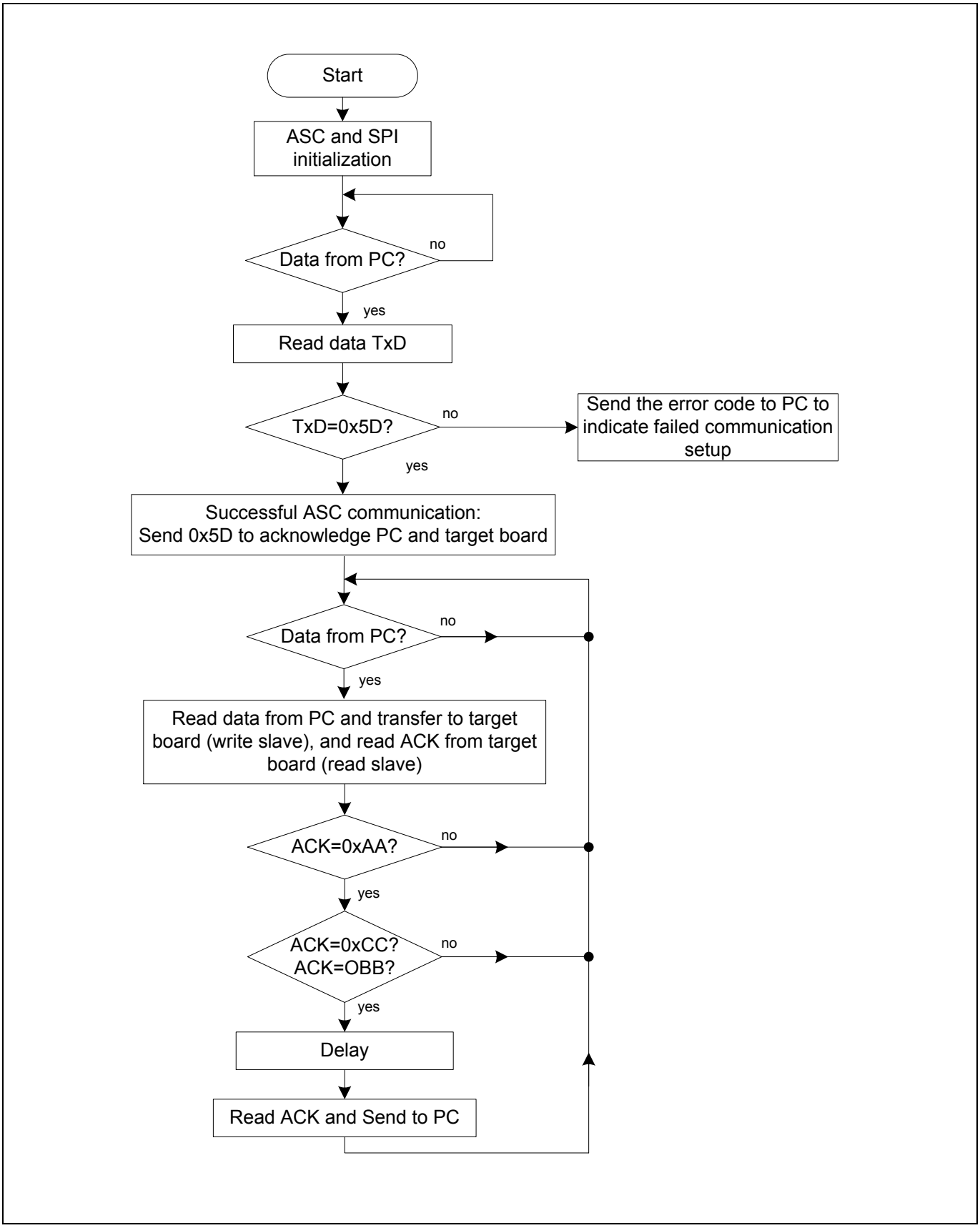

Figure 7 Program flow of the gateway implementation

Gateway **implementation**

Figure 7 shows the program flow of the gateway implementation in this application note. The first part of the code is the ASC and SPI module initialization. After initialization, the communication with the PC via the COM port is performed. In this application, we configure ASC with a fixed baud rate of 19200 bit/s. The communication with the PC will be initiated by the PC program. In the gateway, the program waits for the start command from the PC. First, the PC sends a “0x5D” to the gateway to initiate the communication. After receiving the data, the gateway checks if the data is equal to “0x5D”. If it is, then the gateway sends a “0x5D” in response to the PC, and simultaneously sends “0x5D” to the target board via SPI to indicate the start of the software update. If the data received is not “0x5D”, then the gateway sends an error code to the PC.

The codes within the “while” loop are used for data transfer between the PC and the target board. It must be noted that the SPI protocol is a master controlled communication protocol, meaning that all data communication is controlled by the master. Here, the gateway is the master, and the target board is the slave. Thus, all data communication between the gateway and the target board is controlled by the gateway. To ensure that the command is correctly transferred to the target board, the PC program waits for an acknowledgment (ACK) response for each command. However, the SPI slave is unable to send response data back to the gateway. As a result, the gateway needs a slave read function after each write slave operation. However, we simply need to transfer the ACK codes for commands back to the PC. For the data communication from the PC to the target board the ACK is unnecessary. Therefore, a check operation is performed before sending data back to the PC. In this application note, we use “0xAA” to indicate the unused ACK code. Only ACK codes that are not equal to “0xAA” will be transferred to the PC. Furthermore, we need two flags of “0xBB” and “0xCC” to show the flash erase and programming status. For detailed information please refer to the note text above after the function “main()”.

5 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 an 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 8.
- The flash programming procedure is implemented in bl_download_flash() shown in Figure 9.

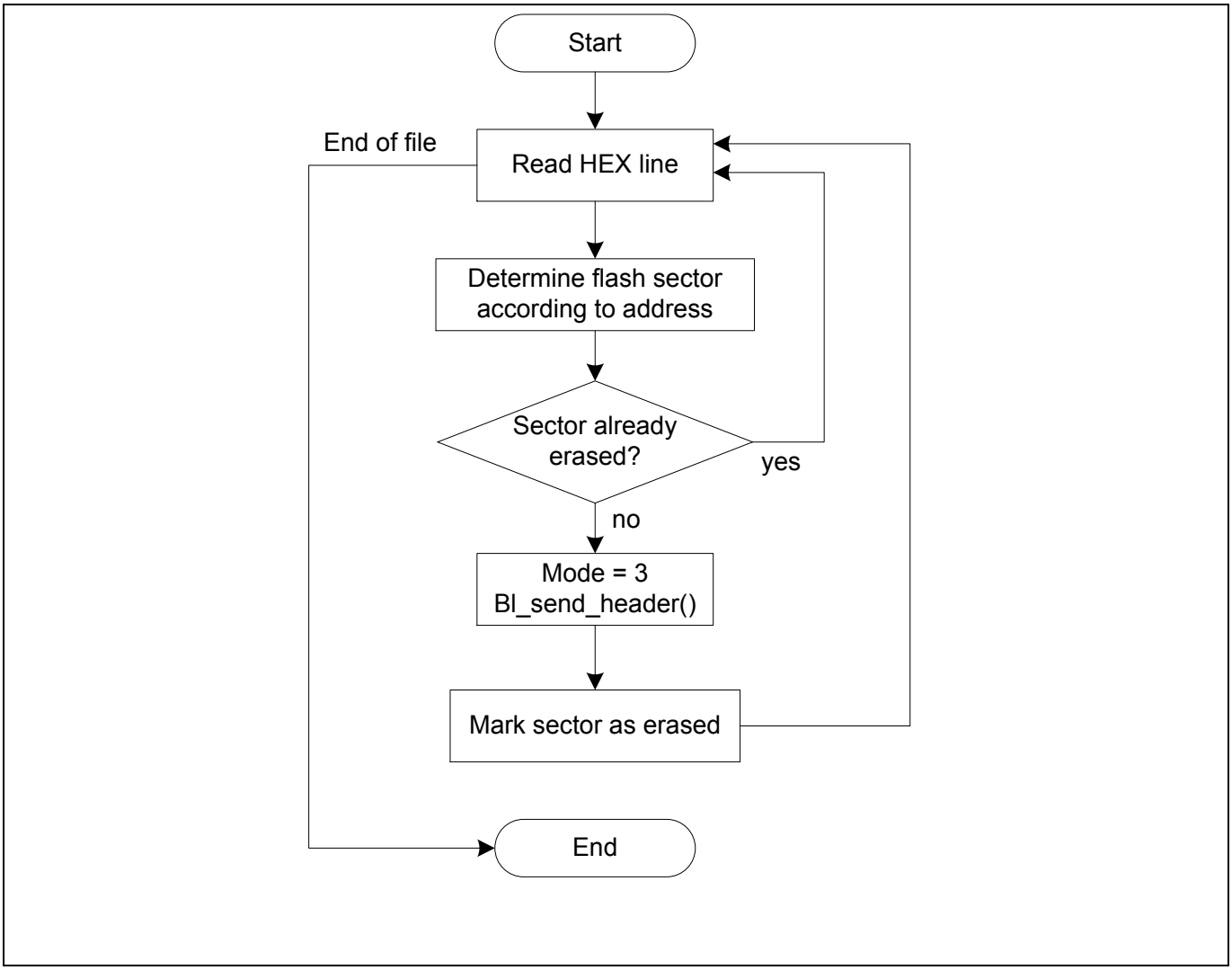

Figure 8 Flash erase procedure implemented in bl_erase_flash()

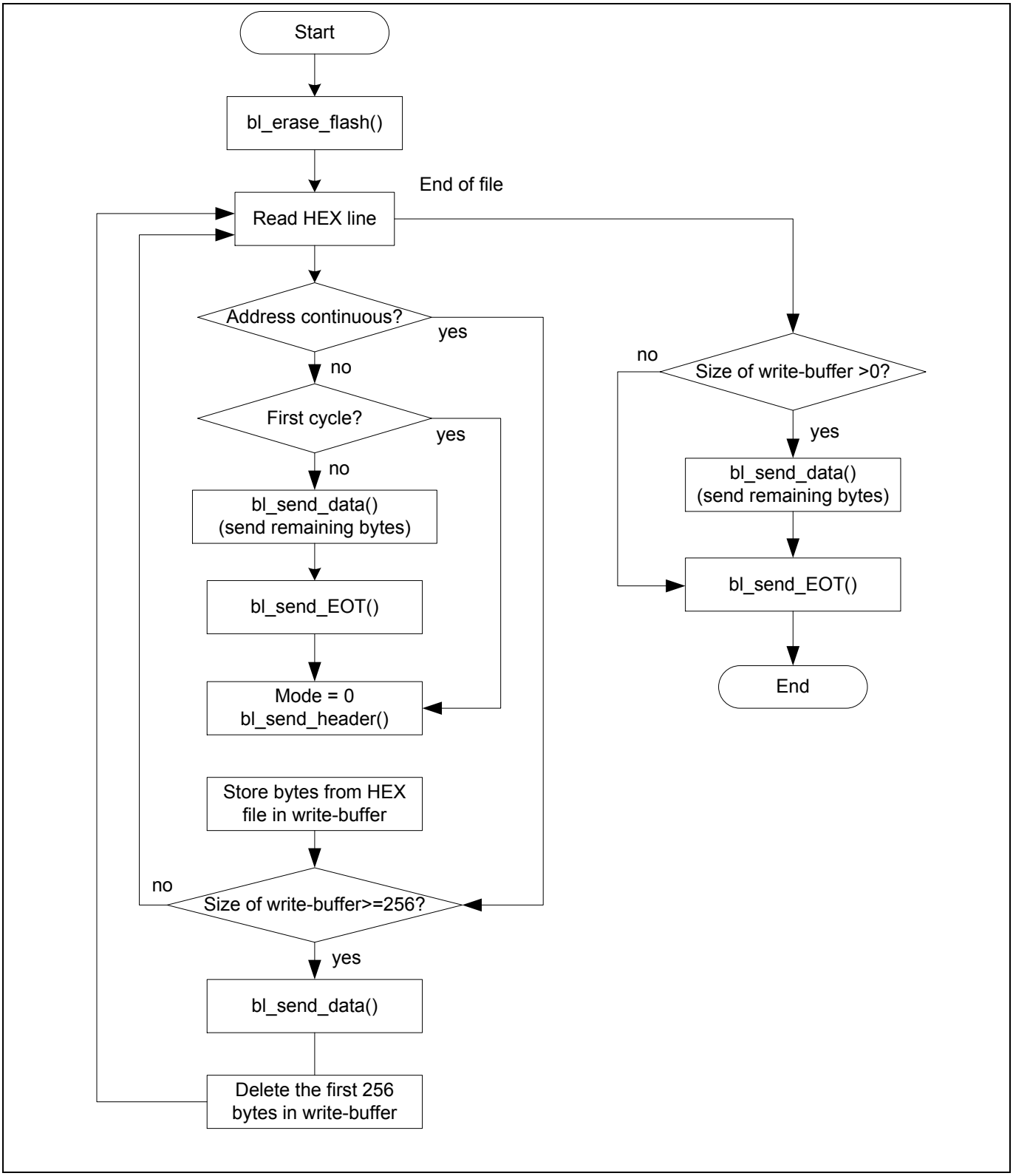

Figure 9 Flash programming procedure implemented in bl_download_flash()

5.1 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.

5.1.1 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 an 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.

5.1.2 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 the example in this application note, the sector address is fixed to partitions whose section address is stored in flash. Therefore, no section address is transmitted from the PC. Here, Byte2 contains the flash partition number.

5.2 Response code to the host

The flash loader program will inform the host whether a block has been successfully received and whether the requested flash routine has been successfully executed by sending out a response code as listed in Table 4.

Table 4 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

6 Usage of demonstrator

The example programs have been tested on the Infineon XMC1300 boot kit. The example program can be used to download user application code (hex file format) into flash as described below.

6.1 Hardware setup

The first step is to prepare a gateway controller using the XMC1300 kit before setting up the test hardware. To download the file “Gateway_XMC13.hex” into the XMC1300 gateway kit, users can simply open the DAVE™ project and download the code using the DAVE debugger.

The next step is to prepare the target board. Before connecting the target board with the PC, the following operations need to be performed in the XMC1300 target kit using one of flash loader tools such as Memtool or XMC™ flasher:

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 first 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 the XMC™ loader, BMI should be configured as User Mode (Debug) SWD_0.

The last step is to connect the gateway controller to the target board and Host PC, respectively. As shown in Figure 5, the target board is connected with the gateway via an SPI interface. Specifically, pins P1.0, P1.1, P0.8 and P0.9 on the target board should be connected to P1.0, P1.1, P0.8 and P0.9 pins on the gateway kit. The connection between the Host PC and gateway kit is via a USB cable.

Note: do not rebuild the SRAMCode project. If users want to rebuild the SRAMCode project in DAVE™ v4 IDE, they need to modify the first line of the SRAMCode.hex file so that the address points to 0x10012000, where the flash loader is stored. Replace the first line in the SRAMCode.hex file as shown below:

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

6.2 Demonstrator file structure

Figure 10 shows the file structure in the example programs.

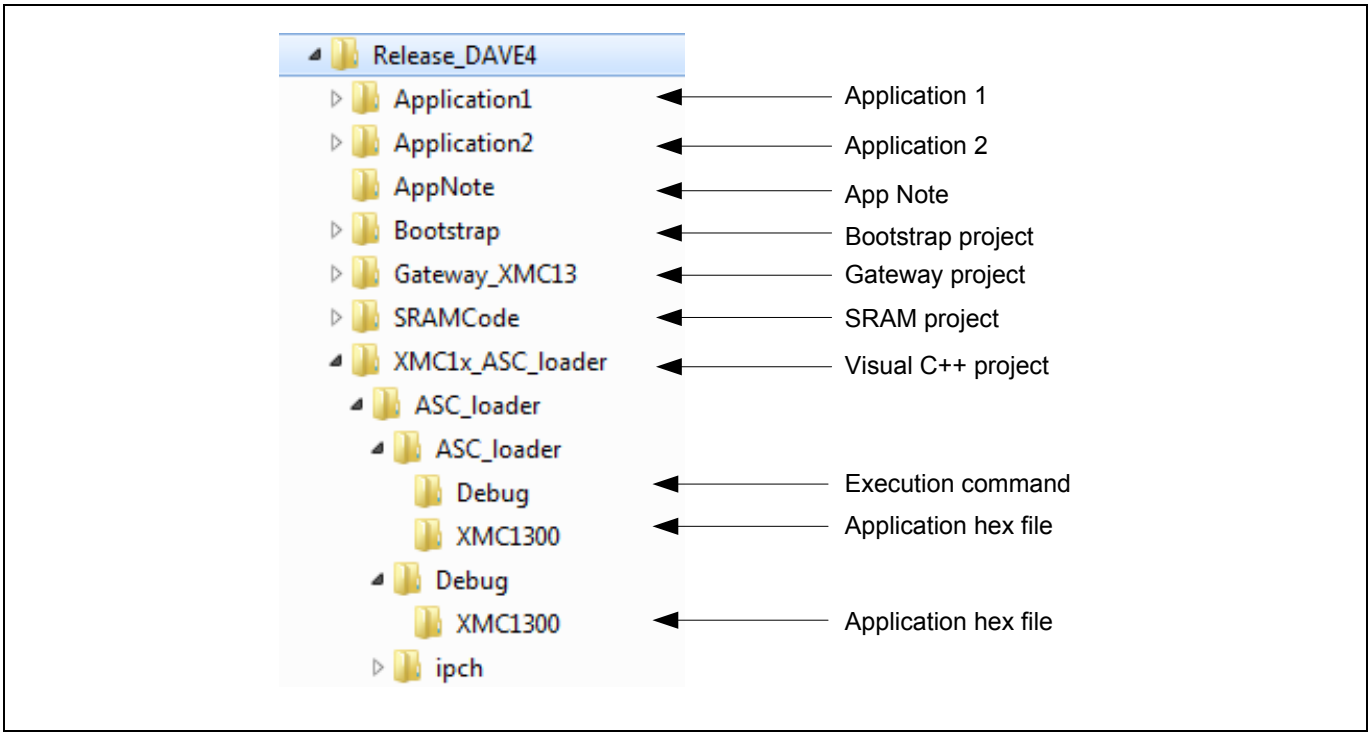

Figure 10 File structure of example programs

6.3 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 is shown in Figure 11:

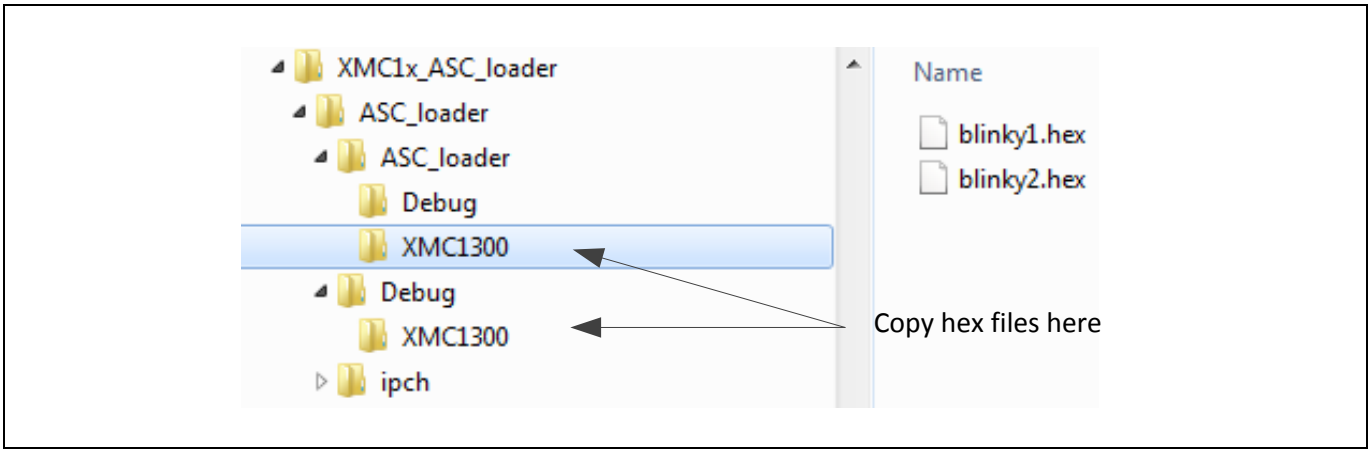

Figure 11 Location of object hex files to be flashed

There are two ways to start the demonstrator.

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

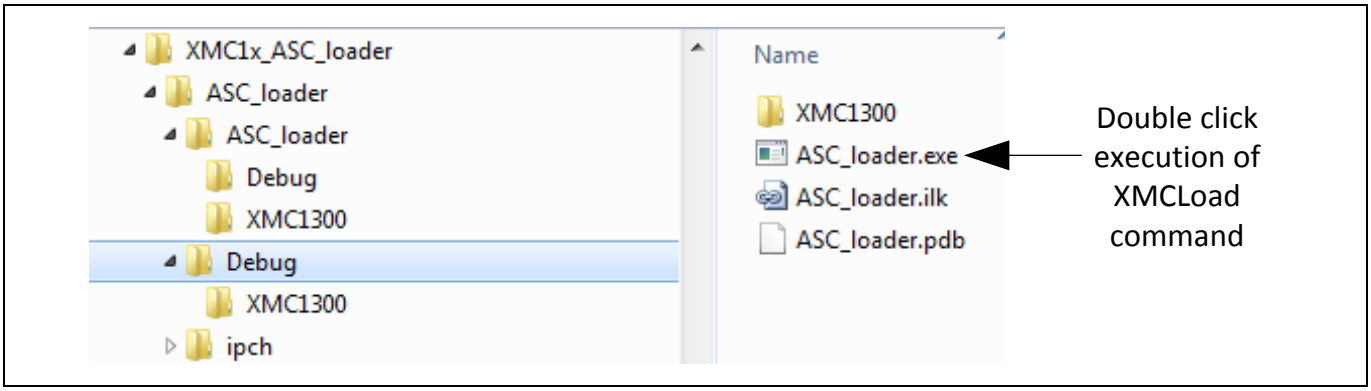

Figure 12 Direct start of demonstrator example

2. Double click the 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.

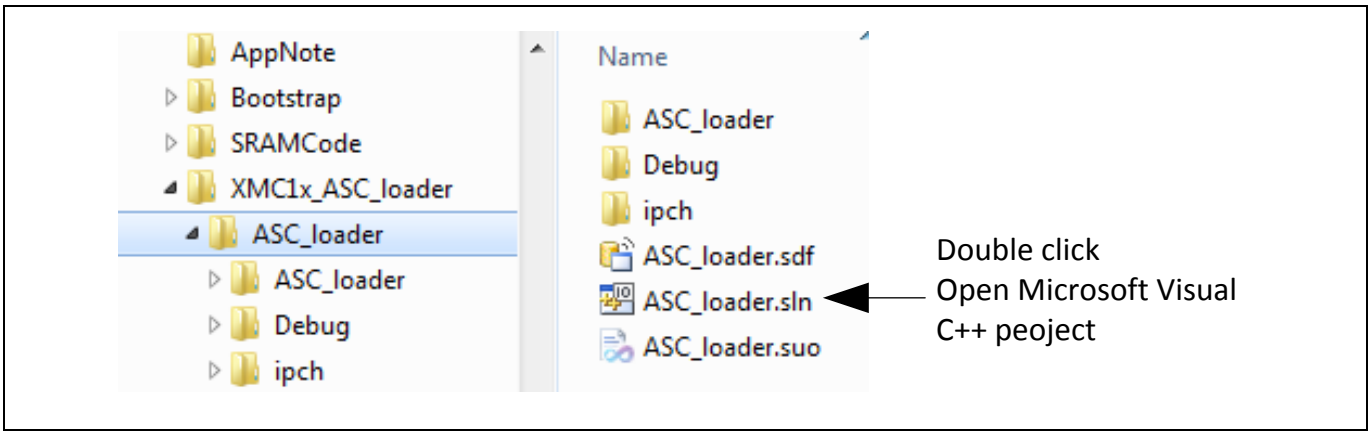

Figure 13 Start using Microsoft Visual project

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

On starting the demonstrator the following window is displayed:

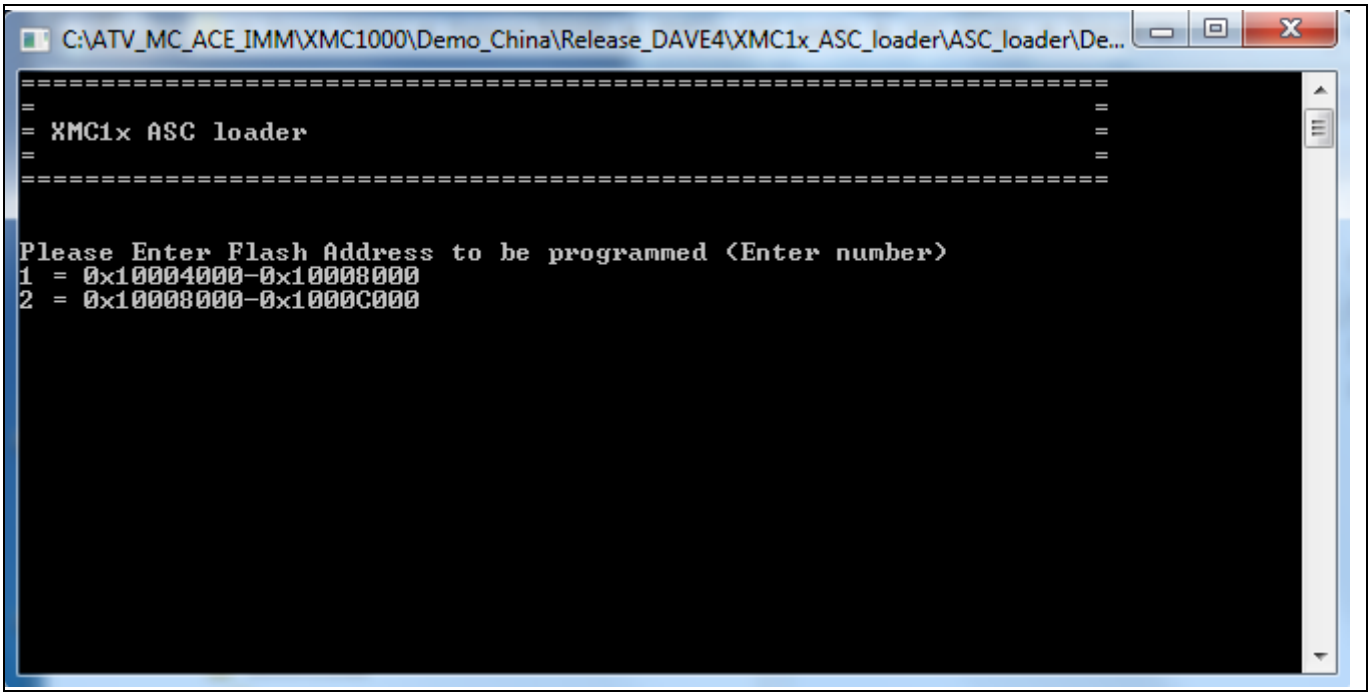

Figure 14 Start window from Visual Project

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

Note: The hex file name that will be programmed into flash must be complete and include 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 performed to return to the Bootstrap program. The application program that was just updated will be executed.

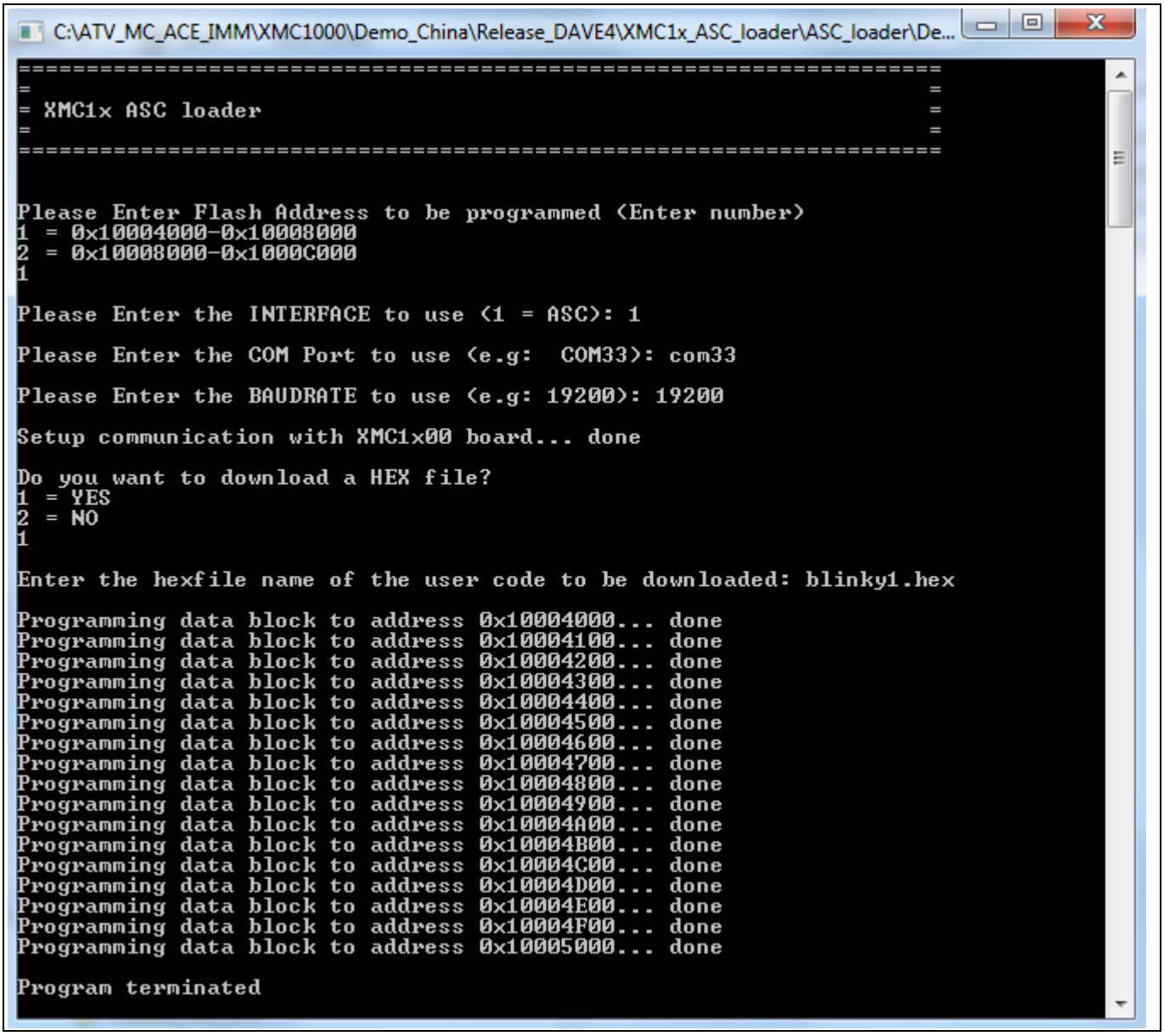

Figure 15 Window GUI illustrates the flash programing with application 1.

Note: To update application 2 just after application1 is successfully downloaded into flash, a power reset of the on-board-debugger (OBD) in the XMC1300 kit is required as the debugger probe (XMC4200) in the OBD must be reset for the next VCOM communication with the PC. However, if the USIC module in the XMC1300 device is used directly, this power reset is not necessary.

The mechanism to protect the currently used flash partition from update is built into the demonstrator example programs. The flash loader running in the device checks first to see if the selected flash partition from the host PC is currently used. If it is, 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 continues as shown in Figure 16.

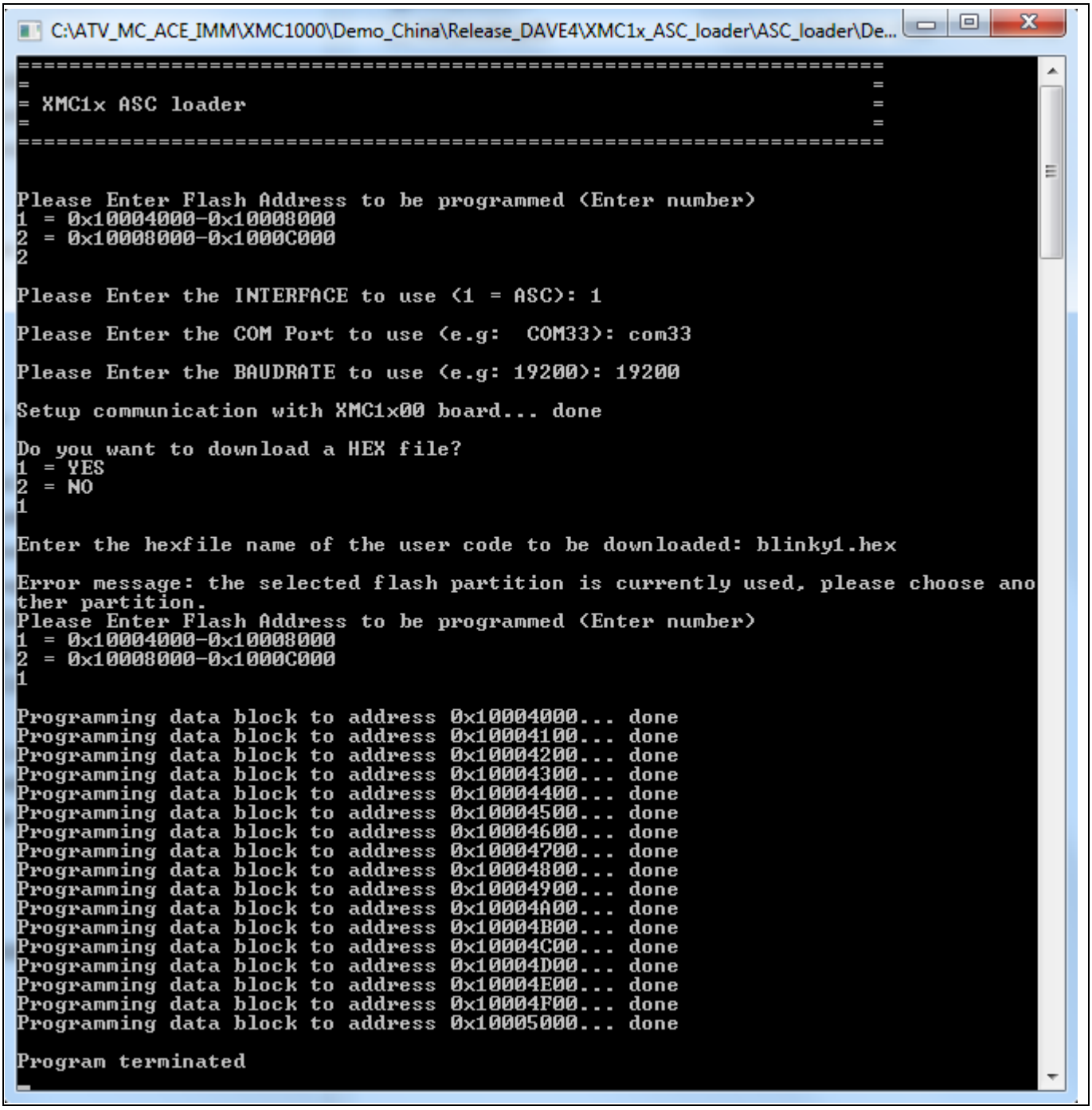

Figure 16 Protection of currently used flash partition

7 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
Software update of XMC1000 microcontroller using ASC interface	Application note for XMC1000	http://www.infineon.com/xmc1000/App

Revision history

Current Version is 1.0, 2016-08

Page or reference	Description of change
V1.0, 2016-08	
	Initial version

Edition 2016-10-18

Published by
Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?
Email: erratum@infineon.com

Document reference
AP32347

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.