Freescale Semiconductor
Application Note

Document Number:AN4370
Rev. 1, 2012

USB DFU Bootloader for MCUs

by: Paolo Alcantara

Contents

1 Introduction

Microcontroller (MCU) firmware upgrades on the field without using an external programming tool is a necessary feature these days. For Freescale MCUs supporting a USB device controller, the USB device firmware update (DFU) class is the solution. The USB DFU bootloader requires only a PC and a USB cable.

This document demonstrates how DFU fits in an embedded device and gives examples of implementation using a PC with Windows OS.

1.1 Audience

This document is intended to be used by all software development engineers, test engineers, and anyone who is implementing a USB DFU class or wants to use it as a solution.

1.2 Scope

This document presents information about USB DFU class implementation in Freescale MCUs such as S08 (JM60), ColdFire +(51JF), ColdFire (MCF52259) and Kinetis K and L family (K20, K40, K60, K70, LK25). Included within this document are details on:

- Running an MQX RTOS application
- Running a bare metal software
- How USB DFU can be ported to other platforms

2 Bootloader Overview

The USB DFU bootloader provides an easy and reliable way to load new user applications to devices that have the USB DFU bootloader preloaded.

After it is loaded, the new user application is able to run in the MCU. The USB DFU bootloader requires an application running on a PC. The DFU PC application supports loading the firmware to the device by using specific requests as stated in the USB DFU specification class.

The USB DFU bootloader is able to enumerate in two ways:

- USB composite device mode: Also known as run time mode. Formed by a DFU device plus another USB device class. For this implementation, the human interface device (HID) mouse device is used to avoid increasing the bootloader memory size. The MCU must be in the following conditions prior to enter to this mode:
  - MCU doesn't contain a valid firmware image or doesn't contain firmware.
  - An external action is applied to MCU, such as pressing a button during a reset event. This is dependent on the USB DFU bootloader implementation.
- DFU device mode: Used when DFU is ready to upload or download firmware images by a request made from the USB DFU PC application. Prior to this mode, the MCU was in USB composite device mode.

2.1 Bootloader example overview: ColdFire V2

A bootloader is a small application that is used to load new user applications to devices. Therefore, the bootloader needs to be able to run in both the user application and bootloader mode. As an example, Figure 1 describes the memory map of the ColdFire V2 bootloader implementation.

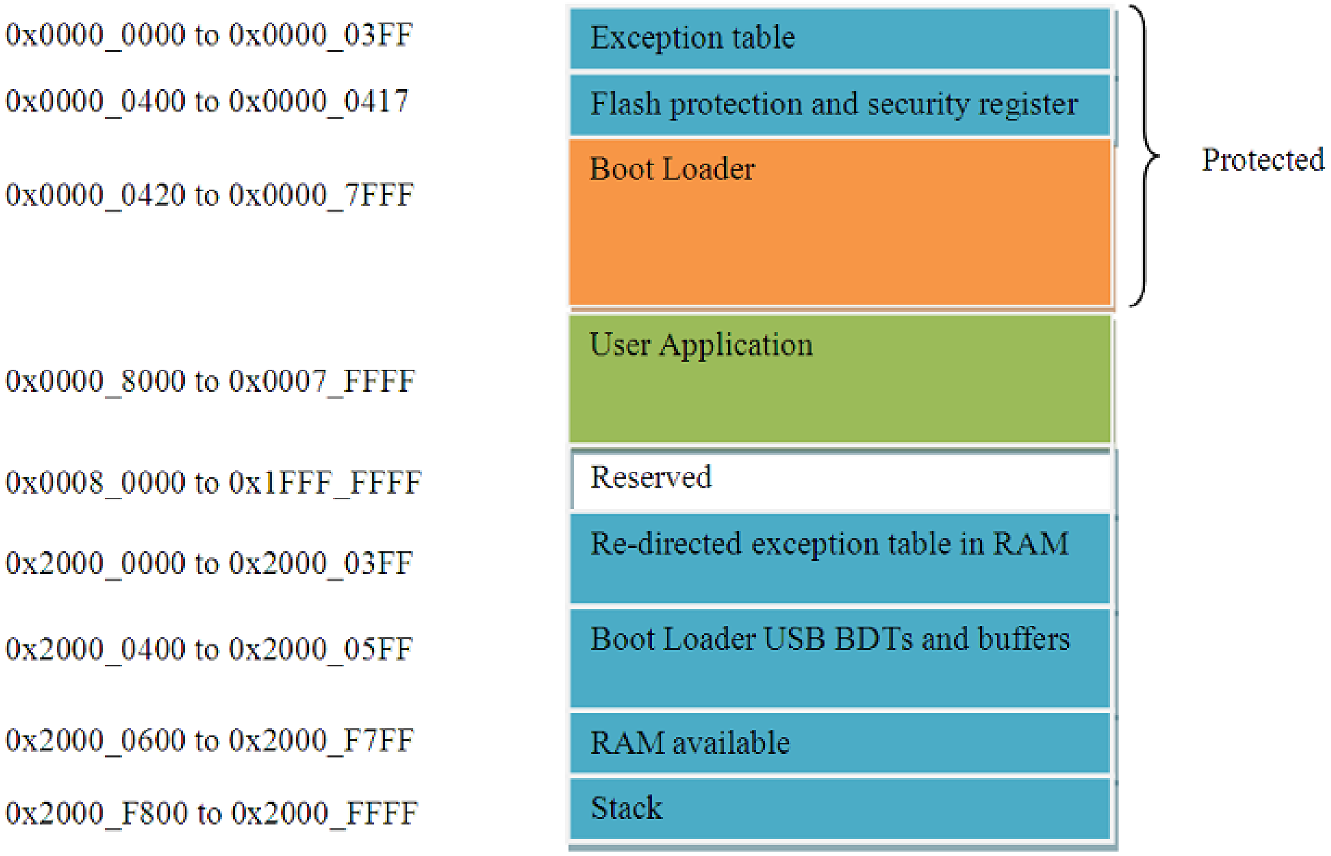

Figure 1. ColdFire V2 bootloader memory map

After reset, the device attempts to run the user application. If the user application is not found or corrupted, the device automatically runs into bootloader mode. If the application is valid and the user wants to run the bootloader program, external intervention is required, such as pressing a specific key at reset time to force the device to enter bootloader mode. The bootloader exception table is in flash memory area and used when bootloader runs. Thus, the bootloader cannot update its exception table when loading a new user application. If the user application requires interrupts, the user application exception table must be redirected to RAM.

The bootloader parses the user application image and flashes the image to flash memory in the user application area, as shown in Figure 1.

As shown in Figure 1, the bootloader holds the flash memory region from 0x0000_0000 to 0x0000_7FFF (32KB). This flash memory region needs to be program-protected to prevent corrupting the bootloader. The rest of flash memory, from 0x0000_8000 to 0x0007_FFFF (480 KB), is for the user application. After redirecting to RAM, the interrupt and exception table are in area from 0x2000_0000 to 0x2000_03FF (1 KB) of RAM memory.

While the user application is running, it can use the whole RAM memory, regardless of RAM space needed by the bootloader. Exception table space at RAM must not be considered for the user application's data space, neither .data nor .bss sections,by using the linker file.

The following table shows the space required by the DFU bootloader for different MCUs:

Table 1. DFU bootloader memory footprint

MCU	Bootloader flash memory required
CFV1, ColdFire+	40KB
CFV2	36KB

Table continues on the next page...

Table 1. DFU bootloader memory footprint (continued)

MCU	Bootloader flash memory required
Kinetis (L and K family)	40KB
S08	~21KB

3 Bootloader Architecture and Boot Sequence

The following section provides an overview of USB DFU bootloader architecture and its software flow.

3.1 Architecture overview

The architecture of USB DFU bootloader is shown in the following figure:

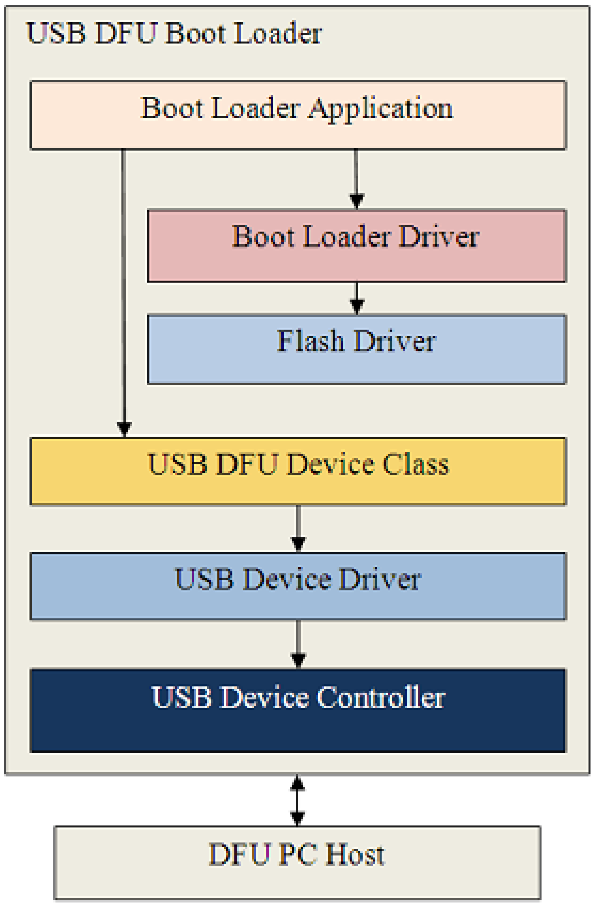

Figure 2. USB DFU bootloader architecture

The architecture of USB DFU bootloader contains the following functional blocks:

- Bootloader application: Controls the loading process. It uses specific requests in DFU class to receive and send firmware image files, then uses the bootloader driver to load the user application's files to and from the flash memory of the device.
- Bootloader driver: Parses firmware image files and flash them to flash memory. The bootloader driver supports parsing image files in CodeWarrior binary, S19, and raw binary file formats.
- Flash driver: Supports functions to erase, read, and write flash memory.
- USB DFU device class: Contains the API specified in DFU class.
- USB device driver and USB device controller: Communicate with the USB host (PC) through USB standard.

The USB DFU PC application supports features to download and upload firmware to and from the device.

3.2 Bootloader sequence

The bootloader is used to load an application that performs the product's main function. At reset, the bootloader is executed and does some simple checks to see if the application or bootloader mode can start. Once it's in DFU bootloader mode, the bootloader is able to receive requests from the USB DFU PC application. If the received request is to download firmware, the DFU bootloader accumulates the data in a buffer. When the buffer is full, it starts parsing the buffer and downloads it to user application region. See Figure 1 for details.

The flow of USB DFU bootloader is shown in the following flow chart:

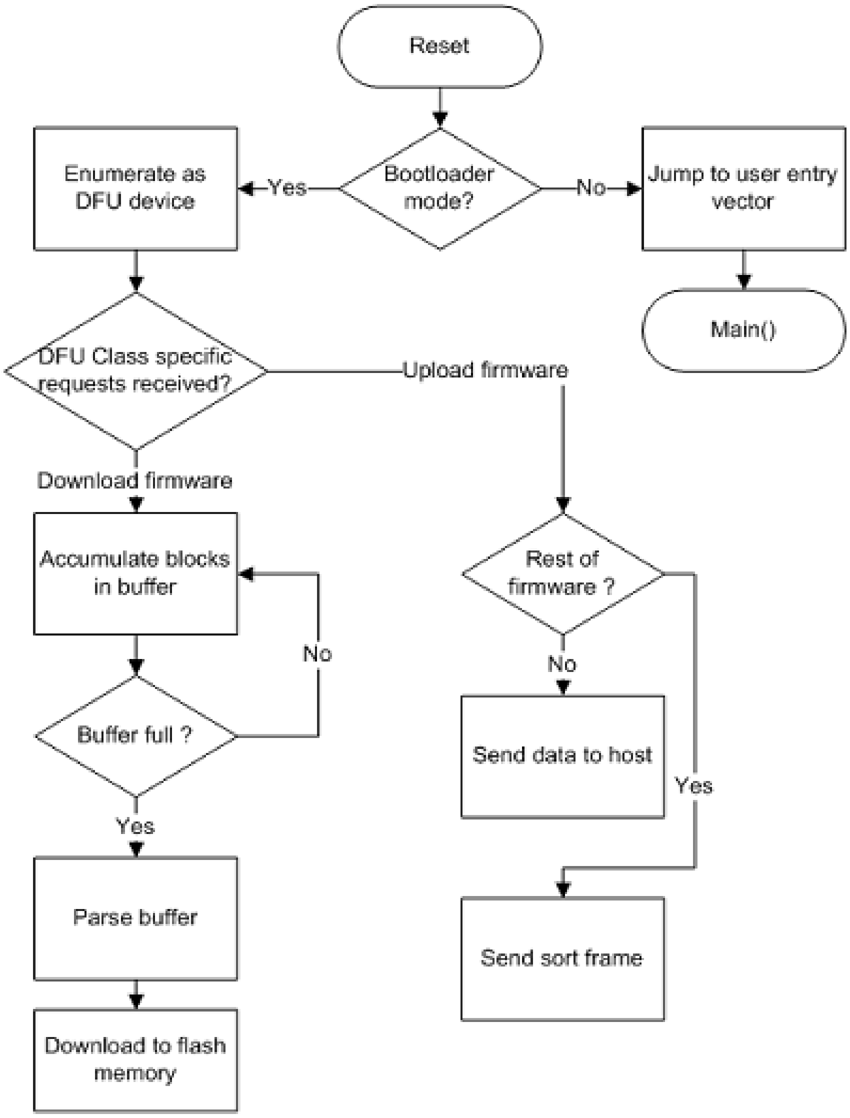

Figure 3. USB DFU bootloader sequence

4 Develop Applications with Bootloader

The following section describes how to modify user applications to be used by the USB DFU bootloader.

4.1 Linker files modifications

Normally, an application will be located at the beginning of flash memory. However, the bootloader needs a flash memory space, therefore the user application must be placed in the rest of flash memory. See Figure 1 for details.

Because of this, the user application linker file must be modified to locate the application at a specific memory region.

The following sections explain the linker file changes needed for ColdFire V1, ColdFire+, ColdFire V2-4, Kinetis (K and L family), and S08 MCUs.

4.1.1 CFV1 linker file: ColdFire V1 and ColdFire+

A normal CFV1 linker file is shown as follows:

```
# Sample Linker Command File for CodeWarrior for ColdFire MCF51JM128

# Memory ranges

MEMORY {
   code        (RX)  : ORIGIN = 0x00000410, LENGTH = 0x0001FBF0
   userram     (RWX) : ORIGIN = 0x00800000, LENGTH = 0x00004000
}
```

To run with the USB DFU bootloader, the user application must indicate that flash memory area starts at address 0x0000_A000. The modified linker file is as follows:

```
# Sample Linker Command File for CodeWarrior for ColdFire MCF51JM128

# Memory ranges

MEMORY {
   code        (RX)  : ORIGIN = 0x0000A410, LENGTH = 0x00017BF0
   userram     (RWX) : ORIGIN = 0x00800000, LENGTH = 0x00004000
}
```

4.1.2 CFV2 linker file: ColdFire V2-4

A normal CFV2 linker file is shown as follows:

```
# Sample Linker Command File for CodeWarrior for ColdFire

KEEP_SECTION {.vectortable}

# Memory ranges

MEMORY {
   vectorrom   (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000400
   cfmprotrom  (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000020
   code        (RX)  : ORIGIN = 0x00000500, LENGTH = 0x0007FB00
   vectorram   (RWX) : ORIGIN = 0x20000000, LENGTH = 0x00000400
   userram     (RWX) : ORIGIN = 0x20000400, LENGTH = 0x00005C00
}
```

To run with the USB DFU bootloader, the user application must indicate that flash memory area starts at address 0x0000_9000. The modified linker file is as follows:

```
# Sample Linker Command File for CodeWarrior for ColdFire

KEEP_SECTION {.vectortable}

# Memory ranges

MEMORY {
   vectorrom   (RX)  : ORIGIN = 0x00009000, LENGTH = 0x00000400
   cfmprotrom  (RX)  : ORIGIN = 0x00009400, LENGTH = 0x00000020
   code        (RX)  : ORIGIN = 0x00009500, LENGTH = 0x00077B00
   vectorram   (RWX) : ORIGIN = 0x20000000, LENGTH = 0x00000400
   userram     (RWX) : ORIGIN = 0x20000400, LENGTH = 0x00005C00
}
```

4.1.3 Kinetis (K and L family) linker file

A normal Kinetis linker file is shown as follows:

```
MEMORY
{
   vectorrom    (RX): ORIGIN = 0x00000000, LENGTH = 0x00000400
   cfmprotrom   (RX): ORIGIN = 0x00000400, LENGTH = 0x00000020
   rom          (RX): ORIGIN = 0x00000420, LENGTH = 0x0001FBE0  # Code + Const data
   ram          (RW): ORIGIN = 0x00800000, LENGTH = 0x00004000  # SRAM - RW data
}
```

To run with the USB DFU bootloader, the user application must indicate that flash memory area starts at address 0x0000_A000. The modified linker file is as follows:

```
MEMORY
{
   vectorrom    (RX): ORIGIN = 0x0000A000, LENGTH = 0x00000400
   cfmprotrom   (RX): ORIGIN = 0x0000A400, LENGTH = 0x00000020
   rom          (RX): ORIGIN = 0x0000A420, LENGTH = 0x00017BE0  # Code + Const data
   ram          (RW): ORIGIN = 0x00800000, LENGTH = 0x00004000  # SRAM - RW data
}
```

4.1.4 S08 linker file

A normal S08 linker file is shown as follows:

```
SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */
    Z_RAM                    =  READ_WRITE   0x00B0 TO 0x00FF;
    RAM                      =  READ_WRITE   0x0100 TO 0x10AF;
    RAM1                     =  READ_WRITE   0x1860 TO 0x195F;
    ROM                      =  READ_ONLY    0x1960 TO 0xFFAD;
    ROM1                     =  READ_ONLY    0x10B0 TO 0x17FF;
    ROM2                     =  READ_ONLY    0xFFC0 TO 0xFFC3;
```

To run with the USB DFU bootloader, the user application must indicate that flash memory area ends at address 0xABA5. The modified linker file is as follows:

```
SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */

    // Application Segments
    Z_RAM                    =  READ_WRITE   0x00B0 TO 0x00FF;
    RAM             =  READ_WRITE   0x0110 TO 0x10AF;
    RAM1      =  READ_WRITE   0x1860 TO 0x195F;
    ROM                      =  READ_ONLY    0x1960 TO 0xABA5;
    ROM1                     =  READ_ONLY    0x10B0 TO 0x17FF;
    ROM2                     =  READ_ONLY    0xFFC0 TO 0xFFC3;
```

NOTE

For CFV1, CFV2, ColdFire+, and Kinetis(L and K family) linker files, the start of the user application data space matches the start of MCU RAM. During exception table relocation, explained in Exception table redirection, the declared RAM exception table space is reserved by the compiler, and then no other data (.data nor .bss) shares this space.

4.2 Exception table redirection

The exception vectors are located by default in flash memory and used by the bootloader, so the bootloader cannot update it when loading new user applications.

If the user application needs interrupts, then the exception table must be redirected to RAM, except for S08 MCUs.

The procedure to redirect the exception table to RAM is different for each MCU.

The following section describes how the exception table is redirected in a MQX and a bare metal user application.

4.2.1 MQX user application

The MQX RTOS can redirect the exception table to RAM by using the C-language macro `MQX_ROM_VECTORS` contained in userconfig.h.

The following example source code shows how to assign the value of 0 to the `MQX_ROM_VECTORS` macro.

```
#define MQX_ROM_VECTORS    0 //1=ROM (default), 0=RAM vector
```

NOTE

MQX RTOS only supports ColdFire, ColdFire+, and Kinetis MCUs. An 8-bit MCU must use a bare metal application instead.

4.2.2 Bare metal user application

The following sections describe how to redirect exception table to RAM for ColdFire V1, ColdFire+, ColdFire V2-4, Kinetis, and S08 MCUs.

4.2.2.1 CFV1 MCU: ColdFire V1 and ColdFire+

CFV1 MCU has a CPU-register named Vector Base Register (VBR) containing the base address of the exception vector table. This register can be used to relocate the exception table from its default position in the flash memory (address 0x0000_0000) to the base of the RAM (0x0080_0000).

Declaring an interrupt service routine (ISR) inside the application source code is different when using a bootloader.

The exception table redirection procedure can be summarized as follows:

1. Declare an exception table within the user application code area and assign ISRs at this space.
2. Reserve an exception table space at user application data area. It must be at the start of RAM space.
3. At runtime, copy the declared exception table to the reserved exception table space.
4. Write to VBR with the address of the reserved exception table which is the start of RAM space.

The new exception table must be declared as shown in the following lines. To add a new ISR, the address vector of the dummy_ISR must be replaced with the name of the new ISR. The address of this new exception table must be part of the user application code space. This example is declared at address 0x0000_A000. See Figure 1 for details. The new exception table in the user application is declared as follows:

```
void  (* const RAM_Vector[])()@0x0000A000=
{
(pFun)&dummy_ISR,             // vector_0  INITSP
(pFun)&dummy_ISR,             // vector_1  INITPC
......
(pFun)&dummy_ISR,             // vector_67 Vspi1
(pFun)&dummy_ISR,             // vector_68 Vspi2
(pFun)&dummy_ISR,             // vector_69 Vusb
(pFun)&dummy_ISR,             // vector_70 VReserved70
```

```
(pFun)&dummy_ISR,              // vector_71 Vtpm1ch0
(pFun)&dummy_ISR,              // vector_72 Vtpm1ch1
(pFun)&dummy_ISR,              // vector_73 Vtpm1ch2
……
}
```

Next, the declared exception table (RAM_Vector) must be copied to the base of RAM at runtime. The following source code performs this task:

```
pdst=(dword)&New_RAM_vector;//0x00800000;//RAM base address
psrc=(dword)&RAM_vector;

for (i=0;i<111;i++,pdst++,psrc++)//112 exceptions
  {
    *pdst=*psrc;
  }
```

Finally, the following software is used to redirect the exception table to RAM with address 0x0080_0000:

```
asm (move.l  #0x00800000,d0);
asm (movec  d0,vbr);
```

4.2.2.2 CFV2 MCU: ColdFire V2-4

Similarly to CFV1, the exception table must be copied from the user application space to RAM at runtime. The following source code shows the `initialize_exceptions` function which copy from user application space (FLASH) to RAM base address:

```
void initialize_exceptions(void)
{
/*
 * Memory map definitions from linker command files used by mcf5xxx_startup
 */

register uint32 n;

/*
     * Copy the vector table to RAM
     */
if (__VECTOR_RAM != (unsigned long*)_vect)
{
for (n = 0; n < 256; n++)
__VECTOR_RAM[n] = (unsigned long)_vect[n];
}
mcf5xxx_wr_vbr((unsigned long)__VECTOR_RAM);
}
```

Using CFV2 version, Freescale USB Stack with PHDC v3.0 also supports the `initialize_exceptions` function to copy the interrupt exception table to the specified area in RAM.

```
void initialize_exceptions(void);
```

The `initialize_exceptions` function copies the interrupt vector table to the RAM area at *__VECTOR_RAM* address. This address needs to be defined at linker file.

If using USB Stack with PHDC v3.0 as the user application project template, the `initialize_exceptions` function is called at startup by default.

4.2.2.3 Kinetis (L and K family) MCU

For Kinetis MCU, the SCB_VTOR register contains the base address of the exception table. To redirect the exception table, the exception table must be copied to RAM. Then SCB_VTOR must be written with the value of the copied address.

The following steps explain in more detail how the redirection must be performed in Kinetis.

1. Declare a ROM area to store the exception table (linker file).

```
.interrupts :
{
  ___VECTOR_ROM = .;
  * (.vectortable)
  . = ALIGN (0x4);
} > interrupts
```

2. Copy the exception table from default user application code space to RAM address, aligned to 128 bytes.

```
extern uint_32 ___VECTOR_RAM[];
extern uint_32 ___VECTOR_ROM[];            //Get vector table in ROM

uint_32 i,n;
/* Copy the vector table to RAM */
if (___VECTOR_RAM != ___VECTOR_ROM)
{
for (n = 0; n < 0x410/4; n++)
___VECTOR_RAM[n] = ___VECTOR_ROM[n];
}
/* Point the VTOR to the new copy of the vector table */
SCB_VTOR = (uint_32)___VECTOR_RAM;
```

4.2.2.4 S08 MCU

The MC9S08 core cannot re-direct the exception table to the RAM like ColdFire or Kinetis. Instead, the bootloader points to the exception table of the application at a re-directed exception table in the user application space.

The re-directed exception table is stored at a specific address. The user application must declare a function pointer to the exception table at the specific address to implement interrupts.

For the DFU bootloader, the array UserJumpVectors is the function pointer to the exception table, and it starts at address VectorAddressTableAddress, which is 0xABA6 according to S08 specifications.

```
// User Interrupt Jump Vector Table
volatile const Addr UserJumpVectors[InterruptVectorsNum]@ VectorAddressTableAddress = {
    Dummy_ISR,                  //  0 - Reset
    Dummy_ISR,                  //  1 - SWI
    IRQ_ISR,                    //  2 - IRQ
    Dummy_ISR,                  //  3 - Low Voltage Detect
    Dummy_ISR,                  //  4 - MCG Loss of Lock
    Dummy_ISR,                  //  5 - SPI1
    Dummy_ISR,                  //  6 - SPI2
    USB_ISR,                    //  7 - USB Status
    Dummy_ISR,                  //  8 - Reserved
    Dummy_ISR,                  //  9 - TPM1 Channel0
    Dummy_ISR,                  //  10 - TPM1 Channel1
    Dummy_ISR,                  //  11 - TPM1 Channel2
    Dummy_ISR,                  //  12 - TPM1 Channel3
    Dummy_ISR,                  //  13 - TPM1 Channel4
    Dummy_ISR,                  //  14 - TPM1 Channel5
    Dummy_ISR,                  //  15 - TPM1 Overflow
    Dummy_ISR,                  //  16 - TPM2 Channel0
    Dummy_ISR,                  //  17 - TPM2 Channel1
    Dummy_ISR,                  //  18 - TPM2 Overflow
    Dummy_ISR,                  //  19 - TPM1 SCI1 Error
    Dummy_ISR,                  //  20 - TPM1 SCI1 Receive
    Dummy_ISR,                  //  21 - TPM1 SCI1 Transmit
    Dummy_ISR,                  //  22 - TPM1 SCI2 Error
    Dummy_ISR,                  //  23 - TPM1 SCI2 Receive
    Dummy_ISR,                  //  24 - TPM1 SCI2 Transmit
    Kbi_ISR,                    //  25 - TPM1 KBI
    Dummy_ISR,                  //  26 - TPM1 ADC Conversion
    Dummy_ISR,                  //  27 - TPM1 ACMP
```

```
    Dummy_ISR,                  //  28 - IIC
    Timer_ISR,                  //  29 - RTC
};
```

The *Addr* is function pointer type as follows:

```
typedef void (* Addr)(void);
```

The bootloader uses the array BootIntVectors in the file Redirect_Vectors_S08.c to load the interrupt vector table in the bootloader flash.

```
volatile const Addr BootISRTable[InterruptVectorsNum] = {
    Dummy_ISR,                  //  0 - Reset
    Dummy_ISR,                  //  1 - SWI
    Dummy_ISR,                  //  2 - IRQ
    Dummy_ISR,                  //  3 - Low Voltage Detect
    Dummy_ISR,                  //  4 - MCG Loss of Lock
    Dummy_ISR,                  //  5 - SPI1
    Dummy_ISR,                  //  6 - SPI2
    USB_ISR,                    //  7 - USB Status
    Dummy_ISR,                  //  8 - Reserved
    Dummy_ISR,                  //  9 - TPM1 Channel0
    Dummy_ISR,                  //  10 - TPM1 Channel1
    Dummy_ISR,                  //  11 - TPM1 Channel2
    Dummy_ISR,                  //  12 - TPM1 Channel3
    Dummy_ISR,                  //  13 - TPM1 Channel4
    Dummy_ISR,                  //  14 - TPM1 Channel5
    Dummy_ISR,                  //  15 - TPM1 Overflow
    Dummy_ISR,                  //  16 - TPM2 Channel0
    Dummy_ISR,                  //  17 - TPM2 Channel1
    Dummy_ISR,                  //  18 - TPM2 Overflow
    Dummy_ISR,                  //  19 - TPM1 SCI1 Error
    Dummy_ISR,                  //  20 - TPM1 SCI1 Receive
    Dummy_ISR,                  //  21 - TPM1 SCI1 Transmit
    Dummy_ISR,                  //  22 - TPM1 SCI2 Error
    Dummy_ISR,                  //  23 - TPM1 SCI2 Receive
    Dummy_ISR,                  //  24 - TPM1 SCI2 Transmit
    Dummy_ISR,                  //  25 - TPM1 KBI
    Dummy_ISR,                  //  26 - TPM1 ADC Conversion
    Dummy_ISR,                  //  27 - TPM1 ACMP
    Dummy_ISR,                  //  28 - IIC
    Dummy_ISR,                  //  29 - RTC
};
```

The file Redirect_Vectors_S08.c contains functions to determine whether to call interrupt functions of bootloader or user application. When an interrupt occurs, the associated interrupt function in file Redirect_Vectors_S08.c is called, and then the function determines whether to call interrupt function of bootloader or user application.

```
extern uint_8 boot_mode;
/* VectorNumber_Vswi */
interrupt VectorNumber_Vswi vector1(void)
{
    if(boot_mode == BOOT_MODE)
    {
         BootISRTable[VectorNumber_Vswi]();
    }
    else
    {
         AppISRTable[VectorNumber_Vswi]();
    }
}
```

For a new application, the files Bootloader.h and Vectortable.c must be added to the application project. Then, load the array UserJumpVectors in Vectortable.c with the proper application ISRs.

5 Bootloader Example: Boot MQX

The following section explains how to use the USB DFU bootloader with a MQX boot example. The example uses an M52259EVB board and CodeWarrior version 7.2.

5.1 Preparing the setup

The DFU bootloader requires a software and hardware configuration. The following sections describe the steps to run the bootloader example in MQX.

5.1.1 Software requirements

The following software is required to run the DFU application:

- DFU PC host application
- CodeWarrior version 7.2
- Serial terminal

Details about how to use these PC applications are explained in the following sections.

5.1.2 Hardware setup

The following hardware is required:

- A PC running Windows XP, Windows Vista, or Windows 7 in 32-bit or 64-bit edition
- A M52259EVB board and +5V power supply
- Two USB cables:
  - USB 2.0 A-B
  - USB 2.0 A to miniB
- A DB9 cable or USB2SER converter

The hardware must be configured as follows:

1. Connect the power supply to the board.
2. Connect the USB debug port of the board to the PC using the USB 2.0 A-B cable.
3. Connect the MCF52259EVB COM1 port to the PC with a DB9 cable or using a USB2SER converter.
4. Turn the board power on.

5.2 Preparing the firmware image file

The following steps must be followed to generate a valid MQX image for the USB DFU bootloader:

1. Set MQX_ROM_VECTORS to 0 in user_config.h file to use the exception table from RAM

   ```
   #define MQX_ROM_VECTORS     0
   ```
2. Build libraries of MQX by running Freescale MQX 3.7.0\config\m52259evb\cwcf72\build_m52259evb_libs.mcp projects. If using CW10.x, build each library individually (bsp_m52259evb, psp_m52259evb, etc) as listed in the next figure.

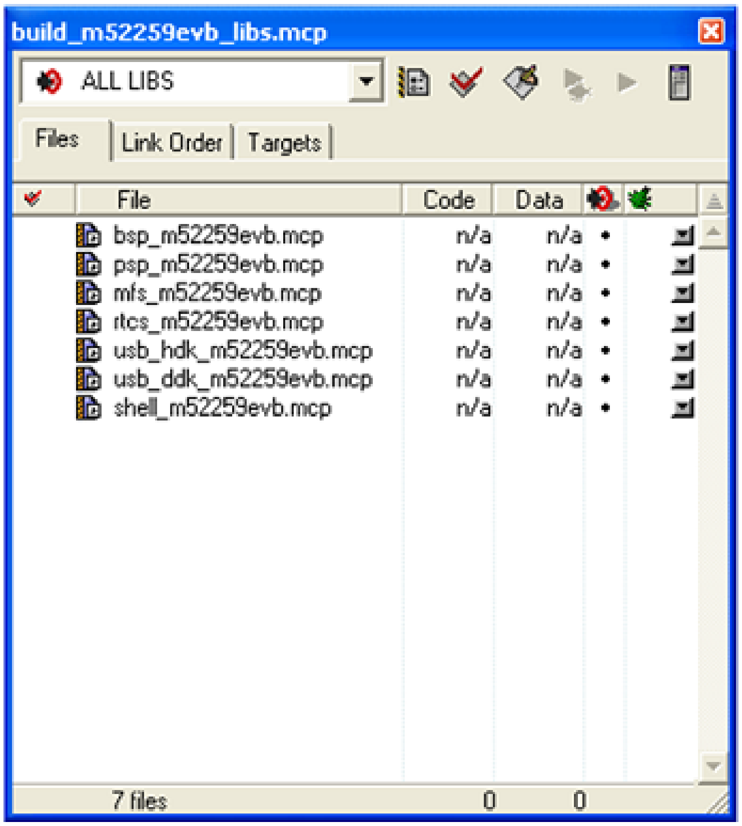

Figure 4. Build MQX libraries

3. Create an MQX application. As a test for this section, project “Freescale MQX 3.7.0\mfs\examples\mfs_usb” is used.
4. Select “Flash Debug” or “Flash Release” target.

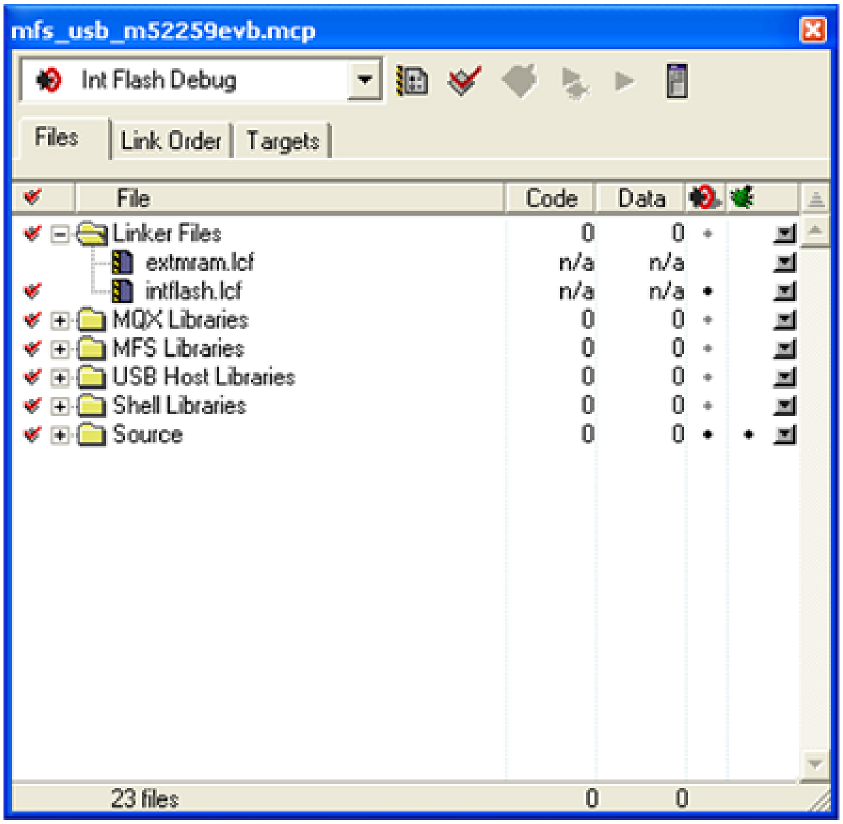

Figure 5. MQX example

5. Modify the intflash.lcf linker file to move the code section (vectorrom, cfmprotrom and rom memory segments) to the user application region of the USB DFU bootloader. The user application region starts at 0x0000_9000.

```
vectorrom   (RX): ORIGIN = 0x00009000, LENGTH = 0x00000400
cfmprotrom  (RX): ORIGIN = 0x00009400, LENGTH = 0x00000020
rom         (RX): ORIGIN = 0x00009420, LENGTH = 0x00075BE0 # Code+Const
data
```

6. Configure project to generate s19 and binary image files. These are valid file formats for the USB DFU PC application.

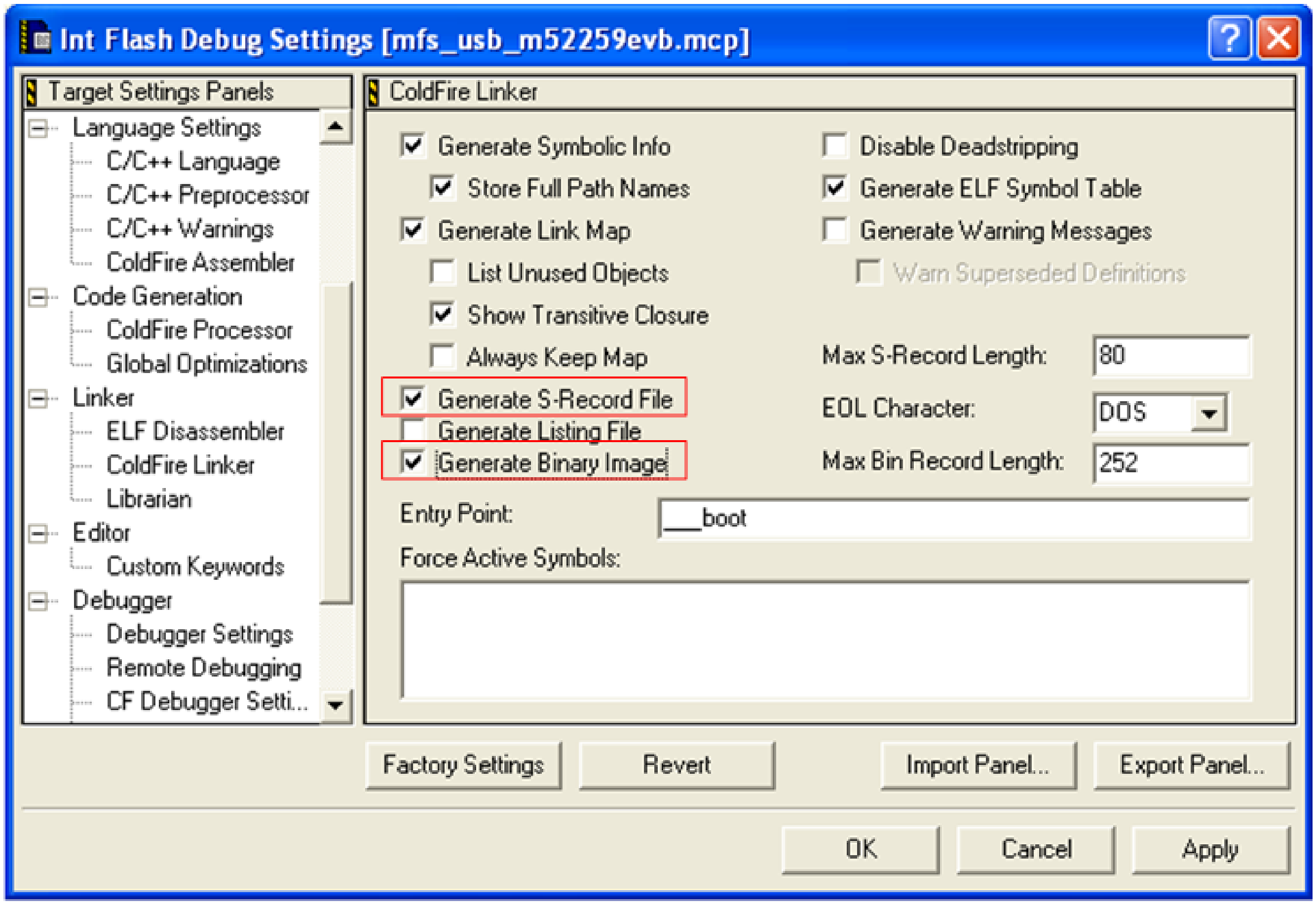

Figure 6. Options to generate s19 and binary firmware image

7. Build user application. After build process, the m52259evb folder contains two valid file formats:
    - intflash_d.elf.S19
    - intflash_d.elf.bin

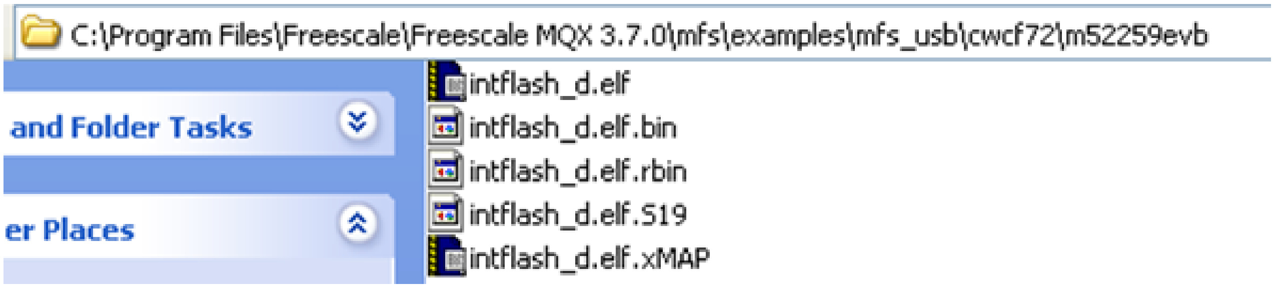

Figure 7. Firmware image files

The generated s19 file has the start address at 0x0000_9000.

8. The s19 and binary files from previous step will be used in Downloading firmware.

5.3 Building the application

1. Open the USB DFU bootloader project for the M52259EVB platform on CodeWarrior version 7.2 IDE and build it. The mcp file is found at the following path:
   - \Source\Device\app\dfu_bootloader\codewarrior\cfv2usbm52259
   - Or using CW10.2: Source\Device\app\dfu_bootloader\cw10\cfv2usbm52259
2. Load the project to MCF52259 flash memory by using CodeWarrior Flash Programmer utility.

5.4 Running the application

The following section describes how to install the USB DFU bootloader device in a PC running Windows OS.

The test firmware used in section 5.2 uses the serial terminal to communicate with the user. Open a Serial Console at 115.2Kbps 8-N-1 No flow control.

5.4.1 Driver installation

The USB DFU PC Application uses WinUSB 2.0. WinUSB is a generic USB driver provided by Microsoft. To install the USB DFU bootloader device driver:

1. Reset the M52259EVB and connect to the PC by using USB 2.0 A to miniB cable. Direct connection of the USB cable to the PC's USB port is strongly advised. Windows asks for the USB driver to use with the new device. A Found New Hardware window appears as shown in next figure.

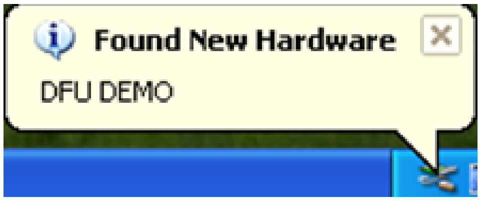

Figure 8. Found new hardware

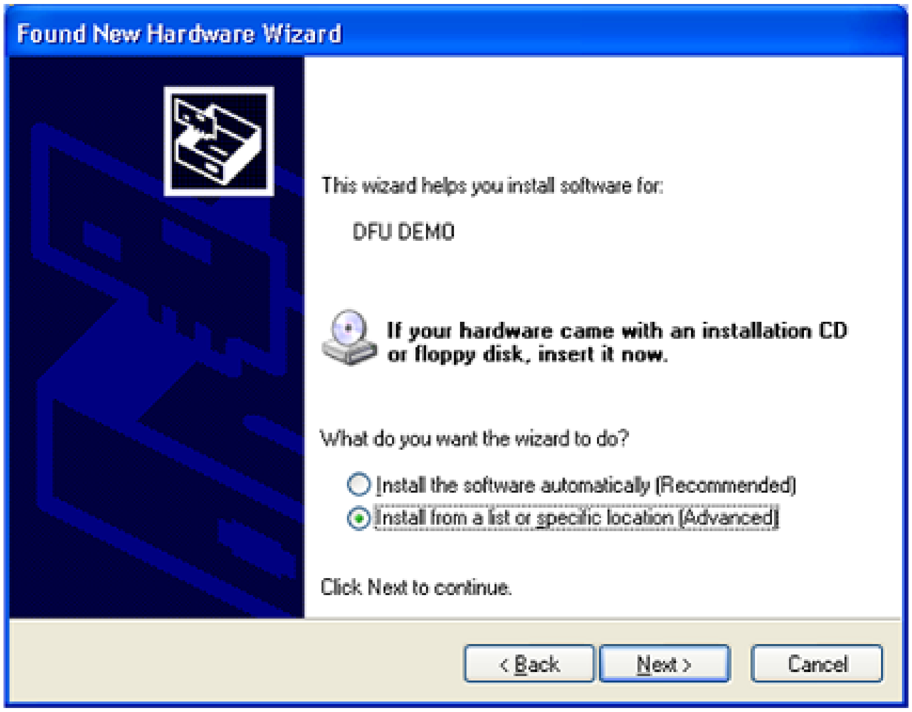

Figure 9. Found new hardware wizard

2. Select “Install from a list or specific location (Advanced)” option and click on the “Next” button. The next figure shows the current message shown by Windows. Select “Don’t search, I will choose the driver to install” option and click “Next.”

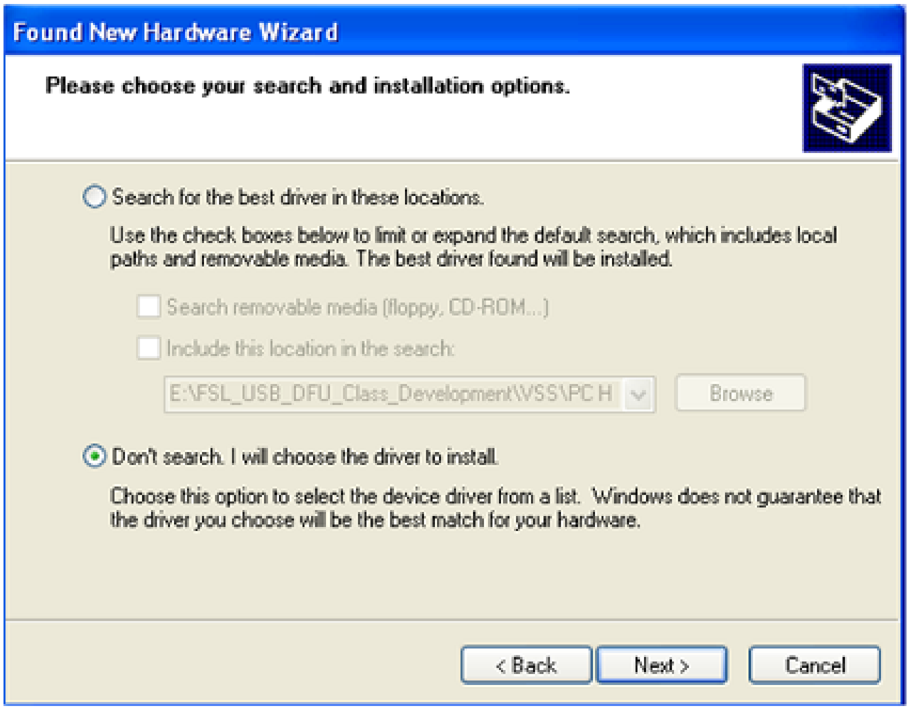

Figure 10. Search and installation options

3. The Hardware Type window appears. Select “Show All Devices” option, and click “Next” button. Select “Have Disk…” button as soon as “Select device driver window” appears.

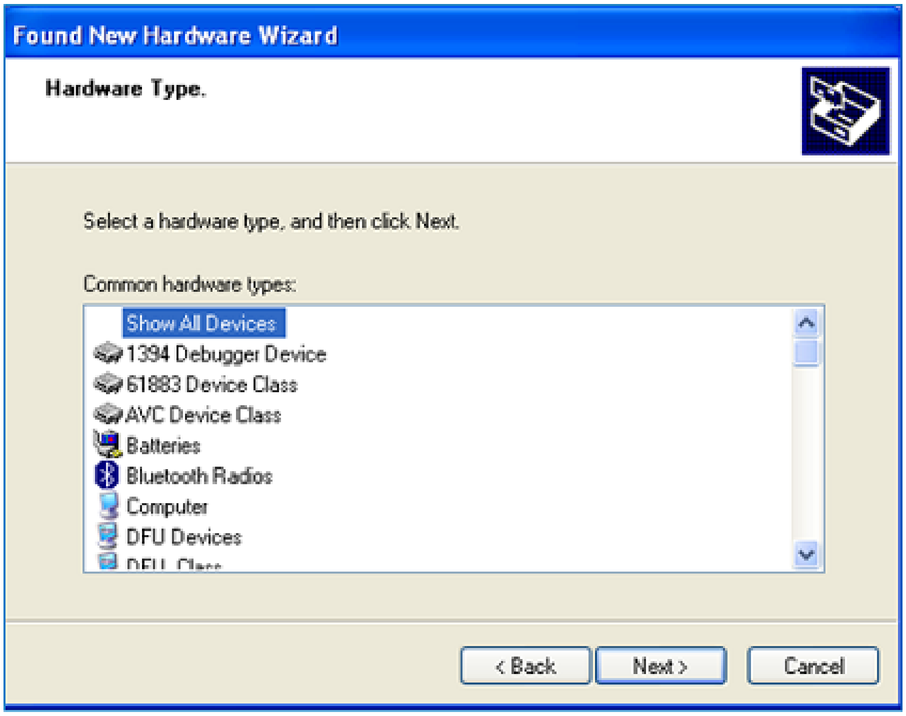

Figure 11. Hardware type

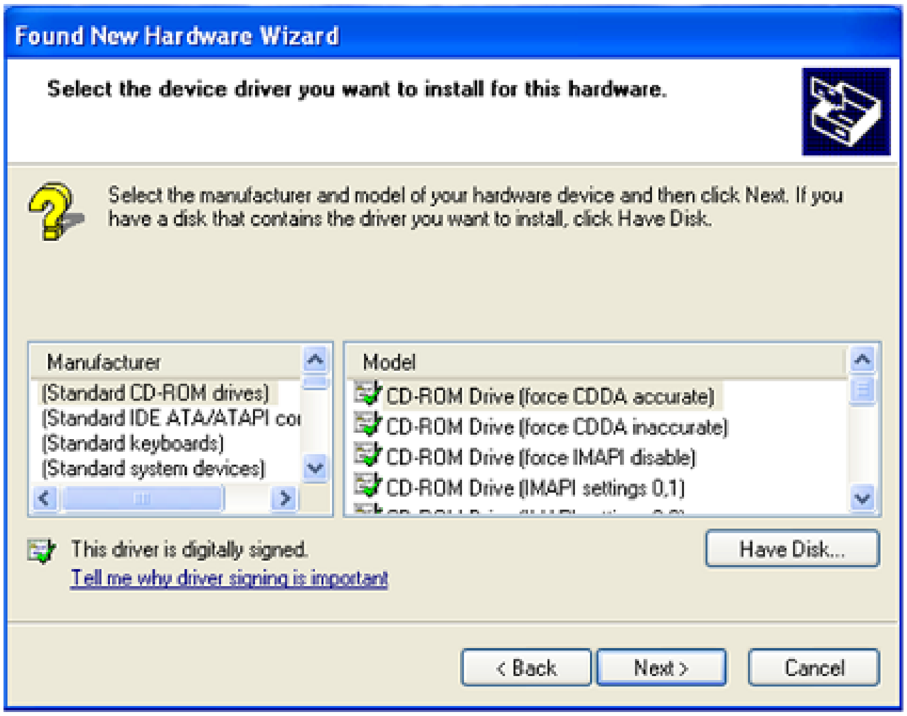

Figure 12. Select device driver

4. Navigate to the INF file located at \DFU_winusb_driver and choose DFU_Device_Runtime.inf file. Click “Open,” then click “Next” to install the USB driver.

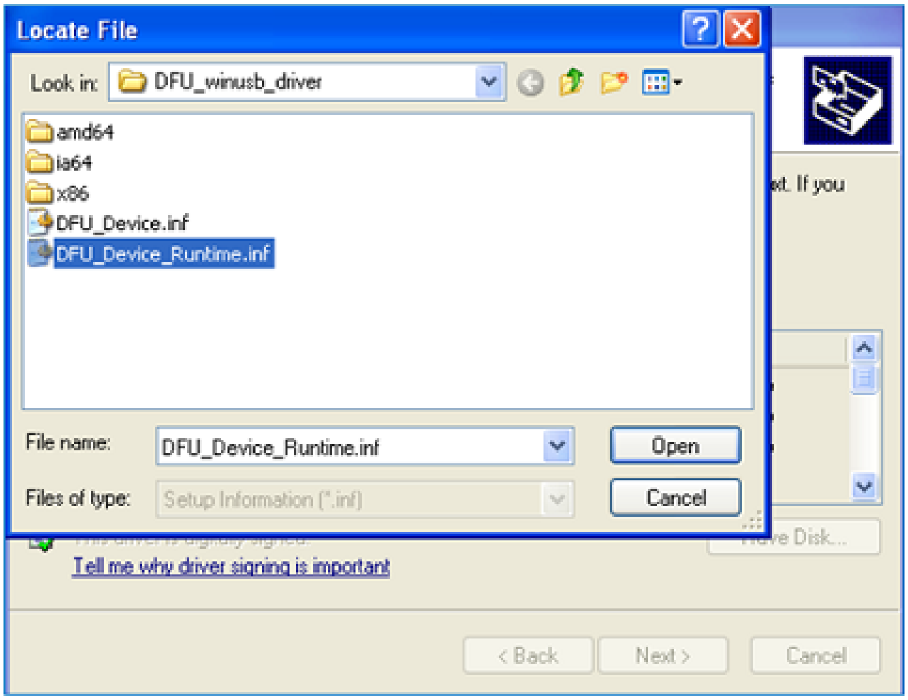

Figure 13. Selecting the driver

5. Once the driver is installed, Windows recognizes it as a composite device made of a DFU class and HID mouse, as explained in Bootloader Overview.

   To verify the USB installation, open the Windows device manager. The “Device firmware upgrade” (DFU) and “USB Human Interface Device” entries are displayed by the device manager in the following figure.

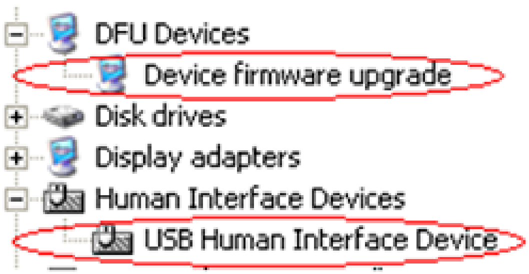

Figure 14. DFU device and Human Interface Device in device manager

6. Open the USB DFU PC application. The PC application automatically recognizes that the run-time mode (USB composite device) is running as shown in the following figure. Click “Enter DFU mode” button to switch the device to DFU mode.

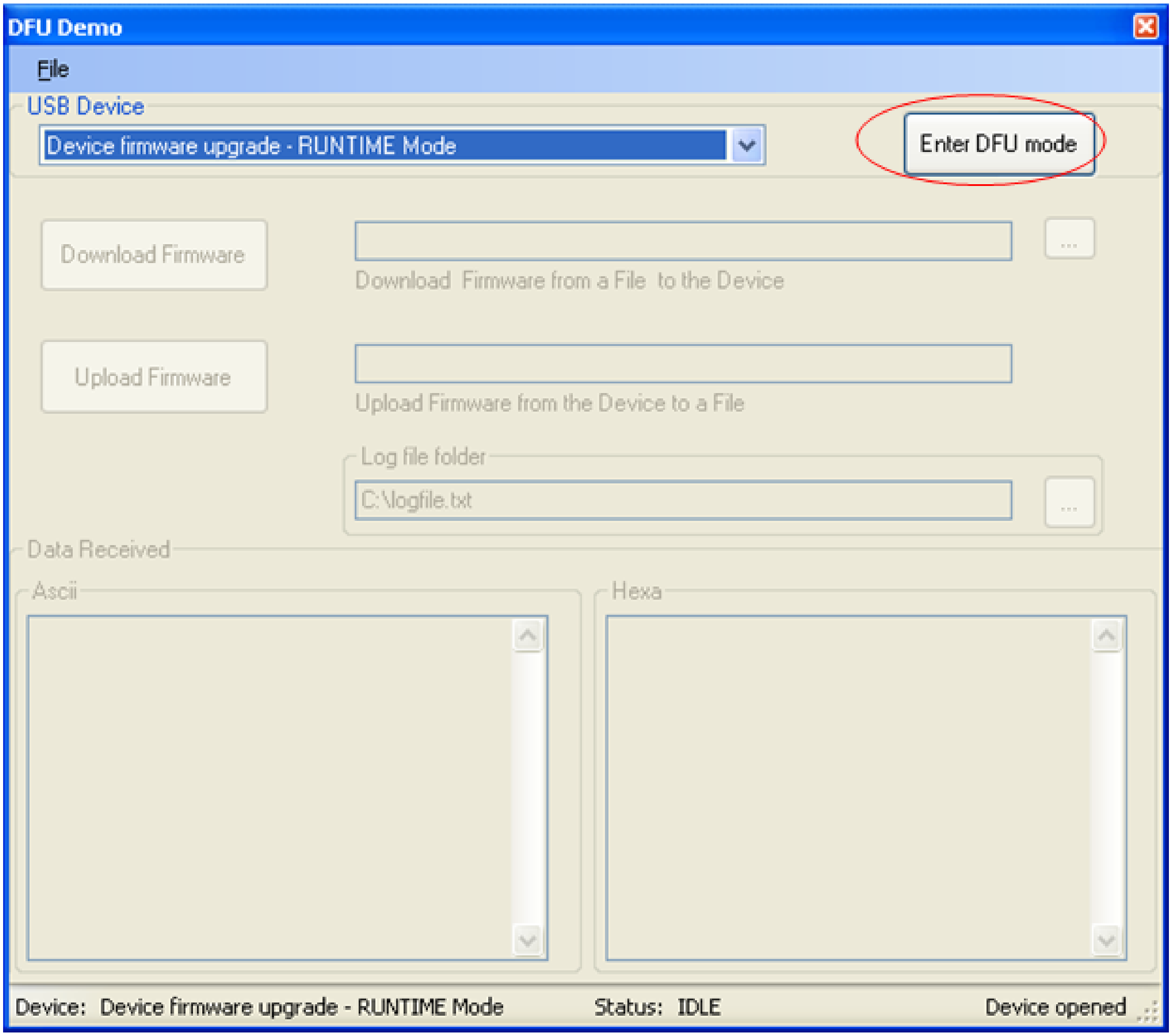

Figure 15. Device firmware upgrade - runtime mode

7. Unplug and plug the USB cable to get a USB bus reset.The M52259EVB USB device will enter in DFU mode.
8. When DFU mode is entered, Windows OS will ask for driver again. Follow steps 2- 4 to install the USB DFU driver, this time selecthing DFU_Device.inf as shown in the next figure.

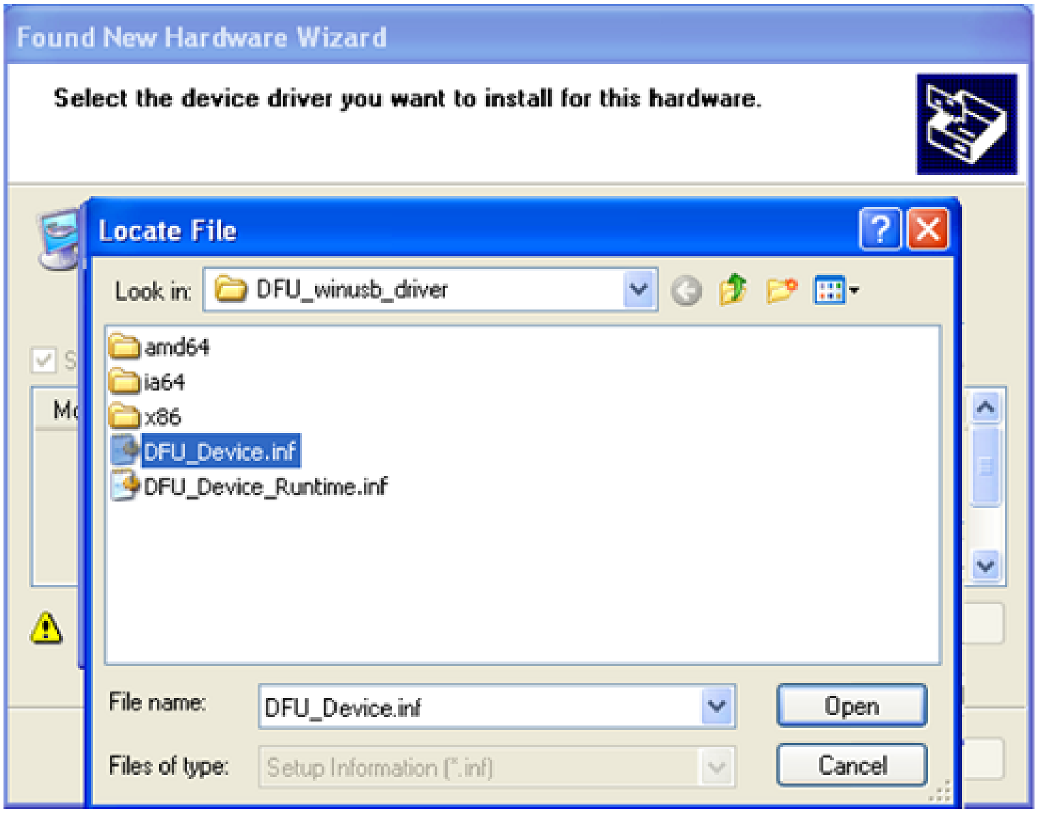

Figure 16. Install driver for DFU mode

9. Once the driver for DFU mode is installed successfully, the USB DFU device bootloader is in DFU mode and ready to use. The USB DFU PC application is shown as follows:

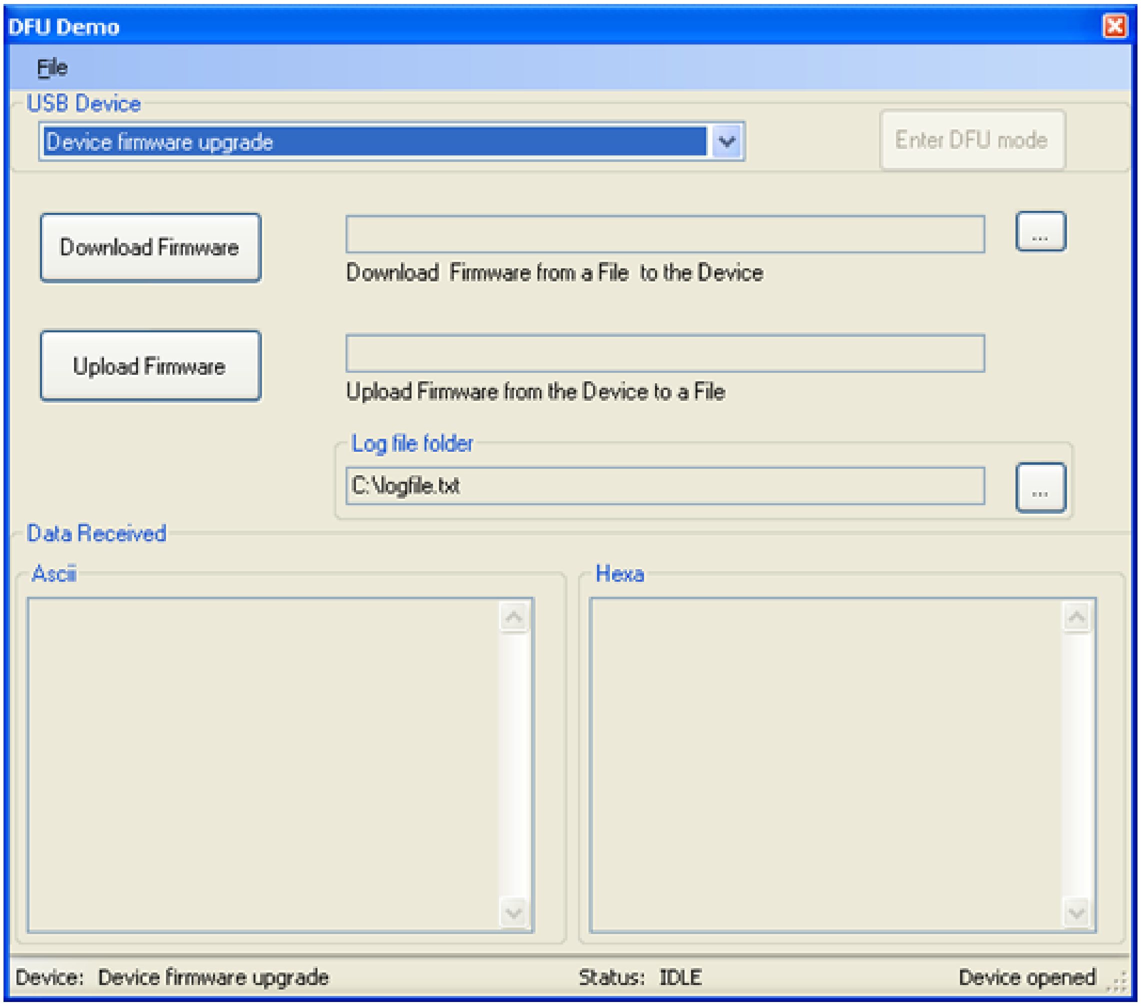

Figure 17. DFU device demo in DFU mode

NOTE

The use of a USB hub or docking station for the USB DFU device bootloader is not recommended.

5.5 Downloading firmware

The following steps must be followed to download the firmware through the USB DFU bootloader.

1. At this point, Driver installation must have already been completed. Using the USB DFU PC Application, select a firmware image file for download to the device as shown in Figure 21. The files generated in Building the application can be used for this step.

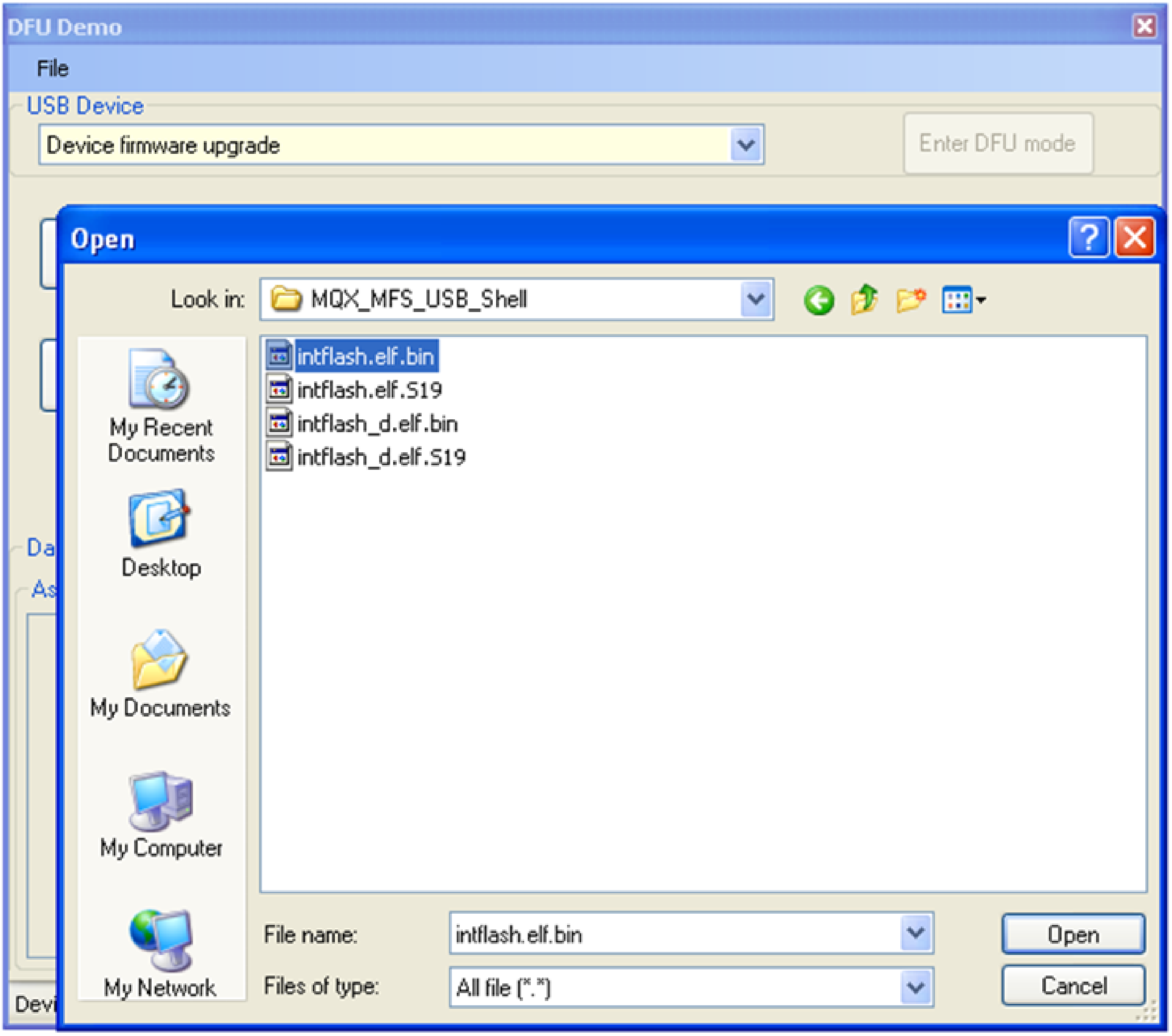

Figure 18. Choosing firmware file

2. When a S19 file is selected, the content of the firmware file is displayed in ASCII and hexadecimal (HEX) format. If a CodeWarrior binary format is selected, the content of the firmware is only displayed in hexadecimal (HEX) format, as shown in next figure.

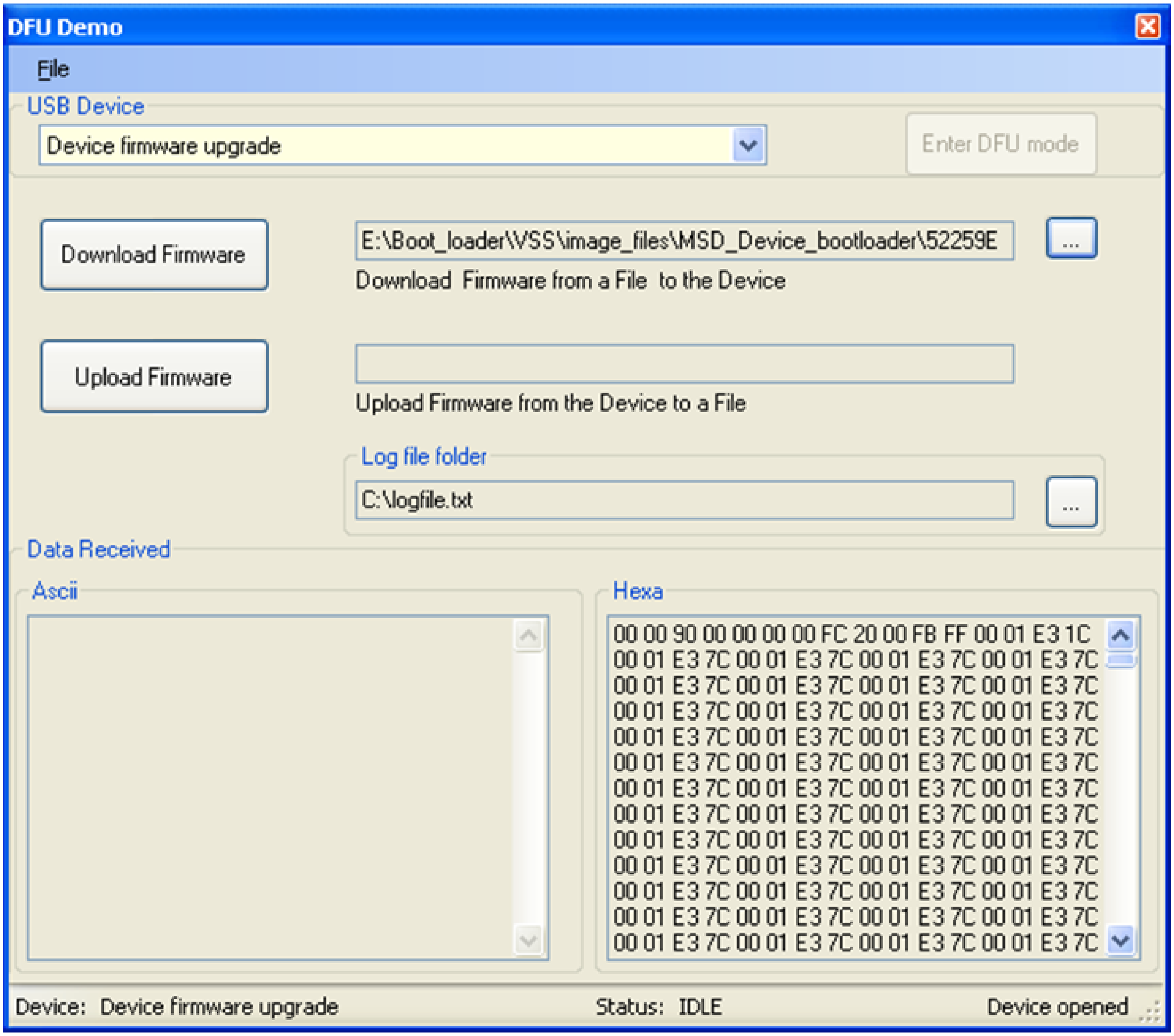

Figure 19. Content of the firmware is displayed

3. Click “Download Firmware” button. The firmware will be downloaded to the device.

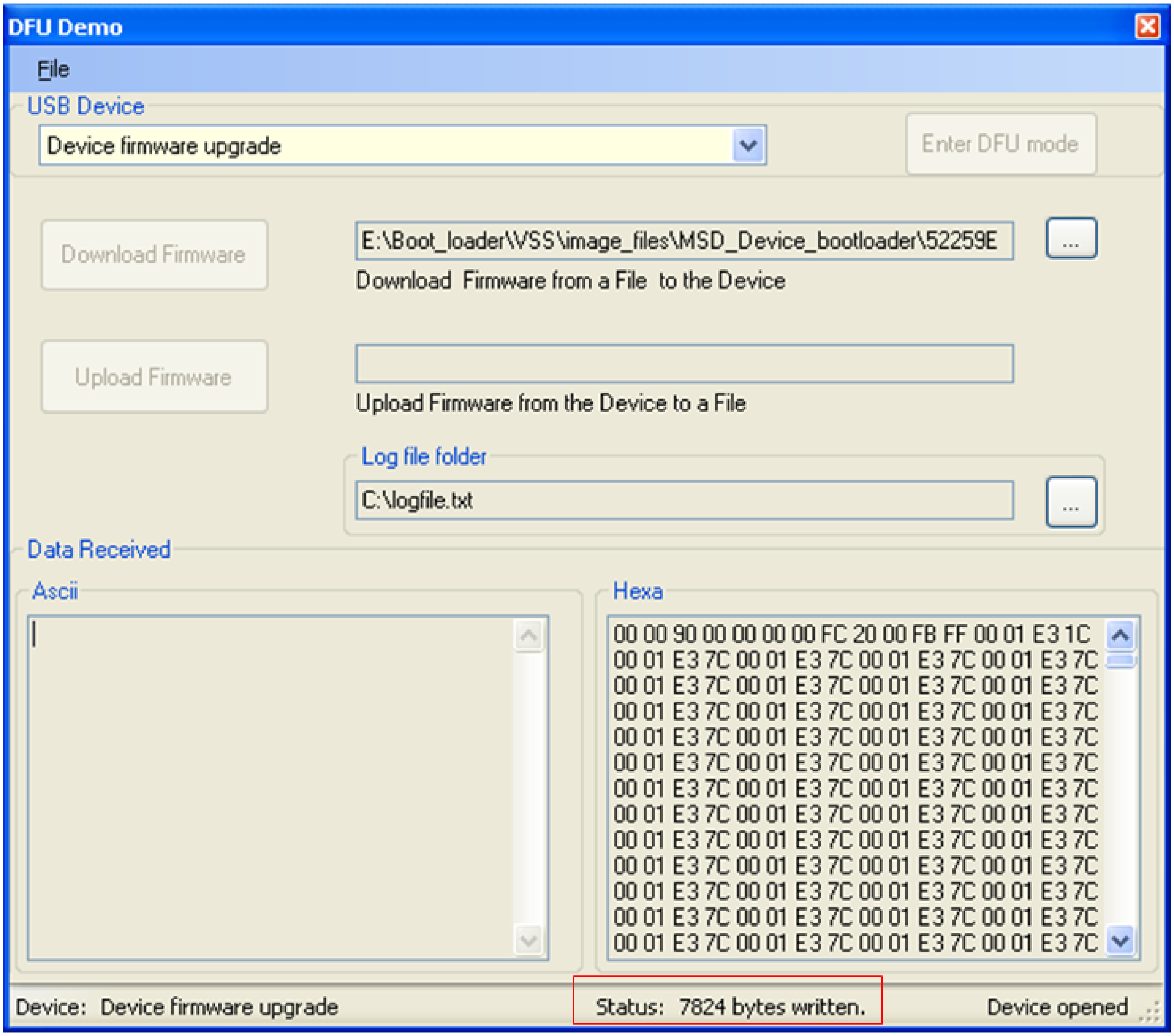

Figure 20. Firmware is downloaded

4. Once the download firmware process is completed, the USB DFU PC Application shows the final status of the download process.

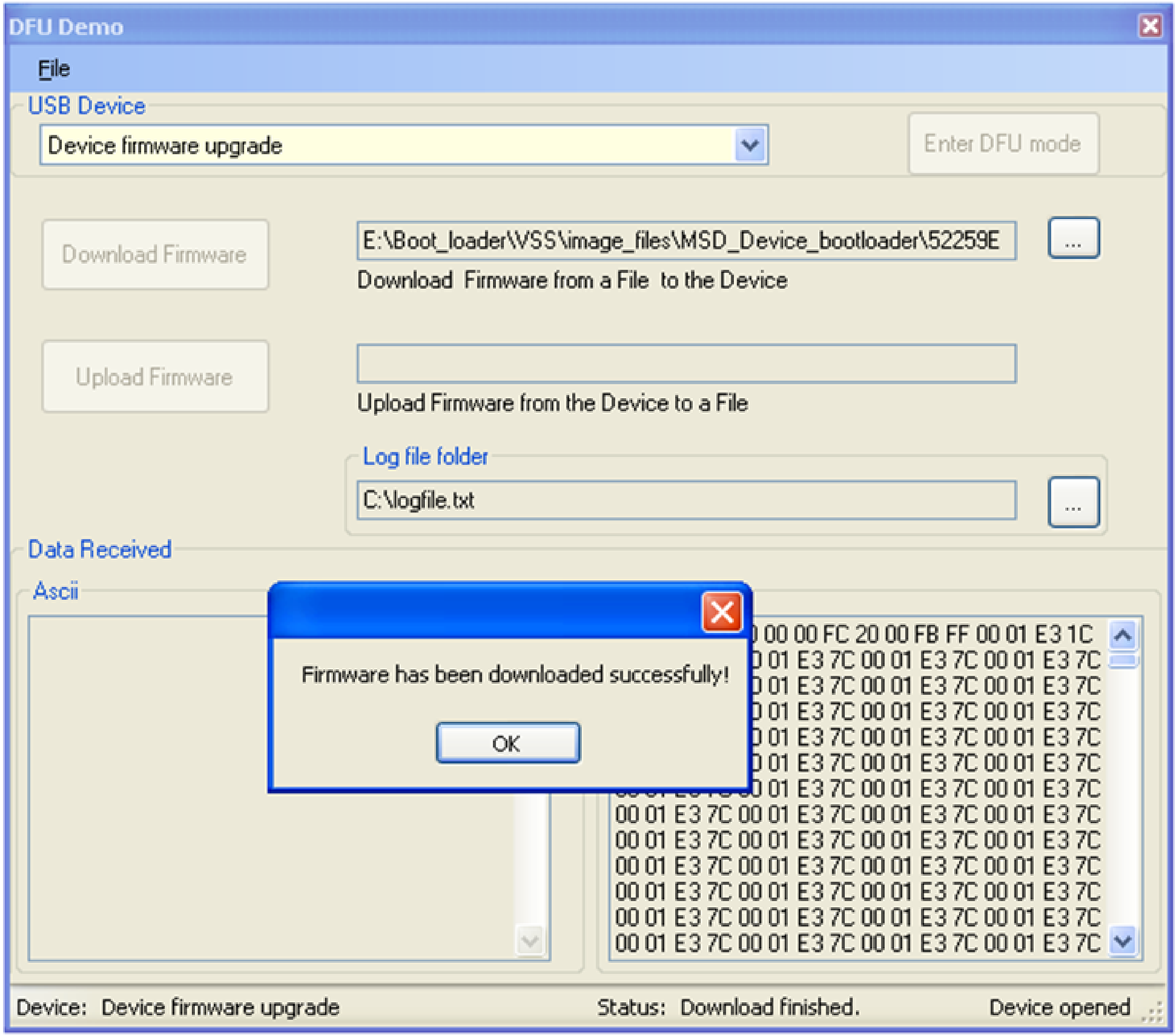

Figure 21. Download is completed

5. As an additional verify process, a log file contains the events which occurred during the download process.

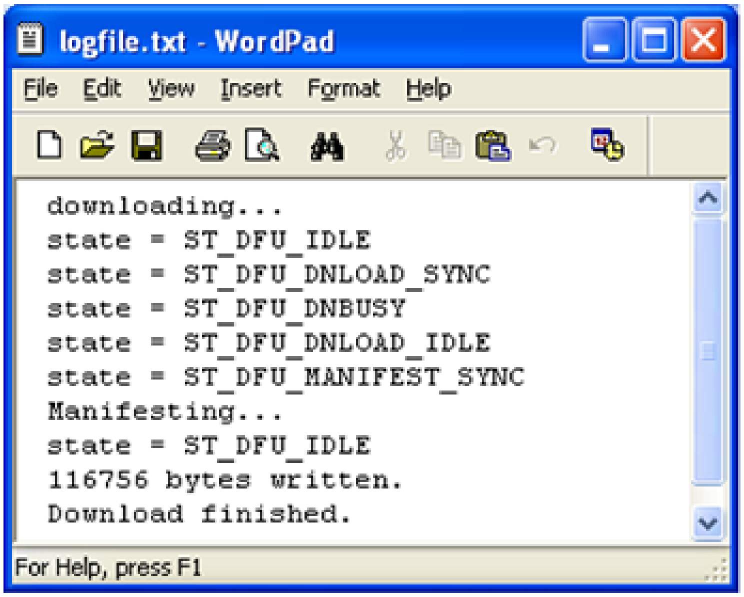

Figure 22. Content of log file

6. Press reset key on board to run the user application. The serial terminal shows a menu sent by MQX user application.

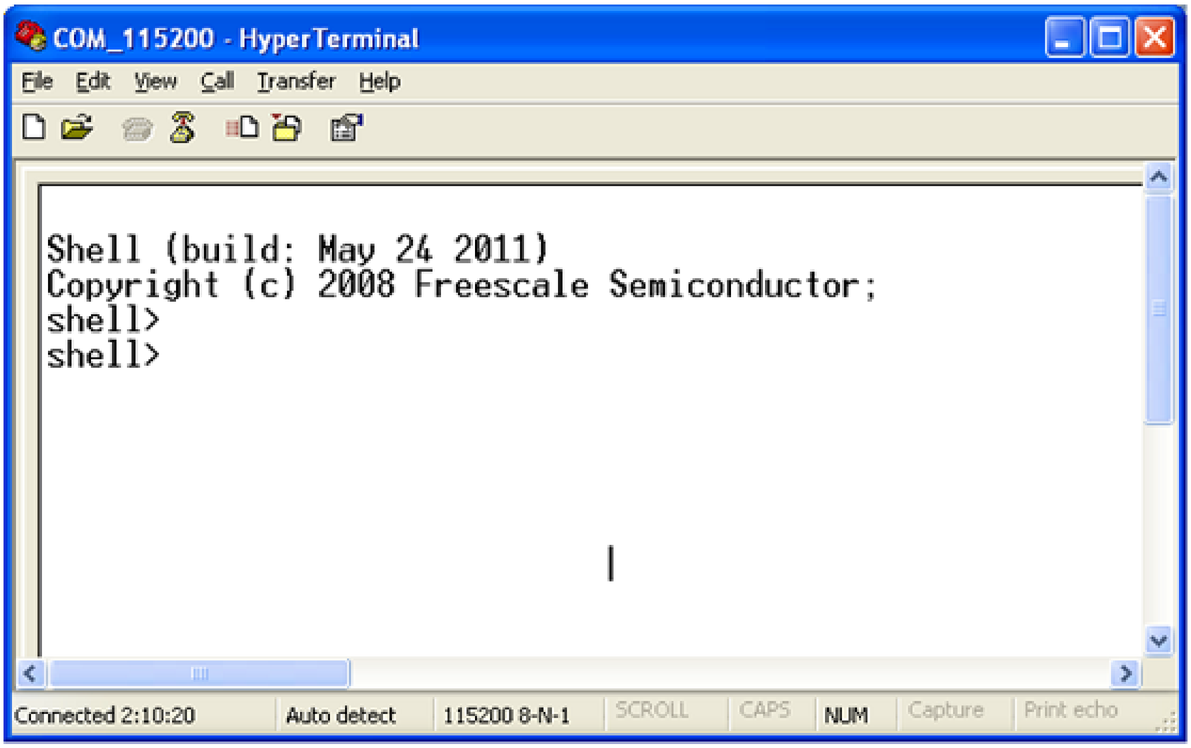

Figure 23. User application running

NOTE

If the USB cable is unplugged during the download process, The USB DFU PC application will ask to continue the download process whenever the USB cable is re-plugged, as shown in Figure 24.

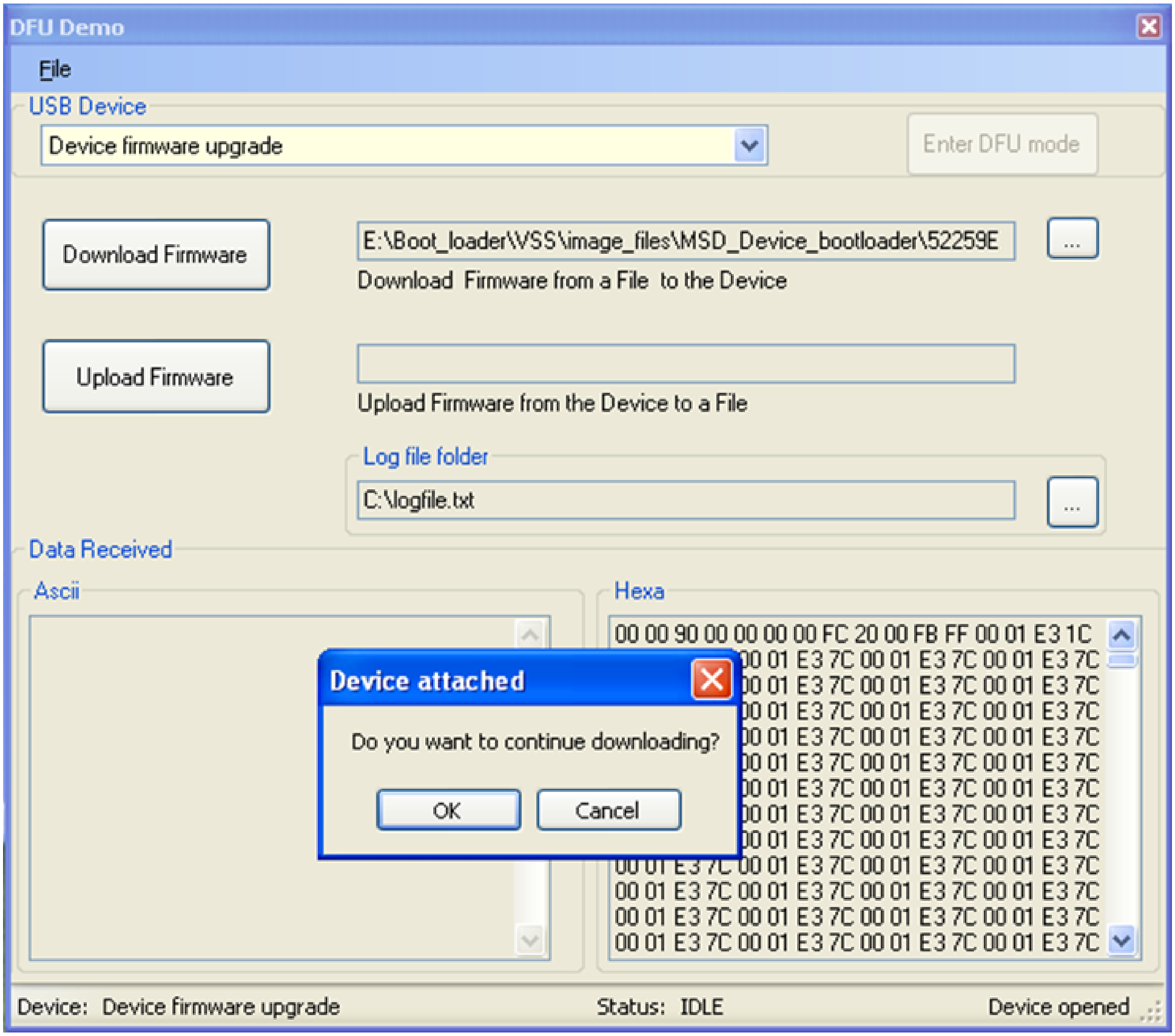

Figure 24. Resuming download

6 Port the Bootloader to Other Platforms

The following section explains how to develop new USB DFU bootloader applications in other platforms. The USB DFU bootloader is developed over the “Freescale USB Stack with PHDC v3.0” software.

6.1 USB DFU bootloader file structure

The following figure shows the folder structure of the DFU source code:

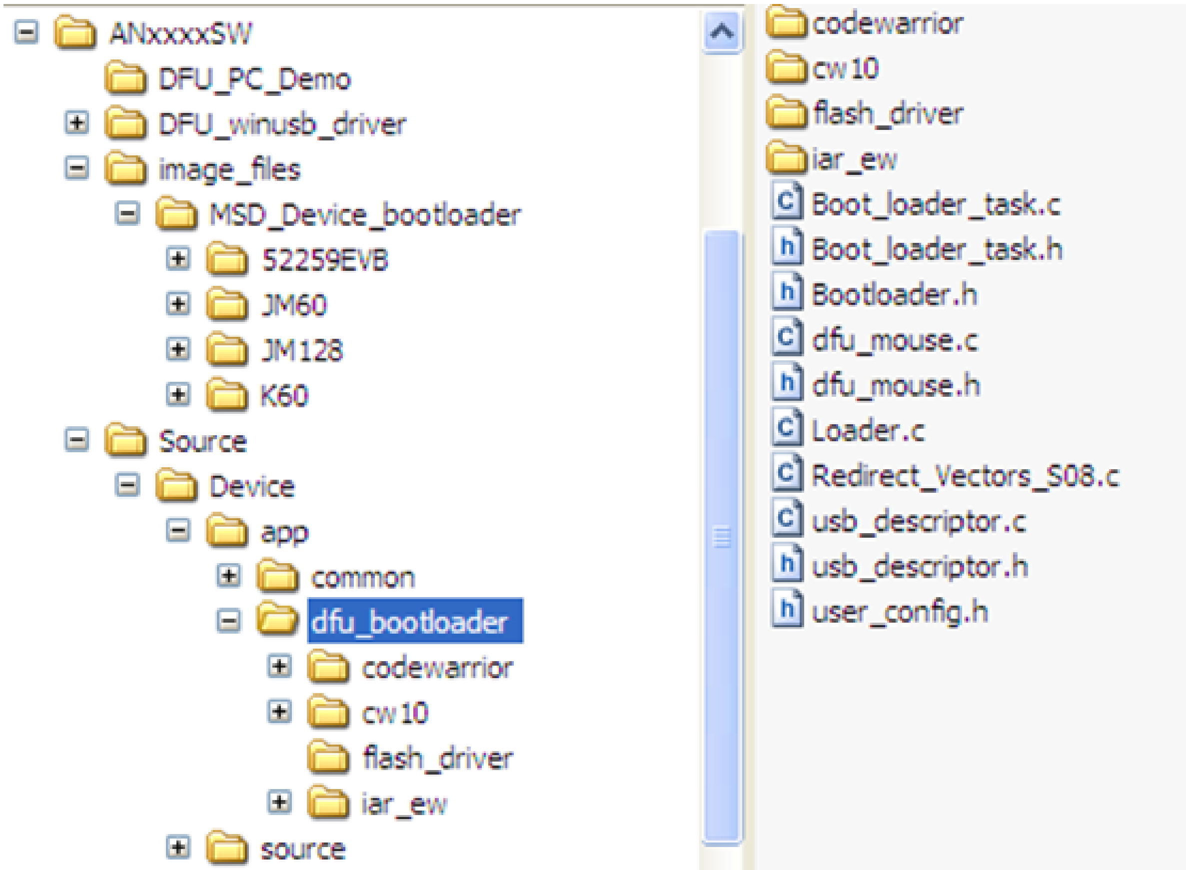

Figure 25. USB DFU bootloader file structure

The top-level folders contain:

- DFU_PC_Demo: contains the USB DFU PC application
- DFU_winusb_driver: contains the USB drivers needed by Windows OS
- image_files: contains examples firmware image files for MC9S08JM60, MCF51JM128, MCF52259, and K60 MCUs
- Source: contains USB DFU bootloader source code

The dfu_bootloader folder contains the following folders:

- codewarrior: contains CodeWarrior v6.3 and v7.2 projects
- cw10: contains CodeWarrior 10.2 projects
- iar_ew: contains IAR projects
- flash_driver: contains flash driver for supported MCUs

The following files are part of the dfu_bootloader folder:

- Boot_loader_task.c: contains bootloader general tasks
- Boot_loader_task.h: includes function prototypes
- Bootloader.h: includes memory map definitions for ported boards to DFU bootloader
- dfu_mouse.c: contains DFU application and mouse functionality
- dfu_mouse.h: contains DFU parameters definitions
- Loader.c: contains functions to parse and load firmware image to MCU flash memory
- Redirect_Vectors_S08.c: contains bootloader interrupts for MC9S08JM60 (S08 MCU)
- usb_descriptor.c: contains USB descriptor structures and functions
- usb_descriptor.h: contains USB descriptor parameters
- user_config.h: contains user configurations

6.2 Creating new projects

To create new USB DFU bootloader projects:

1. Create a new project under:
    - Source\Device\app\dfu_bootloader\codewarrior, or
    - Source\Device\app\dfu_bootloader\cw10

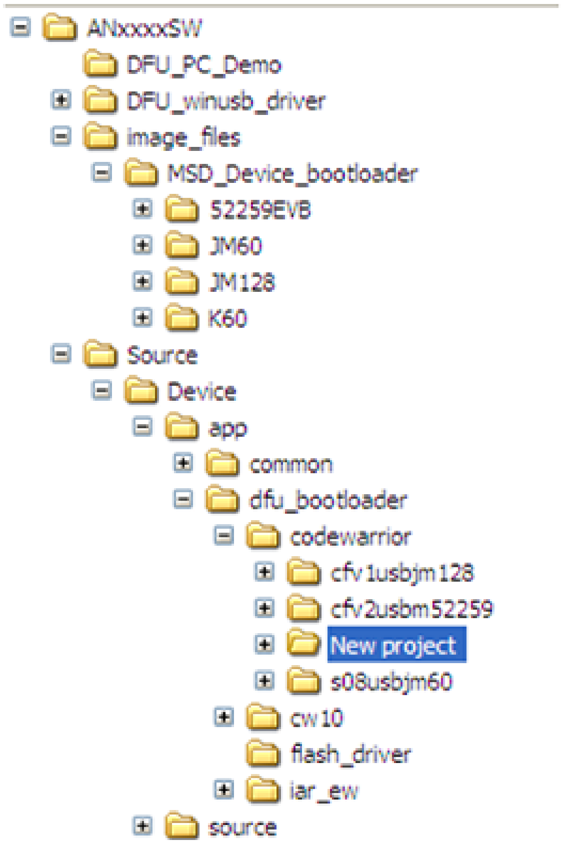

Figure 26. Create a new project folder

2. Create a project with a file structure like bootloader project for M52259EVB. Use the cfv2usbm52259 project as a CodeWarrior template.

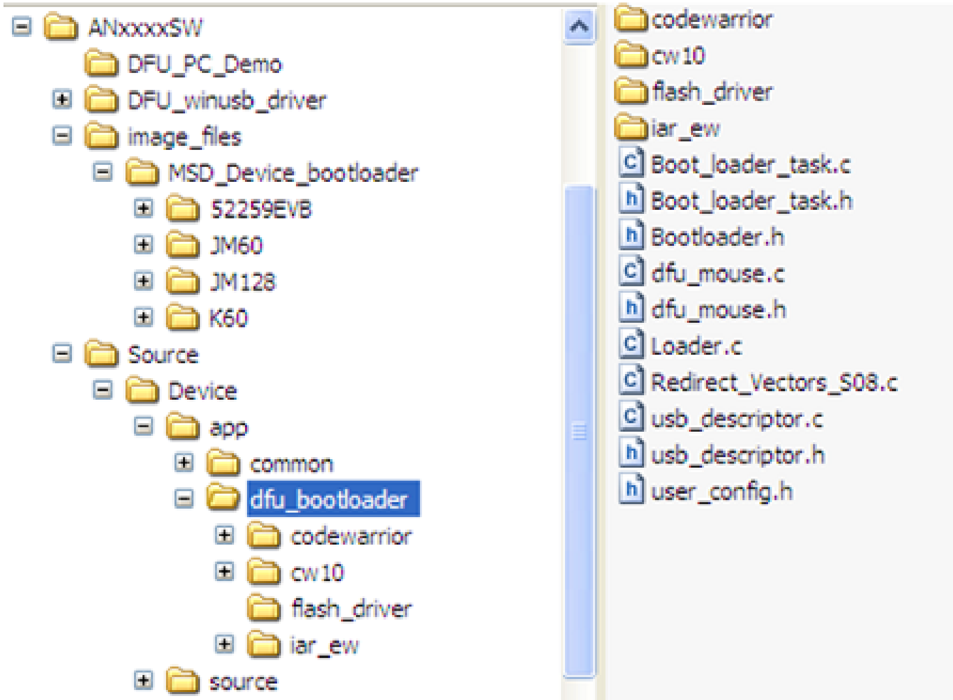

Figure 27. M52259 bootloader project

3. Add files to project:
    - Flash driver source code:
        - flash.c: CFV1 and ColdFire+ flash driver
        - flash_cfv2.c: CFV2 flash driver
        - flash_FTFL: Kinetis (L and K family) flash driver
        - flash_hcs: S08 flash driver
        - flash_NAND.c: NAND flash driver
    - USB classes (DFU and HID classes) source code
    - USB device driver source code
    - dfu_mouse.c, dfu_mouse.h, Boot_loader_task.c, Boot_loader_task.h, Loader.c, Bootloader.h, usb_descriptor.c, usb_descriptor.h, and necessary files specific to boards
4. Modify Boot_loader_task.c file for the specific board willing to implement DFU bootloader.
5. Modify memory map which indicates application region for the platform in Bootloader.h file as shown below:

```
#if (defined __MCF52259_H__)
#define MIN_RAM1_ADDRESS          0x20000000
#define MAX_RAM1_ADDRESS          0x2000FFFF
#define MIN_FLASH1_ADDRESS        0x00000000
#define MAX_FLASH1_ADDRESS        0x0007FFFF
#define IMAGE_ADDR                ((uint_32_ptr)0x9000)
#define ERASE_SECTOR_SIZE         (0x1000)  /* 4K bytes*/
#define FIRMWARE_SIZE_ADD        (0x0007FFF0 )
#elif (defined _MCF51JM128_H)
#define MIN_RAM1_ADDRESS          0x00800000
#define MAX_RAM1_ADDRESS          0x00803FFF
#define MIN_FLASH1_ADDRESS        0x00000000
#define MAX_FLASH1_ADDRESS        0x0001FFFF
#define IMAGE_ADDR                ((uint_32_ptr)0x0A000)
#define ERASE_SECTOR_SIZE         (0x0400)  /* 1K bytes*/
#define FIRMWARE_SIZE_ADD        (0x0001FFF0 )
#elif (defined MCU_MK60N512VMD100)
#define MIN_RAM1_ADDRESS          0x1FFF0000
#define MAX_RAM1_ADDRESS          0x20010000
#define MIN_FLASH1_ADDRESS        0x00000000
#define MAX_FLASH1_ADDRESS        0x0007FFFF
#define IMAGE_ADDR                ((uint_32_ptr)0xA000)
```

```
#define ERASE_SECTOR_SIZE        (0x800)  /* 2K bytes*/
#define FIRMWARE_SIZE_ADD       (0x0007FFF0 )
#endif
```

7 Conclusion

The USB DFU class can be used as an option to make upgrades to the MCU firmware on the field. The application running over the DFU bootloader requires only modifications in the linker file and exception table. The solution outlined in this document can be migrated to any Freescale 8/16/32-bit MCU.

7.1 Problem reporting instructions

Issues and suggestions about this document and drivers should be provided through the support web page at freescale.com/support. Please reference this application note.

7.2 Considerations and references

- Find the newest software updates and information about the USB DFU bootloader for MCUs on the Freescale Semiconductor home page at freescale.com.
- More implementations using USB DFU class in Freescale MCUs can be found in the latest “Freescale USB Stack with PHDC” software from freescale.com/usb.
- More details about USB DFU class can be found in “USB Device Firmware upgrade specifications” from usb.org.
- The AN4370SW software contains all the necessary SW to run USB DFU class in the embedded device and PC running Windows OS.
- Download the source files for AN4370SW software (AN4370SW.zip) from freescale.com/.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4370
Rev. 1, 2012