Freescale Semiconductor
Application Note

Document Number: AN4379
Rev. 0, October 2011

Freescale USB Mass Storage Device Bootloader

by: Derek Snell
Freescale

Contents

1 Introduction

Freescale offers a broad selection of microcontrollers that feature universal serial bus (USB) access. A product with a USB port allows very easy field updates of the firmware. This application note describes a mass storage device (MSD) USB bootloader that has been written to work with several Freescale USB families. A device with this bootloader is connected to a host computer, and the bootloader enumerates as a new drive. The new firmware is copied onto this drive, and the device reprograms itself.

Freescale does offer other bootloaders. For example, application note AN3561, "USB Bootloader for the MC9S08JM60," describes a USB bootloader that was written for the Flexis JM family. The MSD bootloader described in this application note is offered as another option, and has these advantages:

- It does not require a driver to be installed on the host.
- It does not require an application to run on the host.
- Any user can use it with a little training. The only action required is to copy a file onto a drive.
- It can be used with many different host operating systems since it requires no host software or driver

This bootloader was specifically written for several families of Freescale microcontrollers that share similar USB peripherals. These families include, but are not limited to, the following:

- Flexis JM family MCF51JM

- ColdFire MCF522xx parts that have USB
- Kinetis

This bootloader will work on all of these devices with few changes, and examples were written and tested on the following Freescale microcontrollers:

- MCF52259 — 32-bit ColdFire V2 with USB, Ethernet, CAN, and external bus
- MCF51JM128 — 32-bit ColdFire V1 with USB (part of the Flexis JM family)
- MK60N512 — ARM® Cortex™-M4 architecture with USB 2.0 full-speed OTG controller and 10/100 Mbps Ethernet MAC

USB stack used:

- Freescale's USB Stack with PHDC support v3.0

Tested development boards:

- TWR-MCF5225X-KIT — low-cost tower kit for MCF5225x family
- DEMOJM — low-cost development board for Flexis JM family
- TWR-K60N512-KIT — low-cost tower kit for Kinetis K60 family

Tested operating systems:

- Windows XP Pro with Service Pack 2 and Service Pack 3

Tested development tool:

- CodeWarrior for Microcontrollers v10.1
- CodeWarrior for Microcontrollers v6.3
- CodeWarrior for ColdFire v7.2

2 Functional description

2.1 General overview

This section provides an overview of the USB MSD device bootloader architecture and its software flow.

2.1.1 USB MSD device bootloader architecture

The architecture of Freescale's USB MSD device bootloader is shown in Figure 1.

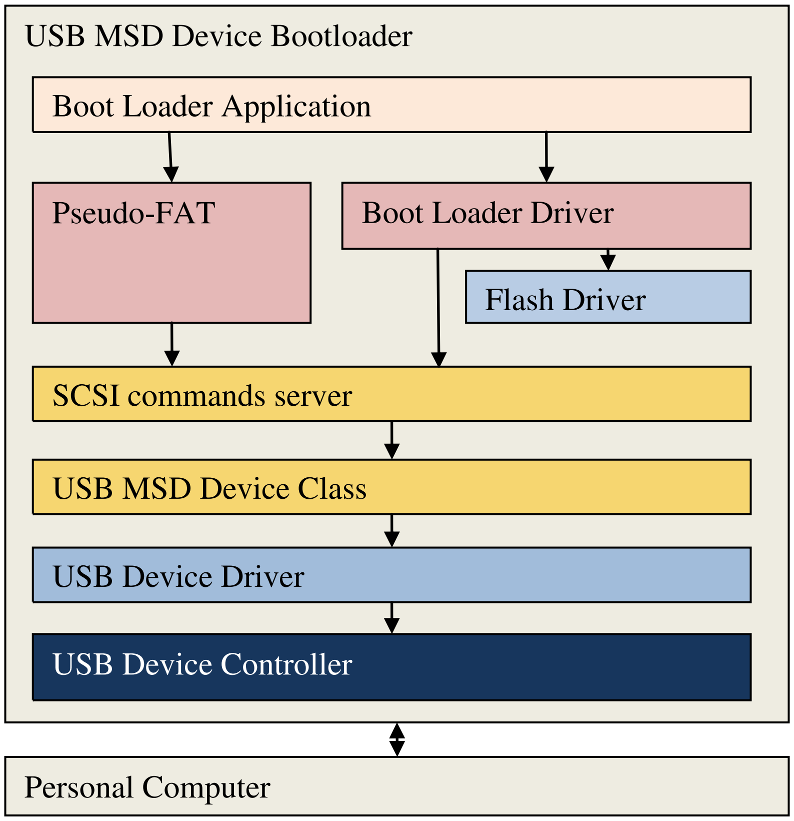


Figure 1. USB MSD bootloader architecture

The USB MSD device bootloader includes:

- Bootloader application: Controls complete loading process and implements general tasks
- Pseudo-FAT structure: Contains structures to respond to SCSI commands from host.
- Bootloader driver: Uses SCSI command server to receive data from host computer, then parses image files and downloads them to flash memory. The bootloader driver supports parsing image files in CodeWarrior binary, S-Record, and raw binary file formats.
- Flash driver: Supports functions to erase, read, and write flash memory.
- SCSI command server: Responds to SCSI commands from host.
- USB MSD device class: Provides APIs specified in MSD class.
- USB device driver and device controller: Communicate with the USB host though USB protocols.

2.1.2 Bootloader flow

The bootloader is integrated with an application that performs the product's main functions. At reset, the bootloader executes and does some simple checks to see if the application should start, or if the bootloader should start. If it enters bootloader mode, it uses USB to enumerate with the host computer. During this enumeration process, the device declares itself as an MSD. The host then creates a new drive in the system. A firmware image file can then be copied onto the drive and the device reprograms itself. The bootloader supports firmware images in S-record, CodeWarrior binary, and raw binary file formats.

An S-record file is a common ASCII file type that is used to specify the program data stored in devices. Freescale's software tool chain called CodeWarrior generates S-record files and CodeWarrior binary files automatically when projects are compiled. S-record files have the extension .S19 and CodeWarrior binary files have the extension .bin. A raw binary file is the image file of flash memory.

After the firmware image file has been transferred to the device and the device reprograms itself, the device re-enumerates with the host. A file is displayed in the drive that represents the status of the bootloader operation.

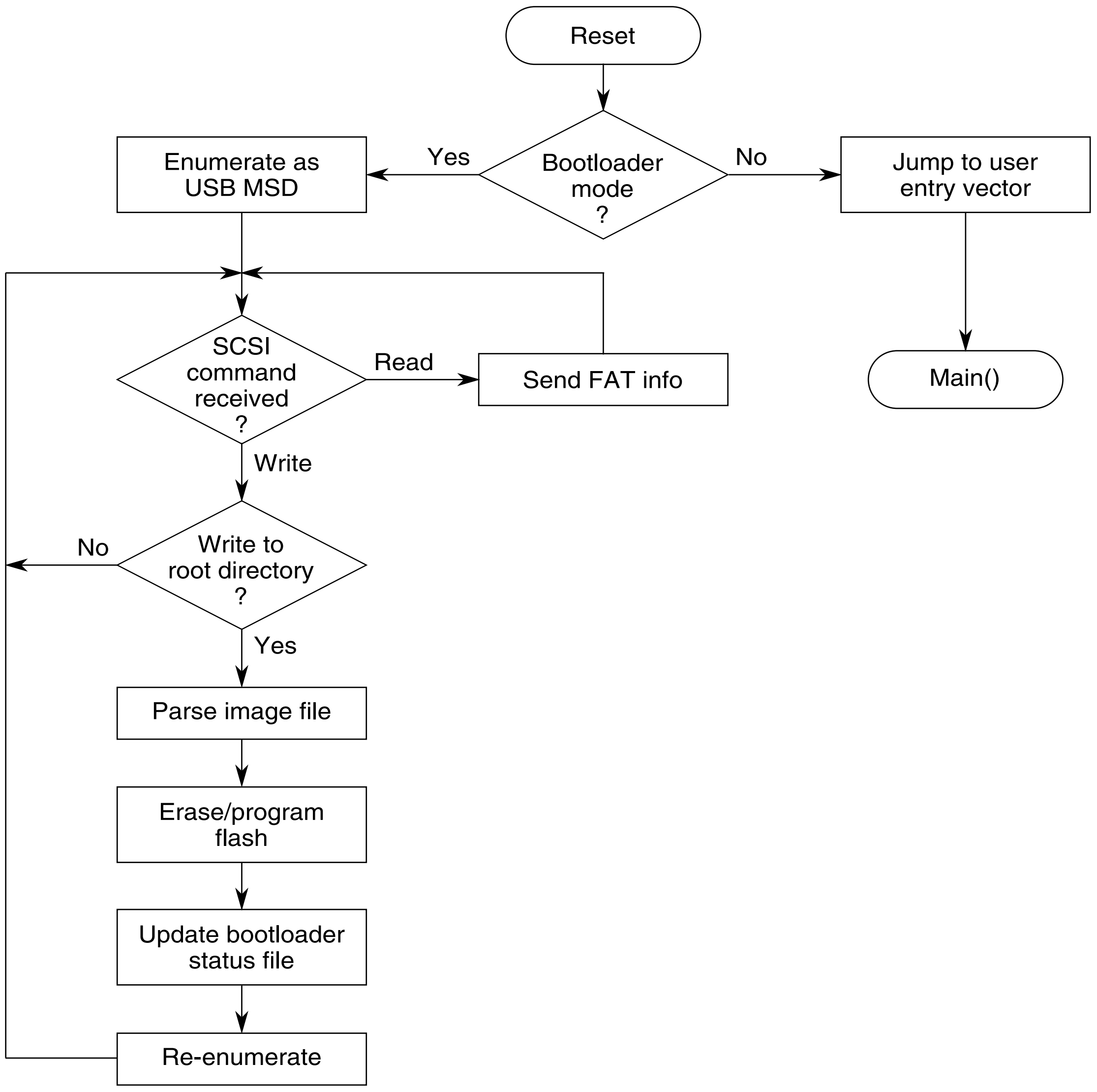


Figure 2. Bootloader functional flow chart

2.2 Bootloader mode vs. application mode

The bootloader starts executing immediately after reset and determines what mode to enter. It is important for the bootloader to be in control after reset, to allow it to run if the application is not present, has been erased, or has become corrupted. Normally, the bootloader attempts to enter application mode. It does this by jumping to the application's reset vector.

Before entering application mode, the bootloader performs some simple checks to ensure the application is present. It looks at the initial stack pointer value and reset vector of the application (the first four bytes and second four bytes of the firmware image), and checks to be sure they are not erased. If they are valid, the bootloader jumps to the application — if not, the bootloader runs to allow an application update.

The bootloader mode can also be forced during a reset. The included examples monitor a button, and if the button is held during reset, the bootloader mode is entered regardless of the values of the stack pointer and reset vector of the application. The button used to enter this mode depends on the development board; see Using the bootloader on a demo board for the specific button. Many applications need to change the method used to force bootloader mode, which can be done by modifying the Boot_loader_task.c file.

2.3 USB enumeration

After the bootloader enters bootloader mode, it enables the USB peripheral to communicate with the host computer. This starts the USB enumeration process. (For more details on enumeration, please refer to document MEDUSBUG, "Freescale USB Stack with PHDC Device Users Guide.") The USB stack used by this bootloader is Freescale's USB Stack with PHDC v3.0. The device sends USB descriptors to the host that declare it as a mass storage device. The host loads the device as a new drive, and uses bulk transfers to communicate with the device. These bulk transfers send SCSI commands to communicate with what the host sees as a removable storage drive.

2.4 Pseudo-FAT

After the host has configured the bootloader as a storage drive, it attempts to read the file allocation table (FAT) of the drive and read the drive's contents. The bootloader responds to these queries with information that looks like a 1 GB drive. The drive has the volume name BOOTLOADER. It looks as if it has no subdirectories but has a file called READY.TXT in the root directory.

The bootloader does not use a true file system, and is only capable of appearing as a drive to the host. It will accept files that can be written to it which can then be transferred to the firmware image files. However, because the bootloader is only a Pseudo-FAT system, it does not update the FAT when the host attempts to change it. The FAT is always fixed, and the only change is the name of the file in the root directory that specifies the status of the bootloader.

When the host writes data to the drive, the bootloader ignores all data that is not written to the root directory of the drive. When file data is written to the root directory, the bootloader assumes it is a firmware image file. It erases the application flash and starts the firmware image parser to get the new firmware update. Only one file can be transferred to the drive after reset. After a file is transferred, the bootloader needs to restart to accept another file.

2.5 Firmware image file parser

The bootloader includes a firmware image parser to allow firmware image files to be sent unaltered to the bootloader. After a file has been transferred to the bootloader, it assumes the file is a firmware image file and starts to parse it. It parses a buffer that stores a part of the firmware image, and if successful, continues parsing the next part of the image until the whole image is parsed.

If the address of the data is valid, it writes all data of the firmware image to the flash memory. For S-Record files, each S-record checksum is verified before the data is written. After all flash operations are successful, the parser returns a message indicating success. If at any point the image file was invalid or an S-record checksum did not verify, the parser returns an error.

NOTE

This parser can also be used by other bootloaders or applications that do not use USB. If a different type of communication is desired, the program simply stores the data image received in a buffer and passes the buffer to the function of the parser (in file loader.c) to parse the image.

2.6 Re-enumeration

After the parser finishes, the device re-enumerates with the host to update the status file on the drive. The status files are discussed in Bootloader status. Within the host computer, the drive disappears as the device drops off the USB bus. After a few seconds, the device re-enumerates and the drive re-appears. You can read the status file on the drive to see the status of the bootloader.

The bootloader re-enumerates because some host operating systems cache the file data on the storage drive and do not refresh. Other operating systems do a delayed write and do not write the file to the device until the drive is unmounted. After unmounting, the device receives the firmware image file, updates the firmware, and then re-enumerates to display the status. In Windows XP and Vista, the OS caches the drive's FAT without re-reading it and the status file is not updated. Re-enumeration forces the host to re-read the FAT and display the bootloader status file.

2.7 Bootloader status

The bootloader uses the file system drive in the host computer to display the status. With this method, the bootloader status can always be seen regardless of the type of display or LEDs available on the device’s hardware. In the case of an S-record error, the address of the error is also displayed.

The status is displayed in the filename in the root directory of the drive. These files are empty and have no data. Before a file is transferred to the bootloader, the user must check the status file, READY.TXT. After the file transfer is complete, and the bootloader re-enumerates, the user must check the status file to see if it was a success. The status filenames are listed below:

- READY.TXT — Shows that the bootloader is ready to receive an image file.
- SUCCESS.TXT — Shows that the bootloader completed a firmware update successfully.
- FFAILED.TXT — Shows that the bootloader had a flash erase or program error. This error is triggered if the flash erase or program routine returns an error. This error is usually triggered because the flash clock is not at the appropriate frequency, or an invalid address is used. If the address is not a flash address or is protected, this error occurs. To diagnose, place a breakpoint where this error is generated in Loader.c and run the debugger to find where the error occurs.
- SFxxxxxx.TXT — Shows that the bootloader S-record parser found an error in an S-record. The parser tests each S-record to ensure that it has a valid format and that the checksum matches. If the test fails, this error is generated. This error can also occur if the S-record specifies an invalid address. The address of the improper S-record appears in the filename. For example, if the S-record for address 0x1234 was invalid, the status filename would be SF001234.TXT. If this error occurs, check the S-record for proper format, checksum, and address.
- STARTED.TXT — Shows that a file has been received with data for the root directory and the firmware image parser has started. Normally, this status should not be visible to the user because the drive only re-enumerates after a success or failure. If seen, debug the bootloader to diagnose the problem.

2.8 Flash protection

A bootloader must always be protected from erasing itself or getting corrupted. With flash protection enabled, the worst-case scenario during a botched firmware update is that the application is erased or corrupted, but the bootloader is still intact. The bootloader can then run again and re-load the new application.

This MSD bootloader uses Freescale’s flash protection features. Depending on the part used, the microcontroller has a non-volatile register that specifies which flash sectors are protected. The bootloader is stored in protected sectors and the application is stored in unprotected sectors. These non-volatile register settings are part of the bootloader and cannot be changed by the application. In addition, this flash protection also protects the reset and interrupt vectors from being changed by the application.

Here is a list of the flash protection registers by core and the sectors protected. Refer to Table 4 to see how the flash memory usage is affected. These protection values can be modified in Bootloader.h.

- ColdFire V2 — Register CFMPROT at 0x408 is set to 0x7 and protects the flash address range 0x0 to 0xBFFF.
- ColdFire V1 — Register NVPROT at 0x40D is set to 0xD7 and protects the flash address range 0x0 to 0x9FFF.
- Kinetis K60N512 — Register FPROT at 0x408 is set to 0xFFFFFFF8 and protects the flash address range 0x0 to 0xBFFF.

NOTE

If the bootloader is modified, these flash-protected sectors can be increased or decreased to properly protect the bootloader. If the address range is changed, change the above registers, and also change the memory map in the linker file and in Bootloader.h.

2.9 Redirecting interrupt vectors

The application cannot change the default interrupt vector table of the microcontroller because that flash memory is protected. Therefore, the application needs to store its interrupt vector table in application flash memory and then move the vectors to RAM. By redirecting in this way, the application can update these vectors. See Redirecting interrupt vectors for more specific details for each device.

2.10 Stack and RAM usage

The bootloader interacts with the application only when it starts the application. The application never calls functions in the bootloader. Therefore, the bootloader and application do not need to have separate RAM memory, and can both use the same physical RAM. From the application's perspective, the bootloader essentially uses no RAM because the application has access to the entire RAM in the device. The linker files are set up to share the physical RAM between the bootloader and the application.

2.11 Bootloader memory maps

The next subsections show the memory maps for the three different device examples. The memory map for both the bootloader and application are shown to see how the RAM overlaps. Notice that the application has access to all RAM and overlaps the bootloader RAM. All other memory sections are identical between the two.

2.11.1 ColdFire V2 bootloader memory map

Table 1. MCF52259 bootloader memory map

Addresses	Bootloader	Application
0x0000_0000 to 0x0000_03FF	Bootloader interrupt vectors	Bootloader interrupt vectors
0x0000_0400 to 0x0000_041F	Flash protection and security registers	Flash protection and security registers
0x0000_0420 to 0x0000_BFFF	Bootloader flash (48 KB)	Bootloader flash (48 KB)
0x0000_C000 to 0x0000_C3FF	Application stored interrupt vectors	Application stored interrupt vectors
0x0000_C400 to 0x0000_C41F	Application flash protection and security (not used)	Application flash protection and security (not used)
0x0000_C500 to 0x0007_FFFF	Application flash memory (463 KB)	Application flash memory (463 KB)

Table continues on the next page...

Table 1. MCF52259 bootloader memory map (continued)

Addresses	Bootloader	Application
0x0008_0000 to 0x1FFF_FFFF	Reserved	Reserved
0x2000_0000 to 0x2000_03FF	Redirected interrupt vector table in RAM	Redirected interrupt vector table in RAM
0x2000_0400 to 0x2000_FFFF	RAM available for bootloader	RAM available for application

2.11.2 ColdFire V1 bootloader memory map

Table 2. MCF51JM128 bootloader memory map

Addresses	Bootloader	Application
0x(00)00_0000 to 0x(00)00_03FF	Bootloader interrupt vectors	Bootloader interrupt vectors
0x(00)00_0400 to 0x(00)00_040F	Flash protection and security registers	Flash protection and security registers
0x(00)00_0410 to 0x(00)00_9FFF	Bootloader flash (39 KB)	Bootloader flash (39 KB)
0x(00)00_A000 to 0x(00)00_A3FF	Application stored interrupt vector table	Application stored interrupt vector table
0x(00)00_A400 to 0x(00)00_A40F	Application flash protection and security (not used)	Application flash protection and security (not used)
0x(00)00_A410 to 0x(00)01_FFFF	Application flash memory (87 KB)	Application flash memory (87 KB)
0x(00)20_0000 to 0x(00)7F_FFFF	Reserved	Reserved
0x(00)80_0000 to 0x(00)80_01BB	Redirected interrupt vector table in RAM	Redirected interrupt vector table in RAM
0x(00)80_01BC to 0x(00)80_3FFF	RAM available for bootloader	RAM available for application

2.11.3 Kinetis bootloader memory map

Table 3. K60N512 bootloader memory map

Addresses	Bootloader	Application
0x0000_0000 to 0x0000_03FF	Bootloader interrupt vectors	Bootloader interrupt vectors
0x0000_0400 to 0x0000_040F	Flash protection and security registers	Flash protection and security registers
0x0000_0410 to 0x0000_BFFF	Bootloader flash (47 KB)	Bootloader flash (47 KB)
0x0000_C000 to 0x0000_C3FF	Application stored interrupt vector table	Application stored interrupt vector table
0x0000_C400 to 0x0000_C410	Application flash protection and security (not used)	Application flash protection and security (not used)
0x0000_C410 to 0x0007_FFFF	Application flash memory (463 KB)	Application flash memory (463 KB)
0x0008_0000 to 0x1FFE_FFFF	Reserved	Reserved
0x1FFF_0000 to 0x1FFF_03FF	Redirected interrupt vector table in RAM	Redirected interrupt vector table in RAM
0x1FFF_0400 to 0x2000_FFFF	RAM available for bootloader	RAM available for application

2.12 Resource usage

Table 4 shows the memory resource usage of the bootloader in the three different device examples. As discussed in Flash protection, the bootloader is protected in flash memory. The minimum flash protection size was selected to protect the bootloader. Therefore, the amount of flash memory used by the bootloader is dictated by the minimum protection size available in the device.

Table 4. Bootloader resource usage

Device	Flash usage (KB)	Secured flash (KB)	RAM usage (bytes)
MCF52259	33	48	0
MCF51JM128	32	40	0
K60N512	33	48	0

See Stack and RAM usage for an explanation of why the bootloader uses no RAM.

3 Using the bootloader

3.1 Bootloader file structure

This section describes the firmware source code provided for the bootloader and application examples. The source code included with this application note includes the bootloader and examples for all three cores. Figure 3 shows the directory structure of the provided source code.

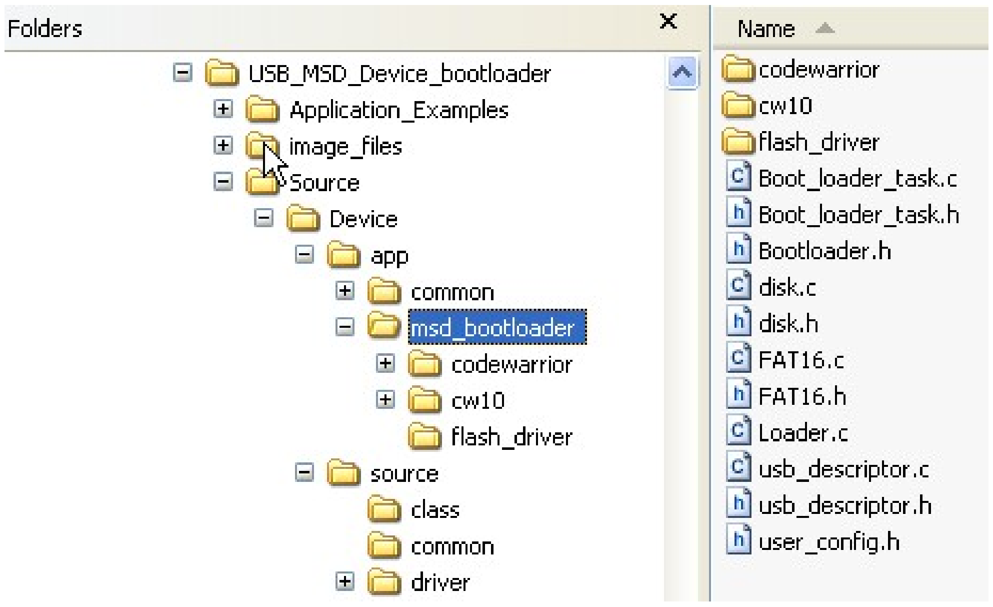

```
Folders
USB_MSD_Device_bootloader
  Application_Examples
  image_files
  Source
    Device
      app
        common
        msd_bootloader
          codewarrior
          cw10
          flash_driver
      source
        class
        common
        driver
```
```
Name
codewarrior
cw10
flash_driver
Boot_loader_task.c
Boot_loader_task.h
Bootloader.h
disk.c
disk.h
FAT16.c
FAT16.h
Loader.c
usb_descriptor.c
usb_descriptor.h
user_config.h
```

Figure 3. Bootloader file structure

- Application_Examples: Contains application example projects to integrate an application with the Bootloader. See Application examples for more details.

- image_files: Contains some example firmware image files of applications to test with the bootloader. See Using the bootloader on a demo board for more details.
- CodeWarrior: Contains projects for CodeWarrior for Microcontrollers v6.3 to use with ColdFire V1 and CodeWarrior for ColdFire v7.2 to use with ColdFire V2.
- cw10: Contains projects for CodeWarrior v10.1 for ColdFire and Kinetis.
- Flash_driver: Contains flash driver source code for many microcontrollers.
- Boot_loader_task.c: Source file for general Bootloader tasks.
- Boot_loader_task.h: Function prototypes
- Bootloader.h: Includes memory map definitions for specified boards.
- disk.c: Contains mass storage disk functions.
- disk.h: Contains disk parameter definitions.
- FAT16.c: Contains pseudo-FAT structures.
- FAT16.h: Contains FAT parameter definitions.
- Loader.c: Contains functions to parse and load firmware image to flash memory.
- usb_descriptor.c: Contains USB descriptor structures and functions.
- usb_descriptor.h: Contains USB descriptor parameters.
- user_config.h: Contains user configurations for USB stack.

3.2 Using the bootloader on a demo board

This section describes how to use the bootloader. It includes how to demonstrate the bootloader on a development board and how to port the bootloader to other platforms.

The included CodeWarrior projects have application examples and image files to test with the TWR-MCF5225X-KIT, DEMOJM, and TWR-K60N512-KIT. Choose a bootloader project to test with the corresponding board. Please refer to the documentation included with these boards to get started and learn how to program the boards with CodeWarrior. After getting familiar with the boards and CodeWarrior, use this procedure:

1. The hardware must be set up for USB device operation. Refer to the board documentation for more details. If using the DEMOJM board, ensure that the proper MCF51JM128 module is plugged into the board. Plug both USB cables into the board and the computer.
2. Open the corresponding CodeWarrior project, then open one of the bootloader project files for one of the application examples. Compile the project and program the flash.
3. Reset the board. The device reboots in bootloader mode and enumerates a new drive in the host computer.
4. Figure 4 shows the bootloader drive in Windows XP. Notice the volume name of the drive is BOOTLOADER, and the status file is READY.TXT. The bootloader is now ready to receive a firmware image file.

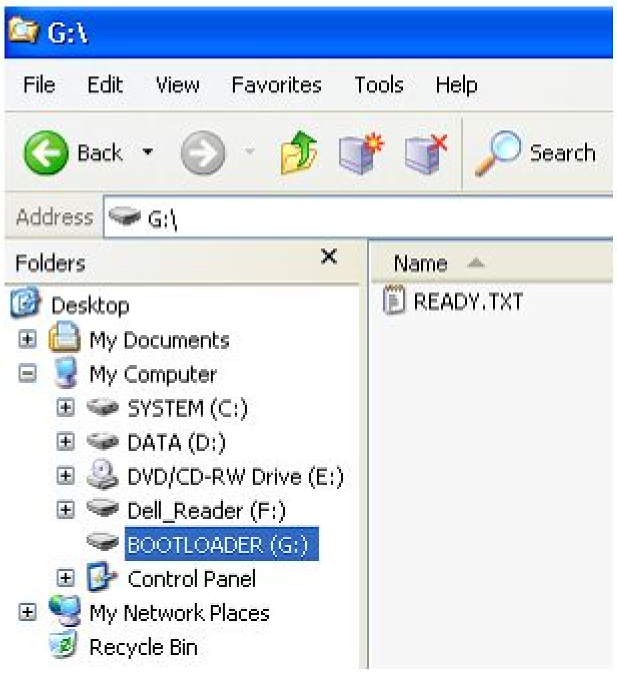


Figure 4. Bootloader drive after enumeration

5. Go to the directory “image_files,” copy one of the image files for that board, and paste it into the bootloader drive. Dragging and dropping the file also works.

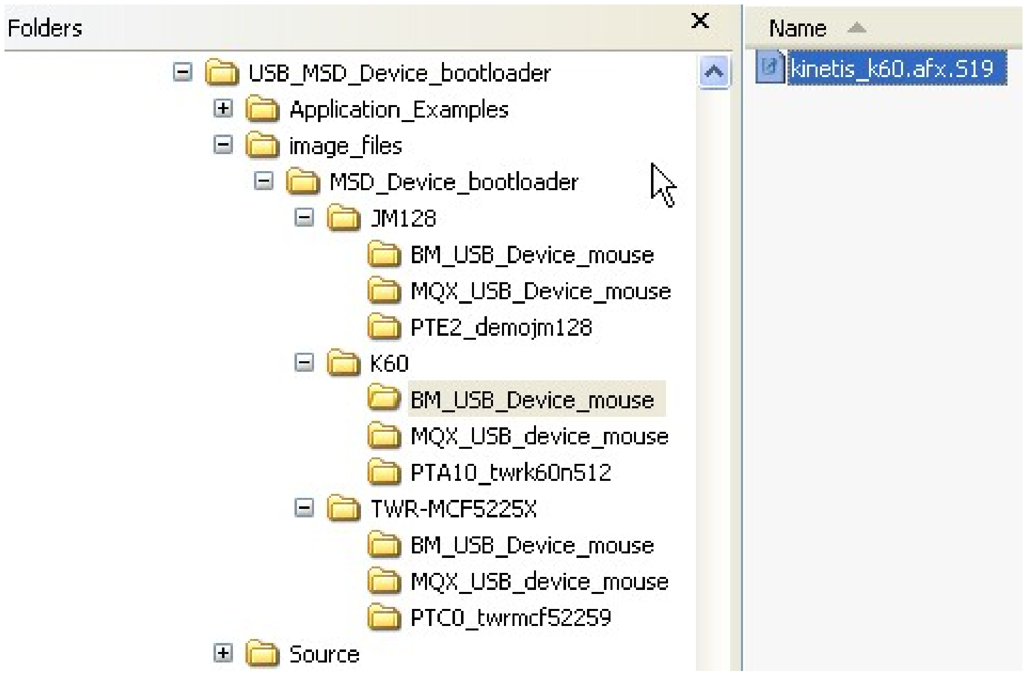


Figure 5. Choose a firmware image file to download

Some operating systems like Linux Fedora do not transfer the file to the drive until the drive is unmounted. If using an operating system like this, unmount the bootloader drive.

6. Check the bootloader file status. After the file transfer is complete, the bootloader drive disappears. After a few seconds, the drive re-enumerates and re-appears. Click on the bootloader drive to verify the status file is SUCCESS.TXT. The firmware update is then complete.

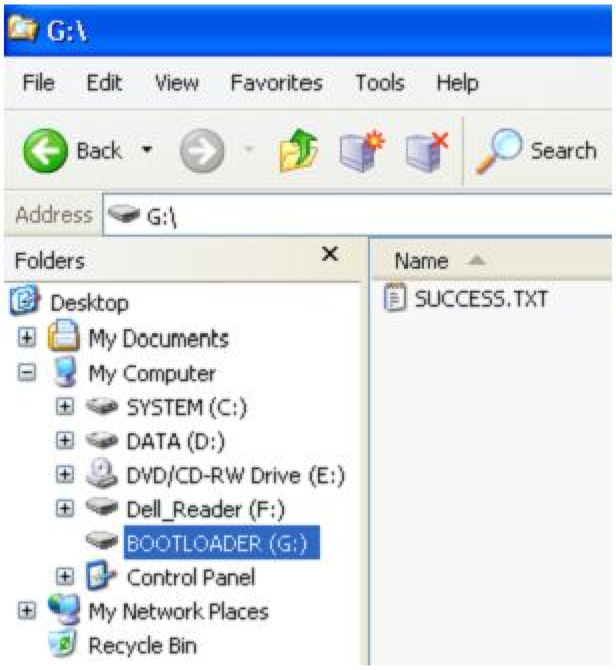


Figure 6. Bootloader drive with successful status file

7. Reset the board using the Reset button. The new application executes.

After an application is loaded, bootloader mode can be forced by holding down a button during reset. Here is the button to use, depending on the board:

- TWR-MCF5225X-KIT — SW1
- DEMOJM — PTG1
- TWR-K60N512-KIT — SW1

3.3 Prevent linking between bootloader and application

It is important to understand that good bootloaders should only interact with the application through absolute addresses. In this case, the application reset vector resides at an absolute address, and the vector points to the entry point of the application. The bootloader knows this absolute address, and to enter application mode it will use the application reset vector to jump to the beginning of the application. This is important because every time the application is re-compiled, the linker might change the address of the entry point. The bootloader does not need to know the changed address since the application reset vector will always be at the same location and will always point to the application entry point.

Using a single project for both bootloader and application has the advantage of causing the device to be programmed only once in production, as well as making debugging easier. However, having a single project also has some potential hazards if the bootloader and application are linked together. It is important to prevent the linker from letting one section refer to a non-absolute address in the other section.

Consider this scenario: a bootloader is used with an ANSI library, and compiled in the same project as an application that uses the same library. RevA of the firmware is compiled, and a library function, called MyFunc(), is linked into the bootloader section at address 0x1000. The application also uses MyFunc(), and jumps into the bootloader when executing it. RevA is released for production and everything works well. After some time passes, the firmware is updated to RevB. When re-compiling the project, the linker moves MyFunc() to address 0x1005. The bootloader and application for RevB are linked together, so this version works well as is. Now the firmware update for RevB is downloaded into a device with the RevA bootloader. The application is updated to RevB, but the bootloader is still RevA. When the application executes and jumps to MyFunc(), it jumps to address 0x1005. But MyFunc() in the RevA bootloader is at address 0x1000, and the code will run away.

This example shows why linking between the application and the bootloader should always be avoided except through absolute addresses. The only interaction this bootloader has with the application is when the bootloader launches the application, which is done using the absolute address of the application reset vector. If the bootloader is included in the application project, the linker should link only to a binary image of the bootloader, not to the source code or a library. With this method, the linker will not be aware of the functions in the bootloader when compiling the application.

4 Porting USB MSD device bootloader to other platforms

This section provides a high-level overview of how to port the USB MSD device bootloader to other Freescale devices, with these assumptions:

- The device is supported by a USB stack with MSD class.
- There is a simple flash driver for this device.

Perform these steps to port the bootloader to other platforms:

1. Create a new project under `\USB_MSD_Device_bootloader\Source\Device\app\msd_bootloader\codewarrior\` or `\USB_MSD_Device_bootloader\Source\Device\app\msd_bootloader\cw10\` directory.

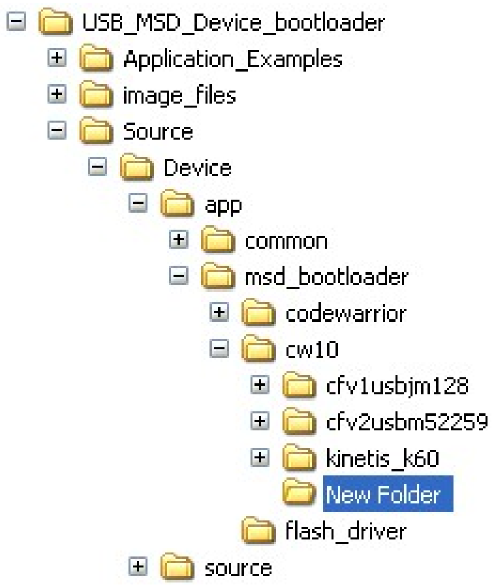


Figure 7. Create new project folder

2. Create a project with a file structure that is similar to an example bootloader project. The CW10 M52259EVB project is shown below.

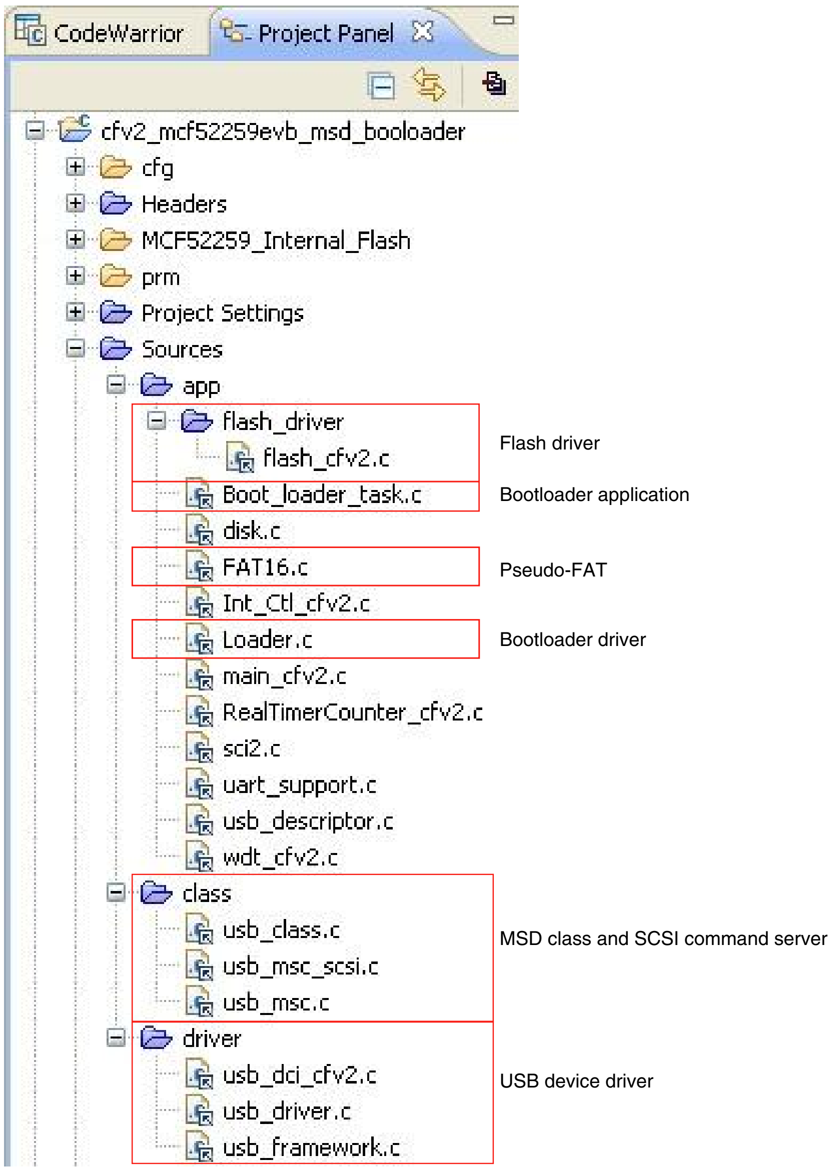


Figure 8. M52259EVB bootloader project

3. Add files to project:
   - Flash driver source code
   - MSD class and SCSI command server source code
   - USB device driver source code
   - `disk.c, disk.h, Boot_loader_task.c, Boot_loader_task.h, Loader.c, Bootloader.h, FAT16.c, FAT16.h, usb_descriptor.c, usb_descriptor.h,` and other files that are specific to each board.
4. Modify the linker command file to segment the bootloader and application, and allow the bootloader to be in protected flash.
5. Modify Boot_loader_task.c file for the specific board.
6. Modify the memory map in Bootloader.h file. The code is shown here:

```
#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)0xC000)
#define PROT_VALUE0x7     // Protects 0x0 - 0xBFFF
```

```
#define ERASE_SECTOR_SIZE       (0x1000)  /* 4K bytes*/
#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 PROT_VALUE0xD7  // Protects 0x0 - 0x9FFF
#define ERASE_SECTOR_SIZE       (0x0400)  /* 1K bytes*/
#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)0xC000)
#define PROT_VALUE00xFF   // Protects 0x0 - 0xBFFF
#define PROT_VALUE10xFF   // Protects 0x0 - 0xBFFF
#define PROT_VALUE20xFF   // Protects 0x0 - 0xBFFF
#define PROT_VALUE30xF8   // Protects 0x0 - 0xBFFF
#define ERASE_SECTOR_SIZE       (0x800)  /* 2K bytes*/
#endif
```

5 Developing new applications

This section describes how to modify applications to work with the USB MSD device bootloader.

5.1 Modify linker command files

Normally, an application will be located at the beginning of flash memory. However, when the bootloader is used it is located at the beginning of flash. The application must be placed after the bootloader. Therefore, the application linker file must be modified to place the application at a specified memory region.

5.1.1 CFV1 linker file

Example of original application linker file:

```
# 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 be compatible with the bootloader, the application code should start after the protected flash and application stored vector table, from address 0xA410. Also, the RAM used for the redirected vector table should be excluded from application RAM. In this case, RAM should start at 0x8001BC. The modified linker file should look like this:

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

# Memory ranges

MEMORY {
   code        (RX)  : ORIGIN = 0x0000A410, LENGTH = 0x0001BBF0
   userram     (RWX) : ORIGIN = 0x008001BC, LENGTH = 0x00003E44
}
```

5.1.2 CFV2 linker file

Example of original application linker file:

```
# 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 be compatible with the bootloader, the stored application interrupt vectors (vectorrom) need to start after the protected flash memory, from address 0xC000. The application code should start after the application stored vector table, from address 0xC500. The modified linker file should look like this:

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

KEEP_SECTION {.vectortable}

# Memory ranges

MEMORY {
   vectorrom    (RX)   : ORIGIN = 0x0000C000, LENGTH = 0x00000400
   cfmprotrom   (RX)   : ORIGIN = 0x0000C400, LENGTH = 0x00000020
   code         (RX)   : ORIGIN = 0x0000C500, LENGTH = 0x00073B00
   vectorram    (RWX)  : ORIGIN = 0x20000000, LENGTH = 0x00000400
   userram      (RWX)  : ORIGIN = 0x20000400, LENGTH = 0x00005C00
}
```

5.1.3 Kinetis linker file

Example of original application linker file:

```
# Default linker command file.
MEMORY {
m_interrupts  (RX) : ORIGIN = 0x00000000, LENGTH = 0x000001E0
m_text        (RX) : ORIGIN = 0x00000800, LENGTH = 0x00080000-0x00000800
m_data        (RW) : ORIGIN = 0x1FFF0000, LENGTH = 0x00020000
m_cfmprotrom  (RX) : ORIGIN = 0x00000400, LENGTH = 0x00000010
}
```

To be compatible with the bootloader, the stored application interrupt vectors (vectorrom) need to start after the protected flash memory, from address 0xC000. The application code should start after the application stored vector table, from address 0xC410. Also, the RAM used for the redirected vector table should be excluded from application RAM. In this case, RAM should start at 0x1FFF_0400.The modified linker file should look like this:

```
# Default linker command file.
MEMORY {
m_interrupts  (RX) : ORIGIN = 0x0000C000, LENGTH = 0x00000400
m_cfmprotrom  (RX) : ORIGIN = 0x0000C400, LENGTH = 0x00000010
m_text        (RX) : ORIGIN = 0x0000C410, LENGTH = 0x00073BF0
m_data        (RW) : ORIGIN = 0x1FFF0400, LENGTH = 0x0001FC00
}
```

5.2 Redirecting interrupt vectors

The default locations of the interrupt and exception vectors are in a flash memory area protected by the bootloader, so that the application cannot update them when doing a firmware update. For the application to use interrupts, it needs to redirect the interrupt vectors into RAM.

This section describes how to redirect the interrupt vectors to RAM with Freescale MQX and the Freescale USB stack.

5.2.1 Freescale MQX

The Freescale MQX RTOS has the ability to configure applications to implement interrupts in RAM or ROM. By setting the macro MQX_ROM_VECTORS to zero in user_config.h, the MQX kernel will redirect the vectors to RAM. Add the line of code below to user_config.h, then rebuild the BSP and PSP. Refer to the MQX documentation for details of how to do this.

Configure the parameter to zero to implement interrupts in RAM:

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

5.2.2 Bare-metal applications

The following sections describe how to redirect a vector table to RAM for CFV1, CFV2, and Kinetis families.

5.2.2.1 CFV1 microcontrollers

CodeWarrior projects are typically configured for ease of use, with the vector table fixed at the default location in flash memory at address 0x0000. Therefore, a new vector table can be created in the application to store the interrupt vectors in the application section of flash memory.

An example vector table is provided below that can be included in the application. The whole vector table is not shown here, but the full table is available in the application software example provided. This vector table should be located at the beginning of unprotected flash, in this case 0xA000, because the bootloader will look for the application reset vector at that address.

Just as with the default vector table, vector 0 should have the initial stack pointer value and vector 1 needs the application reset vector. This example fills the rest of the vectors with the dummy interrupt service routine dummy_ISR. To add a new ISR, replace the address of the dummy_ISR with that of the new ISR, like Timer_Overflow shown below. The new vector table in the application is declared by using this code:

```
// Declare the stored vector table for the application,
// and locate it at the beginning of unprotected flash
void  (* const RAM_vector[])()@0xA000= {
     (pFun)&__SP_INIT,               // vector_0  INITSP
     (pFun)&_startup,                // vector_1  INITPC
     (pFun)&dummy_ISR,               // vector_2  Vaccerr
     (pFun)&dummy_ISR,               // vector_3  Vadderr
     (pFun)&dummy_ISR,               // vector_4  Viinstr

…

     (pFun)&dummy_ISR,               // vector_74 Vtpm1ch3
     (pFun)&dummy_ISR,               // vector_75 Vtpm1ch4
     (pFun)&dummy_ISR,               // vector_76 Vtpm1ch5
     (pFun)&Timer_Overflow,               // vector_77 Vtpm1ovf
     (pFun)&dummy_ISR,               // vector_78 Vtpm2ch0
     (pFun)&dummy_ISR,               // vector_79 Vtpm2ch1
```

```
    (pFun)&dummy_ISR,              // vector_80 Vtpm2ovf

…

    (pFun)&dummy_ISR,              // vector_108 VL3swi
    (pFun)&dummy_ISR,              // vector_109 VL2swi
    (pFun)&dummy_ISR,              // vector_110 VL1swi
};
```

In the application, this table should be copied to the base of RAM. This code copies the above table (RAM_Vector) to the vector table region in RAM:

```
// Copy stored application interrupt vectors
// to re-directed vector table in RAM
pdst=(uint_32_ptr)0x00800000;
psrc=(uint_32_ptr)&RAM_vector;
  for (i=0;i<111;i++,pdst++,psrc++)
  {
    *pdst=*psrc;
  }
```

CFV1 has a register named Vector Base Register (VBR) that contains the base address of the exception vector table. This register is used to relocate the exception vector table from its default position in flash memory (address 0x0) to the base of the RAM (0x800000).

```
// Move vector table to start of RAM, 0x800000
asm (move.l  #0x00800000,d0);
asm (movec  d0,vbr);
```

5.2.2.2 CFV2 microcontrollers

5.2.2.2.1 Freescale USB stack

With the CFV2 version, the Freescale USB stack includes a function to copy the interrupt vector table to a specified area in RAM.

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

This function copies the interrupt vector table to RAM at __VECTOR_RAM address. This address is defined in the linker command file. It also moves the vector table to RAM by changing the VBR.

This function is called at startup by default so the application does not need to call this function.

5.2.2.2.2 Other bare-metal applications

For other applications, the application needs to copy the stored vector table from flash to RAM and move the vector table by changing VBR. This code can be used to copy the vector table:

```
// Copy Application Stored Interrupt Vector table to RAM
pdst=(uint_32*)0x20000000;
psrc=(uint_32*)0xC000;
  for (i=0;i<0x100;i++,pdst++,psrc++)
  {
    *pdst=*psrc;
  }
```

CFV2 has a register named Vector Base Register (VBR) that contains the base address of the exception vector table. This register is used to relocate the exception vector table from its default position in the flash memory (address 0x0) to the base of the RAM (0x2000_0000).

The following code is used to redirect vector table to the RAM with address 0x2000_0000.

```
// Move vector table to RAM
asm(move.l#0x20000000, d0);
    asm(movec d0,VBR);
asm(nop);
```

5.2.2.3 Kinetis microcontrollers

In Kinetis, the SCB_VTOR register contains the base address of the exception vector table. To redirect the vector table, the vector table has to be copied to RAM, then the value of SCB_VTOR changed to the copied address. These steps describe the way to redirect the vector table for Kinetis.

Implement this code to copy the interrupt vector table to RAM:

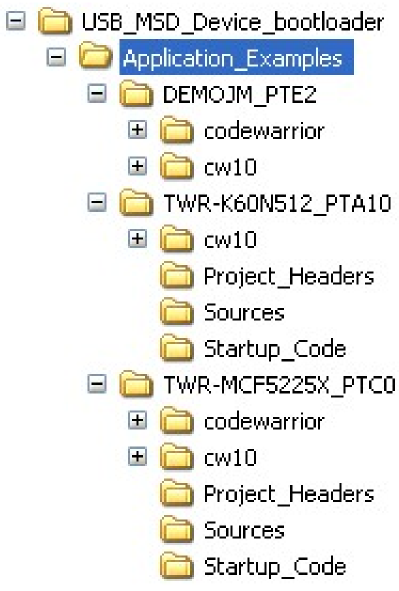
```
// Copy Application Stored Interrupt Vector table to RAM
pdst=(uint_32*)0x1FFF0000;
psrc=(uint_32*)0xC000;
  for (i=0;i<0x100;i++,pdst++,psrc++)
  {
    *pdst=*psrc;
  }
```

Then redirect the vector table to RAM by changing the SCB_VTOR like this:

```
// Redirect the vector table to the new copy in RAM
  SCB_VTOR = (uint_32)0x1FFF0000;
```

5.3 Application examples

This software comes with some simple application examples to show how to create a project for an application integrated with the bootloader. These examples use a periodic interrupt to toggle an LED on the board, and show how to redirect the interrupt vector table. The binary images of these example projects are also provided in the “image_files” directory to be used when evaluating the bootloader.



```
USB_MSD_Device_bootloader
  Application_Examples
    DEMOJM_PTE2
      codewarrior
      cw10
    TWR-K60N512_PTA10
      cw10
      Project_Headers
      Sources
      Startup_Code
    TWR-MCF5225X_PTC0
      codewarrior
      cw10
      Project_Headers
      Sources
      Startup_Code
```

Figure 9. Application example projects

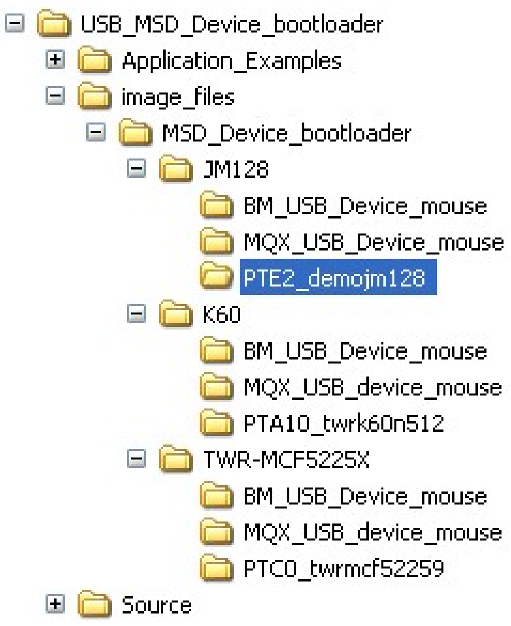


Figure 10. Application example image files

There are three example projects provided, one for each of the evaluation boards supported. Each example is provided with CodeWarrior projects. These projects can run in the debugger, but they require the bootloader to be present in the flash when booting without the debugger, since the application vector table is not at the default reset vector location. The example applications include:

- DEMOJM_PTE2 — for Flexis ColdFire V1 MCF51JM128 running on the DEMOJM board. Toggles pin PTE2 to blink LED.
- TWR-K60N512_PTA10 — for Kinetis MK60N512 running on the TWR-K60N512-KIT. Toggles pin PTA10 to blink LED E4.
- TWR-MCF5225X_PTC0 — for ColdFire V2 MCF52259 running on the TWR-MCF5225X-KIT. Toggles pin PTC0 to toggle LED1.

6 Conclusion

Freescale's USB Mass Storage Device Bootloader allows for very easy firmware updates. It does not require a driver, does not require any software on the host computer, and can be performed by anyone. This bootloader also works with a broad selection of Freescale microcontrollers. The information in this application note should allow any application to be integrated with the bootloader.

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

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4379
Rev. 0, October 2011