USBFS Bootloader Datasheet BootLdrUSBFS V 1.60

Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	Flash	RAM	
CY8C24x94, CY8CLED04, CY7C64215, CY8C20x66, CY8C20x36, CY8C20x46, CY8C20x96, CY8C20xx6AS, CY8C20XX6L, CY7C643xx, CYONS2000, CYONS2110, CY8CTST120, CY8CTMG120, CY8CTMA120, CY8CTST200, CY8CTMG2xx						
HID support	0	0	0	4615-4982	46	2
Without HID support	0	0	0	4516-4982	46	2

Note Expect an expansion of flash and RAM when adding interfaces, HID classes, and other USBFS extensions. When the bootloader is in actual operation, it uses a large amount of RAM to download program data, but frees it upon exit. Since the bootloader operation prevents application operation, this RAM requirement is essentially invisible. ROM/flash usage includes a complete USB interface. Additional code used for the bootloader function is only 2 kilobytes above the normal requirement of about 1.9 kilobytes of code used by USB.

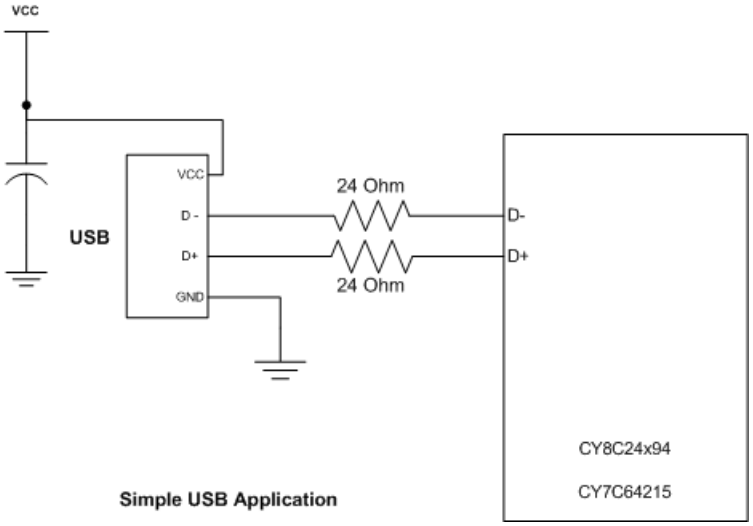
Features and Overview

- Flexible memory map
- Device reprogramming without engineering tools
- Product resident reprogrammability
- Communication interface integrated to minimize code overhead
- Field deployment of firmware upgrades
- USB Full Speed device interface driver
- Support for interrupt and control transfer types
- Setup wizard for easy and accurate descriptor generation
- Runtime support for descriptor set selection
- Optional USB string descriptors
- Optional USB HID class support

The USB bootloader supports fully functional device reprogramming with built-in error detection and an industry standard communication interface.

Multiple USB device descriptors reside in the system to allow commanding a running device to self reconfigure and reprogram. Core USB functions are maintained during the reconfiguration to support host communication, while program data is transferred and stored. At the end of the reconfiguration process, the device resets itself, verifies the new program, and automatically executes it.

Figure 1. USBFS Bootloader Block Diagram



Quick Start

1. For a successful implementation of a bootloader project, review this datasheet.
2. Add the user module to a project.
3. Place the user module, selecting either a HID or non-HID class application.
4. Right click on the user module icon. Select **Boot Loader Tools**. Select **Get Files**. After this step is completed, the boot.tpl, custom.lkp, HTLinkOpts.lkp, and flashsecurity.example files must be in the project root directory.
5. Right click on the user module icon. Select **Device > Application USB Setup Wizard…** Verify that there is at least one string in the **Strings** area. At least one string must be present by default; if not, add a one-character string. Select **OK**.

   Note If an HID class application is selected in step 3, the following setup is required. For non-HID applications, skip the following setup and go to step 6.

   - Click the **Import HID Report Template** operation.
   - Select the 3-button mouse template.
   - Click the **Apply** operation on the right side of the template.
   - Edit the HID class descriptor: select the 3-button mouse for the HID Report field.
   - Click **OK** to save the USB descriptor information.
6. Right click on the user module icon. Select **Device > Bootloader USB Setup Wizard…** It is not necessary to make any modifications. Select **OK**.
7. (Image Craft compiler only) From the **Project > Settings > Linker** dialog box, set the **Relocatable code start address** to the value obtained by multiplying the ApplicationCode_Start_Block X device block size, to avoid unintentionally attempting to create an application code in the bootloader ROM area. If these settings leave unused ROM areas, the settings can be optimized later. The linker gives unhelpful messages when it encounters memory overlap errors. Initial project development is less frustrating if linker problems are kept to a minimum. Setting the Relocatable Code Start Address to approximately 0x1700 should be correct for most default settings. For example, for CYONS2xxx

(which has a 0x80 byte block size), it is 0x2e*0x80 = 0x1700, while for CY8C24794 (0x40-byte block size), it is 0x5c*0x40 = 0x1700.

8. Generate source code and compile the project.
9. Review the output file *<project>.mp* and *<project>.hex* to see how the project is built.
10. After creating a project that compiles without errors, go to the Sample Firmware Code section. Modify and adapt the code given in the sample.

Functional Description

The USBFS User Module gives:

- A USB Full Speed Chapter 9 compliant device framework.
- A low level driver for the control endpoint that decodes and dispatches requests from the USB host.
- A USBFS Setup Wizard to enable easy descriptor construction.

You have the option of constructing an HID based device or a generic USB device. Make your choice when you select the USBFS User Module. To change your choice after initial selection, delete the existing instance of the USBFS User Module and then add a new instance.

The bootloader portion of the user module gives a method to organize the memory map and major code functional blocks into areas that are compatible with device reprogramming. The memory organization of the project is considerably different from that of a conventional PSoC Designer™ project. Modifications to the memory map are necessary to meet the minimum device functionality requirements while the device application is being reprogrammed. Effectively, a project incorporating a bootloader contains two independent programs supporting different functions. A map of the memory is shown in Figure 2.

After a project incorporating a bootloader is deployed, the memory locations highlighted in red are never reprogrammed. The application code and the checksums may be altered by running the bootloader. With the exception of the first two blocks of program space, you can move the other major functional groups of code to locations you determine.

In addition to the parameters that are adjusted within the user module, two other important features are given. You can access a built-in set of tools by right-clicking on the bootloader icon in the device manager view. Additionally, a host application (including source code) is given along with instructions on how to set it up and use it on a system to demonstrate bootloader capability.

Further information about USB, including specifications, resource examples, and forums regarding use of USB are available at www.usb.org.

Theory of Operation

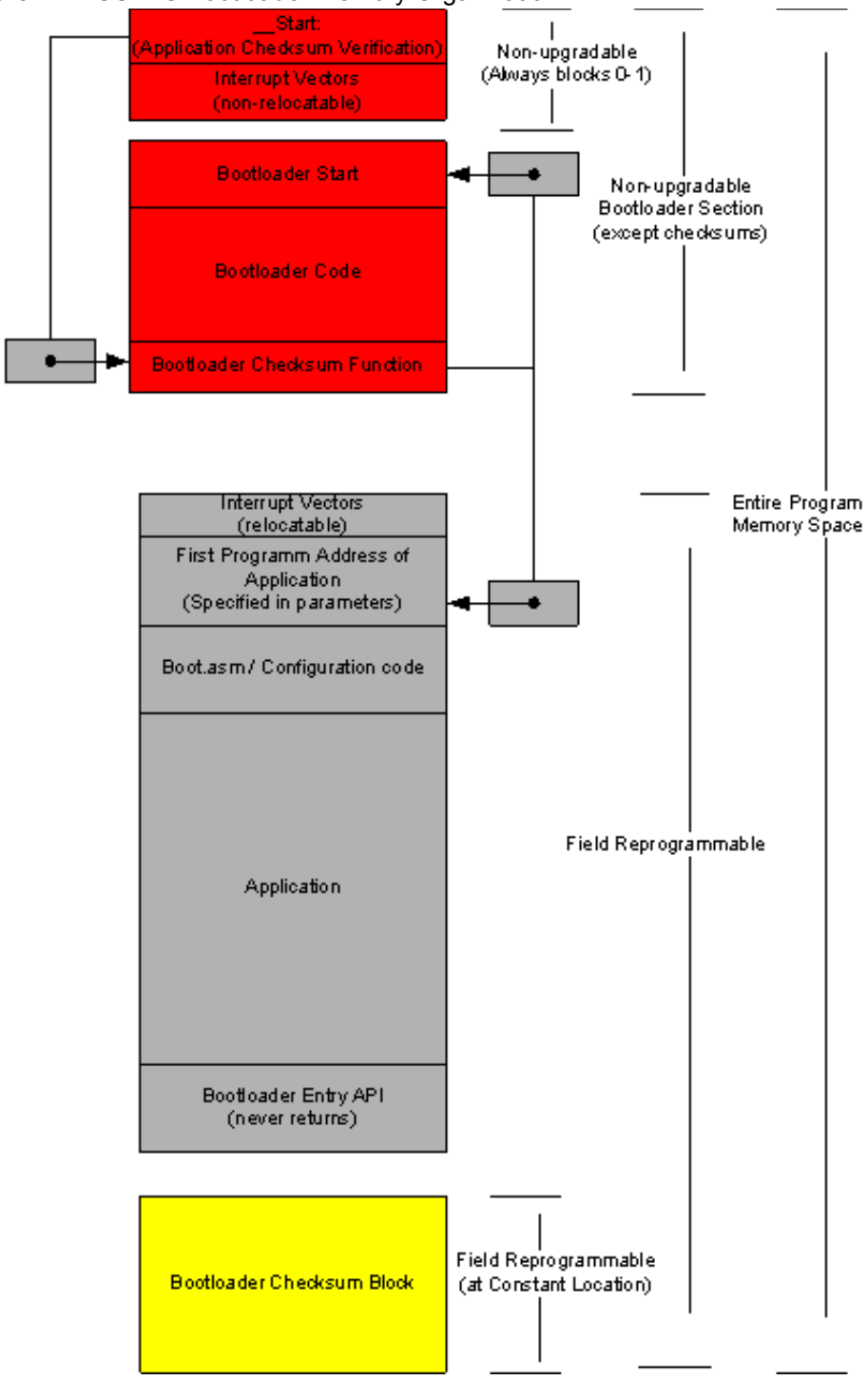
Creating a project with a bootloader requires several nonstandard modifications to the PSoC Designer standard model. To facilitate this, the Bootloader User Module gives customized files and specialized tools to assist you in bootloader project development. The special tools are accessed by switching to the Device Editor view and right clicking the USBFS Bootloader User Module icon. In addition to the tools and files given as part of the user module, a host application example is also given with the user module installation that can demonstrate basic capability of the bootloader. This PC-based application and source code for Microsoft Visual Studio® 2005 is contained in a .zip file in the installation directory of PSoC Programmer 3.

<install_path>\Cypress\Programmer\3.xx\Bootloaders\BootLdrUSBFS\USB_BootLoaderHostApp\…

Using this application requires installation and limited customization of a generic USB driver capable of supporting the host demonstration application. This file is supplied as part of the installation and may be registered upon initial operation of a bootloader device. Use Windows' manual installation method to specify the location of the driver contained in the "\USB driver" directory of the location specified earlier.

The included driver.inf file must be modified to correctly specify the VID and PID of your chosen bootloader device. Note that this change must be made in two locations within the driver.inf file: one location is near the top of the file and the second is near the bottom.

Figure 2. USBFS Bootloader Memory Organization



USBFS Bootloader Memory Organization

PSoC Designer uses standardized files, built-in data about the part family, and attributes of specific devices to create projects that can be compiled and correct API definitions. A project with a bootloader requires a memory map that is considerably different from that of a standard PSoC Designer project. Selection of the memory areas represents a core design decision that is maintained throughout the life of the design. A project without the requirements of a bootloader simply allows the compiler and linker to allocate RAM and ROM. However, a bootloader must group RAM and ROM in specific areas so that the program does not crash while a new application is being loaded.

In the memory layout shown in Figure 2, there are five key areas of ROM that are managed:

- The first is blocks 0 and 1 of ROM. These blocks contain critical interrupt vectors and restart vectors. Since it is nearly impossible to control read access to these blocks by any operating device, they are never erased and reprogrammed. The first two blocks of ROM must not be modified and cannot be placed in any other location.
- The second area of memory to be defined is the area containing the bootloader code itself. This area may be placed at different ROM locations. However, after the project or device containing the bootloader is deployed, this area is not reprogrammable and cannot be field upgraded.
- The third memory area is the relocatable interrupt table. This table may consist of one or two blocks depending on the architecture of the device. This area contains interrupt and general purpose vectors to give a jump table for interrupts or code entries that may be altered when a new application is loaded using the bootloader. For example, this area contains the application start address. The bootloader can use this address to start the new application after the checksum is validated at power up. You can determine the location of this area at design time. After the application and bootloader is deployed, this area may be rewritten but its location must not be modified. The characteristics of this area are similar to the checksum area described later in this section.
- The fourth memory area is the application area. The application area contains a set of interrupt vectors that are reprogrammable with the condition that they are never accessed when they are being rewritten. Consider the problems that a program might encounter if the executing code is changed during execution. This requirement is easily met by turning off all application interrupts during bootload. This is automatically done when the bootloader starts. In addition to interrupt vectors, the application area also contains most of the device bootcode and all of the foreground runtime application.
- The fifth area of ROM defined is the checksum area. This area contains important data that the bootloader software uses to download and verify the foreground application. The checksum area contains the start address and size in blocks of the foreground application. The first two bytes of the checksum block are a checksum of the checksum block itself, and the last two bytes are the checksum of the runtime application. The structure of the checksum block contains space for you to define your own data, in addition to that used by the bootloader. This structure is exposed as a C-structure definition, and may be modified as long as data used by the bootloader utility is not changed or repositioned within the block.

If your application has some code that must be always operational, including during a bootload process, the design of the Bootloader User Module can allow sufficient customization to accommodate this. This is accomplished by adding the code to the bootloader ROM area using the assembler AREA directive. Any RAM used by your code during the bootload process must be added to the RAM area defined for the bootloader.

Definition of Memory Areas in the User Module Parameters

The USBFS Bootloader User Module parameters enable you to customize where major program elements are placed in ROM. The defaults in the user module give an initial working setup. Use these settings until a complete project is successfully compiled. After compiling a project, you can look at the program memory map and .hex output file to determine how to optimize your program structure. If you reconfigure the parameters and accidentally create memory area conflicts, it may be difficult to determine the correct locations without a valid memory map to look at.

Bootloader Utility

The Bootloader User Module gives a complete utility that coexists with your foreground application. When the device is started or reset, the bootloader utility is always invoked. Once invoked at system startup, the bootloader validates the foreground application by calculating a checksum on the foreground application ROM area. The calculated checksum is compared to the one stored in the checksum block (which is created with the application). If the two checksums are equal, the bootloader utility allows the foreground application to execute. If the two checksums are not equal, the bootloader enters a wait loop and waits for a host application to download a valid application. It also enables its own USB subsystem to allow the host to transmit data. When the host system observes that this interface is enabled, it may choose to execute its own set of applications. Although the default USB descriptor that is provided runs successfully with the examples given, you may choose to alter any of the parameters on the host or PSoC device. Source code for VisualStudio 2005 is included for the host application. An example application and source code is given in the installation directory for PSoC Programmer 3.
<install_path>\Cypress\Programmer\3.xx\Bootloaders\BootLdrUSBFS\USB_BootLoaderHostApp\…

Bootloader Tools

Several tools are available from the shortcut menu and are accessed by right clicking on the user module icon.

Special versions of boot.tpl, custom.lkp, and HTLinkOpts.lkp can be placed in the project or removed. From the main menu select **Tools** > **Restore Default Boot files**. If you remove the USBFS Bootloader User Module, the option to restore the default boot files moves to the File menu in PSoC Designer.

Generate Checksum – After your project builds correctly you can use the bootloader tools to create and autovalidate checksums. Upon entry into the bootloader ‘tools selection’ screen, the project code is generated and a complete compile of the entire project is executed. Then a checksum calculation is performed on the resulting hex file, which is compared to a checksum stored by the user module. If the checksums do not match, a message is displayed. You can recalculate and store a new checksum if you wish. If build or compile errors occur in the automated generate and build invoked by the Bootloader Tools, and no hex file is successfully created, an error is reported but no error debug information is displayed in the build dialog of PSoC Designer. Error reporting is suppressed when the generate and build is invoked from the automation interface. To debug build errors, it is necessary to use the conventional build and generate process external to the bootloader tools menu.

Generate dld file – This tool item derives a download file from the hex project output file. This file contains only the hex blocks that are reprogrammed by the bootloader including the checksum block. The Host Demonstration application is capable of reading this file and downloading it to a working project incorporating a bootloader. This download file can be deployed to a field application to upgrade a PSoC device.

The dld download file is generated by the Bootloader User Module tool and listed in “Output Files” in Workspace Explorer.

Checksum Semiautomatic Generation

After your project is built and compiled without errors, the application checksum must be generated. The application checksum is created using one of the utilities available by right-clicking on the Bootloader User Module Icon in the Device Editor view and selecting **Bootloader Tools**. An application checksum (previously calculated or default) is stored as a hidden user module parameter. When the Bootloader Tools menu page is invoked, any previous checksum is validated against the one calculated on the current output\<prj_name>.hex file. Necessarily, the checksum cannot be generated before a successful compile. After a checksum is created, it must be integrated into the compiled files. This requires a second compile. Lastly, to validate the block of code that the checksum is stored in, a checksum on that block itself is given. This requires a third compile. Multiple compiles are required because the built in checksum parameters are precompile elements that can only be calculated post-compile.

Special Files Given

You can access several important files by opening the **Bootloader Tools** menu and selecting **Get Files**. A device specific boot.tpl file is placed in the main project directory along with files called custom.lkp (ImageCraft), HTLinkOpts.lkp (Hi Tech) and a predefined flashsecurity.txt file. Each file is briefly described here (the original versions of these files are placed in the project backup directory):

Boot.tpl. – This file contains a relocatable and nonrelocatable definition of interrupt vector tables and device specific boot setup that is specified in a relocatable area of ROM rather than the fixed location specified in the standard boot.tpl file.

Custom.lkp – When source generation takes place, the custom.lkp file is populated with autogenerated ROM areas for major code blocks as defined in the user module parameters. Do not modify the code blocks in the custom.lkp file, named:

- -bSSCParmBlk – Contains specified critical RAM used during flash operations.
- -bBootloader
- -bBLChecksum
- -bUserAPP – Changes to any of the last three lines will result in an error dialog indicating the inability of the project to detect the correct custom.lkp file.

During code generation, each of the last three lines of the custom.lkp file are rewritten under control of the code generation software. Changes made within or below the last three lines either cause an error or are simply lost. You can make changes to the rest of the custom.lkp file. To debug the memory allocation of the project, you can comment out all three lines mentioned earlier by inserting a semicolon in the first space. This allows the linker to place code automatically and may be helpful in determining application code size requirements.

HTLinkOpts.lkp – When source generation takes place, the HTLinkOpts.lkp file is populated with auto-generated ROM areas for major code blocks as defined in the user module parameters. Do not modify the code blocks in the HTLinkOpts.lkp file.

- -L-ACODE... & -L-AROM... Lines contain data providing overall ROM size
- -L-PPD_startup... contains linker directives to locate bootloader specific ROM areas
- -L-P
- -L-Pbss0= Changes to any of the last several lines result in an error dialog indicating the inability of the project to detect the correct HTLinkOpts.lkp file.

During code generation, many of the last lines of the HTLinkOpts.lkp file are rewritten under control of the code generation software. Changes made within or below the last three lines either cause an error or are simply lost.

Flashsecurity.example – This is a default file that is laid out according to the default memory map specified by the default user module parameters. For final project creation, you may have to manually modify this file according the final memory map and application size for the deployed device and firmware. Note that this file is not directly used by PSoC Designer. If, for some reason, the project is updated or tagged for out of data files, the flashsecurity file is not overwritten. You can edit and rename the given file flashsecurity.example.

Flashsecurity.txt – This is a default file given by PSoC Designer. The data in this file is added to the .hex file and instructs the device how to manage access to the internal ROM memory. If memory blocks are protected from write access, the bootloader does not work. Since read and write protection is built into the programmed PSoC, this file must be correctly configured before the first deployment of the bootloader.

USB Descriptors

The standard USBFS User Module incorporates a tool to develop the USB descriptor used in the runtime application. The Bootloader adds a tool to allow development or modification of the default USB_Bootloader descriptor. These two descriptors are stored in different areas of ROM. The descriptor for the foreground application may be upgraded with the application. The USB_Bootloader descriptor is part of the bootloader ROM area and cannot be upgraded in the field. To maintain core functionality, key USB code is also placed in the bootloader ROM area. This is to overcome the problem of executing code that is being rewritten (which is not a good programming practice).

USB Compliance for Self Powered Devices

In the *USB Compliance Checklist* there is a question that reads, “Is the device’s pull up active only when V_{BUS} is high?”

The question lists Section 7.1.5 in the *Universal Serial Bus Specification Revision 2.0* as a reference. This section reads, in part, “The voltage source on the pull up resistor must be derived from or controlled by the power supplied on the USB cable such that when V_{BUS} is removed, the pull up resistor does not supply current on the data line to which it is attached.”

If the device that you are creating is self powered, you must connect a GPIO pin to V_{BUS} through a resistive network and write firmware to monitor the status of the GPIO. Application Note AN15813, *Monitoring the EZ-USB FX2LP VBUS*, explains the necessary hardware and software components required. Use the USB IO Control Register 1 (USBIO_CR1) to control the pull up resistor on the D+ line.

Bootloader VID and PID

For final deployment of a USB device, a Vendor ID and Product ID must be assigned. These are assigned by the USB standards organization upon request by USB developers. For development purposes, any VID and PID that does not conflict with existing VIDs and PIDs on a host may be used to debug a project. However, for the purposes of project release or deployment, you must not use VIDs and PIDs assigned to Cypress.

Block Entry of Parameters

All memory parameters are entered in the bootloader in blocks numbered from 0x00 through 0xFF for 16K devices; 0x00 through 0x1FF for 32K devices. Although this is not the most convenient format to enter memory addressees, it prevents accidental assignment of partial block addresses to different sections of the memory map. The PSoC devices in question are only capable of storing 64-byte flash blocks (128 bytes for the 20x45) and this is a simple way to maintain the boundaries between different sections of the project code correctly.

Host Application Debugging

An application with a built-in bootloader may be difficult to debug. Therefore, you can make additional adjustments within the Bootloader User Module files. These are contained in the file BootLdrUSBFS_Bt_loader.inc. There is a section containing the following equates:

```
BOOT_TIMEOUT:           EQU    40   ;set to zero to make timeout infinite
CHECKSUM_ON_CKSUMBLK:   EQU     1   ;Apply a checksum to the checksum block
                                   ;(adds compile steps and code to verify)
```

The BOOT_TIMEOUT equate allows you to lengthen, shorten, or make infinite code that timeouts if no communication is received from a host after a user command calls the bootloader. This may be useful when developing or debugging the host application.

The second equate controls the use of the checksum inside the checksum block. If this equate is set to 0, no verification is done on the checksum contained inside the checksum block. A checksum verification is still performed on the entire user application area as defined in the user module parameters.

Timing

The USBFS User Module supports USB 2.0 full speed operation on the CY8C24x94, CY8CLED04, CY7C64215, CY8C20xx6, CY7C643xx, CYONS2000, CYONS2100, and CYONS2110 devices.

USBFS Setup Wizard

The USBFS Bootloader User Module does not use the PSoC Designer parameter grid display for personalization. Instead, it uses a form driven USBFS Setup Wizard to define the USB descriptors for the application. From the descriptors, the wizard personalizes the user module.

The user module is driven by information generated by the USBFS Setup Wizard. This wizard facilitates the construction of the USB descriptors and integrates the information generated into the driver firmware used for device enumeration. The USBFS Bootloader User Module does not function without first running the wizard, selecting the appropriate attributes, and generating code.

Using Blocks Instead of Addresses

Most PSoC parts have 64-byte block sizes. Some PSoC devices with 128-byte blocks are now being introduced. Two important facts are: any bootloader needs to write to flash and PSoC can only write to flash “block” by “block”. Therefore, for bootloader applications it is more useful to think of memory as a group of “blocks” to be written.

To translate from blocks to absolute addresses multiply: Abs_addr = block_number X Block Size. Block_0 starts at addr 0, Block_n starts at address n x Block_size. All blocks are delimited in hex for the bootloader parameters so a hex address can be obtained by multiplying by 0x40 (64-byte blocks) or 0x80 (128-byte blocks).

Hex output files contain an absolute address for each line. Regardless of the block size of the device in question (0x40/0x80), the hex output file breaks the code into lines of 64(d)/0x40 bytes per line. Therefore, for a 64-byte block device each line represents a block of code. For a 128-byte block device, two lines from the hex file go into a block (since block 0 starts at address 0, 128-byte blocks must be always considered to have an “even” half representing the lower (address) half and an “odd” half representing the upper (address) half).

See a hex file and become familiar with the flash block size for the part that you are working with.

Common Problems

This section discusses the common issues that occur when creating bootloader projects, and gives advice on how to work around them.

Updating Bootloader Projects, Service Pack Upgrades, and Compilers

Changes to the PSoC Developer environment should be avoided when using a bootloader application. This includes not updating PSoC Designer, the Bootloader User Module, and the compiler.

To understand the reasons for this, keep in mind that initially the bootloader and application are compiled together, but after a bootloadable system is deployed, only the application section is reprogrammed. A new or revised application must be compiled with the identical version of the Bootloader User Module so that the new application matches the bootloader from the original deployment. Ideally, all versions of the elements in the development environment are compatible. However, in the case of a bootloader, it is essential to maintain compatibility. By not changing the development environment compatibility risks can be eliminated.

The USB based bootloader exposes its USB subsystem to the application as APIs. This is done to reduce code size. Exposure of these functions is not done through a redirected call table. The implication of this strategy is that the application makes direct calls to specific addresses within the bootloader. Because the bootloader and application are compiled together, any change to the bootloader that results in a change to addresses of the USB API functions results in incompatibility with any other version of the bootloader.

Although multiple compilers are supported by PSoC Designer, do not assume that a bootloader compiled under one compiler is compatible with an application compiled under another. One critical difference regards assumptions about RAM allocation. The implementation of RAM paging may be different from one compiler to another. An added difficulty is that because a bootloader and application are compiled together, it is not possible to debug a bootloader/application pair that had mismatches in the development tools used.

Internal Use of the Watchdog Timer

Coordination with the watchdog timer is linked to the global parameter WATCHDOG_ENABLE, contained in the file globalparams.inc. If your project uses a watchdog timer, it conditionally compiles code linked to the global parameter, and automatically sets the watchdog during bootload checksum and download operations. CPU clock speed affects how fast the watchdog timer is updated. A practical minimum setting for the watchdog timer is approximately 0.125 seconds.

Improper Settings in Flashsecurity.txt

The default settings for this file are set when the project is created. An example configuration is given in the file “Flashsecurity.example”. Flashsecurity.example is given with the BootLoader Tools - Get Files user module menu item. The map must allow flash write at all the locations that are eventually bootloaded. One strategy is to make all blocks writeable. Another strategy is to take a moment to lay out your memory map now and edit this file accordingly. Regardless of the strategy you choose, taking action at the beginning of the project is quicker than debugging it later. It is your responsibility to write protect the areas of code used by the bootloader executable. Failure to correctly map flash security can be a contributing factor in a broken system and an extremely difficult debug task.

For development and debugging purposes, a flash security of 'U' (unprotected) is recommended for the application area. For final production, a flash security setting of 'R' (read protected) is recommended on the application area to prevent external reads and writes from occurring.

Incorrect Relocatable Code Start Address (Linker Parameter ImageCraft Compiler Only)

Since the memory map for a bootloader project is considerably different than that for a conventional project, the relocatable code start address usually needs to be altered. This is a common source of the errors generated by the linker when it attempts to write more than one block code to the same address. This parameter can be changed in the Relocate code start address filed in the Project > Settings > Linker tab. Calculate the absolute hex start address to be a little bigger than the highest block used by the bootloader code, or to occupy an unused area of ROM. For the USB version of the bootloader, set this value to the value obtained by multiplying the parameter:

ApplicationCode_Start_Block X block size = Relocatable Code Start Address.

You can choose to make this value larger than the noted calculation if necessary. Remember that some devices have a block size of 0x80 bytes and some have a block size of 0x40 bytes.

Memory Overlap

To correct the relocatable code start address (refer to the previous section), use a leading semicolon to comment out the last three lines of the custom.lkp file, attempt to build the file again, and examine the resulting memory map. Memory overlap problems are difficult to diagnose because they prevent output files from being generated. Modifying the custom.lkp file may allow the linker to place object blocks, which gives a starting point for correcting the memory overlap root cause.

Power Stability

Power noise, glitches, brownout, slow power ramp, and poor connections can cause difficult to diagnose problems with flash programming. Program execution is rapid in comparison to power ramps, and in some cases, a part may still have changing power levels when flash programming is taking place. One example is some sort of status write to flash at power up. Evaluate your use model and the potential for changing power supply conditions during flash operations. Poor power stability may contribute to nonfunctional parts and may cause poor flash retention.

Application or Interrupts Not Completely Stopped During Bootload Process

The application that is to be replaced by a new bootloaded application must be completely terminated before a bootload operation can take place. It is especially important to turn application interrupts off. When the bootload process takes place, interrupt vector addresses are changed to zero before they are rewritten to their new address. Running interrupts causes random resets (through a vector to 0) if they are not disabled. Note that this does not apply to the specific communication interrupts used by the bootloader.

There are two USB interfaces: one is used by the application and the other is used by the bootloader. A method to explicitly shut down the USB application interface and turn on the bootloader interface should be implemented, which also includes a complete shutdown of the running application. The example code given later in this datasheet contains an example of this procedure.

Downloading a New File Causes the Device to Stop Working

It is possible to construct applications with no facility to enter the bootloader utility. It is easy to do this unintentionally. For example, a main() function with a simple while(1); loop never returns and enters the bootloader. As a result, it cannot be reprogrammed after it begins executing (as long as it has a correct checksum). There are multiple strategies to address this problem, but no default method is included in this user module. A few suggestions are:

1. Apply a reset condition that allows a period of time when the bootloader is enabled when the device first powers up. By setting timeout parameters, the device can be configured to enter the bootloader upon reset and exit to the foreground application when the timeout expires.
2. Set a test at some point in the code that causes the device to enter the bootloader. This can be a switch closure or holding a port pin low/high.
3. Using built-in USB capabilities such as feature reports or a spare endpoint, define USB communication that can be sent to the device to instruct it to enter bootload mode. When this command is sent, the device drops off the USB bus briefly, and when it returns it should enumerate as a bootloader.
4. Use the watchdog timer to reset the device if it is not serviced regularly. This can be combined with one of the above strategies to allow a WDT interrupt to initiate a bootloadable state. Upon reset from a watchdog timer, monitor a status bit associated with the watchdog timer to determine if this is the cause of the reset condition. See the Technical Reference Manual for additional information.
5. Two projects have been developed and the bootloader in each project is different in some subtle way. Keep in mind that “bootloading" implies that programming part of a device is taking place. This implies that the implementation of the bootloader for each of two mutually reprogrammable applications must be identical. All bootloader parameters and relocatable code start addresses should be identical (this is different from first application block). Debug strategies for this problem include comparison of the two hex files in question paying particular attention to the areas of hex code used by the bootloader. Another method is to compare the <project>.lst files. The bootloader makes use of some redirect vectors to allow certain application address parameters to change. All of these jump vectors must match for an application and a bootloader. After a bootloader is deployed to a field application, there is no way to alter the code within it. A future application must still ‘agree’ about where mutually used jump vectors are stored.

Parameters and Resources

The USBFS Bootloader consists of a bootloader utility integrated with a fully functional USBFS User Module.

Parameters defined for the bootloader enable you to define where the major program areas are located when the program is compiled and linked.

Renaming User Modules

Renaming the user module may require specific action on your part, because this user module requires a wizard to fill out and/or overwrite source files. Wizard generated variable names inside wizard generated files may not be updated in all cases. You must open each wizard and select "OK" or "APPLY" to force the regeneration of internal variable names. If compile errors still occur because variable names are not updated, remove the problem file from the project, reopen the wizard, select "OK" or "APPLY", and rebuild the project. The file is replaced with the corrected variable names.

Default Parameters

Default parameters are for information purposes only. Defaults in your project may be tailored to the block size of the part in use, or may have been adjusted to give adequate sizes of code areas. After a project compiles and has been tested a developer may choose to adjust block sizes to optimize memory use.

Figure 3. Default Parameters for a Device with 0x80 (128 Byte) Blocks, for HID Selected Option

ONE_Block_Relocatable_Interrupt_Table	0x2D
ApplicationCode_Start_Block	0x2F
Last_Application_Block	0xFE
Application_Checksum_Block	0xFF
Bootloader_Start_Block	0x2
Bootloader_Size	0x2A
BootLoaderKey	0001020304050607
Flash_Program_Temperature_deg_C	-20C
ICE_Debug_FLASH_DISABLE	DISABLE_FLASH_WRITE
BootLdrUSBFS_ver	0x1000
Bootload_when_CKSUM_fails	ENABLE_(deployment)

Figure 4. Default Parameters for a Device with 0x40 (64 Byte) Blocks, for HID Selected Option

TWO_Block_Relocatable_Interrupt_Table	0x54
ApplicationCode_Start_Block	0x57
Last_Application_Block	0xFE
Application_Checksum_Block	0xFF
Bootloader_Start_Block	0x2
Bootloader_Size	0x51
BootLoaderKey	0001020304050607
Flash_Program_Temperature_deg_C	-20C
ICE_Debug_FLASH_DISABLE	DISABLE_FLASH_WRITE
BootLdrUSBFS_ver	0x1000
Bootload_when_CKSUM_fails	ENABLE_(deployment)

TWO_Block Relocatable_Interrupt_Table

ONE_Block Relocatable_Interrupt_Table

Most PSoC parts have 64 byte block sizes. On these parts, two blocks of the ROM area are used to define a set of interrupt tables that match the tables that are always present in blocks 0 and 1 of PSoC ROM. Some PSoC devices have 128 byte blocks and require only one ROM block to hold the tables. These tables must remain located at the same point in memory, because the block 0 and 1 interrupt tables are redirected here. These relocatable tables point to relocatable interrupt vectors for the application program.

This parameter is also used by the Bootloader Tools to determine what blocks of code to process for a .dld file and what blocks of code to calculate checksums on. This variable is propagated into the checksum block for use when the bootloader utility automatically verifies the application checksum.

ApplicationCode_Start_Block

This is the first block of code assigned to the user application. This code is bootloadable. This parameter is also used by the Bootloader Tools to determine what blocks of code to process for a .dld file and what blocks of code to calculate checksums on. This variable is propagated into the checksum block for use when the bootloader utility automatically verifies the application checksum.

Last_Application_Block

This is the last block of code assigned to the user application. This parameter is also used by the bootloader tools to determine what blocks of code to process for a <prj_name>.dld file and what blocks of code to calculate checksums on. This variable is propagated into the checksum block (after conversion to an absolute address) for use when the bootloader utility automatically verifies the application

checksum. After the application is completed, this value may be reduced to save processing time during the checksum operation and avoid bootloading empty blocks of ROM. If an upgraded application grows, this value may be increased to accommodate the larger code space requirement up to the limit of the available ROM space.

Application_Checksum_Block

This is the location of the Checksum storage block. This block may be rewritten if the application changes but its location must not be moved without redeploying the bootloader.

This block must not be placed within the application area, because doing so would require linking the application "around" the checksum block, which is beyond the capability of the current linker. For this reason, the block must be placed away from the application where it is not "in the way". Possible locations are: a single block between the bootloader and the application or at one of the last blocks of ROM.

Bootloader_Start_Block

This is the first block of the bootloader utility. Ordinarily, you will never have to alter this location. In certain cases though, moving it may be necessary to maintain compatibility with a currently implemented custom bootloader.

Bootloader_Size

This is the size in blocks for the bootloader application. Normally this size does not require adjustment. If you want to add code to the bootloader, you could increase this size to accommodate the extra code. Other blocks would have to be similarly adjusted.

BootLoaderKey

This is the key value prepended to the transactions sent to the bootloader application. The key represents an extra verification step to ensure the bootloader upgrade utility is not started accidentally.

The default value "0001020304050607".

Flash_Program_Temperature_deg_C

This is the typical programming temperature expected when the device is reprogrammed. Programming the device at a different temperature than that specified in this parameter may adversely effect program retention.

Matching the program temperature parameter to the actual temperature during bootload impacts memory retention and maximum number of write cycles. PSoC implements a stronger flash write at colder temperatures. Bootloading at significantly lower temperatures than the parameter setting may reduce memory retention. For this reason, take precautions to ensure that the bootloader is never operated more than 20C from the value in this parameter. Refer to the Cypress device specification for more information.

ICE_Debug_Flash_DISABLE

This parameter is used to overcome an anomaly in the debug behavior of the ICE when executing an SSC while the USB resource is turned on and operating. Whenever an SSC operation is called (and it is during a flash write), the USB SIE is disabled. Disabling flash write allows an application to be completely tested without actually writing code to flash.

The default value is "Flash Write DISABLE".

BootLdrUSBFS_ver

This is the version of the bootloader. It is currently unused by internal firmware, but is available as part of the Checksum block. You can set this value use it to verify the correct version of bootloader executable code.

Bootload_when_CKSUM_fails

Normally if the application checksum is incorrect at reset time the bootloader becomes active and waits for a new application to be bootloaded. During code development with an ICE or debugger it may be inconvenient to take the extra step of correcting the checksum after each compile. This feature allows you to run and debug an application without considering bootloader operations. The checksum feature should be re-enabled for application field deployment.

Application Programming Interface

This section discusses the APIs for the bootloader and the USB functionality contained within the bootloader. The Bootloader contains a very limited set of APIs, because the main purpose of the Bootloader is to completely remove and replace the user application.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns BootLdrUSBFS_1 to the first instance of this user module in a given project. It can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of every global function name, variable and constant symbol. In the following descriptions the instance name is shortened to BootLdrUSBFS for simplicity. Some of the APIs do not have the instance name prepended. These routines, such as *ENTER_BOOTLOADER*, are invariant.

Note For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

Bootloader APIs

ENTER_BOOTLOADER()

GenericBootloaderEntry

Description:

Enters the bootloader application and returns after timeout (if a timeout is defined) if no bootloader host begins to talk to the device. A generic parameter is defined that resides at a fixed address for the life of the deployed part. This function could also be implemented by a direct call to the known hex address of this function.

This function executes a ljmp to the GenericBootloaderEntry and resides at 0x7C.

C Prototype:

```
void ENTER_BOOTLOADER(void)
```

Assembly:

```
lcall  ENTER_BOOTLOADER ; Call the Start Function
```

Alternately:

```
GenericHardDefinition: equ (0x7C)
lcall  GenericHardDefinition            ; Call the Start Function
```

Parameters:

None

GenericApplicationStart

Description:

Enters the application at the beginning of boot.asm. Similar to a warm boot.

C Prototype:

```
void GenericApplicationStart(void)
```

Assembly:

```
lcall  GenericApplicationStart             ; Call the Start Function
```

Parameters:

None

bootLoaderVerify

Description:

Performs a checksum verification of the application storage area. If checksum verification fails, the bootloader is entered and this function never returns. Otherwise the foreground application is executed starting with boot.asm.

C Prototype:

```
void bootLoaderVerify(void)
```

Assembly:

```
lcall bootLoaderVerify             ; Call the Start Function
```

Parameters:

None.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBFS APIs

The application programming interface (API) routines in this section allow programmatic control of the USBFS User Module. The following sections describe descriptor generation and integration, and list the basic and device specific API functions. To develop applications, you need a basic understanding of the USB protocol and familiarity with the USB 2.0 specification, especially Chapter 9, USB Device Framework.

The USBFS User Module supports control, interrupt, bulk, and isochronous transfers. Some functions, or groups of functions, such as LoadInEP and EnableOutEP, are designed for use with bulk and interrupt endpoints. Other functions, such as BootLdrUSBFS_LoadINISOCEP, are designed for use with Isochronous endpoints. Refer to the Technical Reference Manual (TRM) for more information on how to do these transfer types.

Note

1. ** In all user module APIs, the values of the A and X registers may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This “registers are volatile" policy was selected for efficiency reasons. The C compiler automatically takes care of this requirement. Assembly language programmers must ensure their code observes the policy, too. Though some user module API function may leave A and X unchanged, there is no guarantee they may do so in the future.
2. The API routines for the USB user modules are not re-entrant. Because they depend on internal global variables in RAM, executing these routines from an interrupt is not supported by the API supplied with this user module. If this is a requirement for a design, contact the local Cypress Field Application Engineer.

Function	Description
void BootLdrUSBFS_Start(BYTE bDevice, BYTE bMode)	Activate the user module for use with the device and specific voltage mode.
void BootLdrUSBFS_Stop(void)	Disable user module.
BYTE BootLdrUSBFS_bCheckActivity(void)	Checks and clears the USB bus activity flag. Returns 1 if the USB was active since the last check, otherwise returns 0.
void BootLdrUSBFS_SetPowerStatus(BYTE bPowerStatus)	Sets the device to self powered or bus powered
BYTE BootLdrUSBFS_bGetConfiguration(void)	Returns the currently assigned configuration. Returns 0 if the device is not configured.
BYTE BootLdrUSBFS_bGetEPState(BYTE bEPNumber)	Returns the current state of the specified USBFS endpoint. 2 = NO_EVENT_ALLOWED 1 = EVENT PENDING 0 = NO_EVENT_PENDING
BYTE BootLdrUSBFS_bGetEPAckState(BYTE bEPNumber)	Identifies whether ACK was set by returning a nonzero value.
BYTE BootLdrUSBFS_wGetEPCount(BYTE bEPNumber)	Returns the current byte count from the specified USBFS endpoint.
void BootLdrUSBFS_LoadInEP(BYTE bEPNumber, BYTE *pData, WORD wLength, BYTE bToggle)	Loads and enables the specified USBFS endpoint for an IN transfer.
void USB_LoadInISOCEP(BYTE bEPNumber, BYTE *pData, WORD wLength, BYTE bToggle)	
BYTE BootLdrUSBFS_bReadOutEP(BYTE bEPNumber, BYTE *pData, WORD wLength)	Reads the specified number of bytes from the endpoint RAM and places it in the RAM array pointed to by pSrc. The function returns the number of bytes sent by the host.

Function	Description
void USB_EnableOutEP(BYTE bEPNumber)	Enables the specified USB endpoint to accept OUT transfers.
void USB_EnableOutISOCEP(BYTE bEPNumber)	
void BootLdrUSBFS_DisableOutEP(BYTE bEPNumber)	Disables the specified USB endpoint to NAK OUT transfers.

BootLdrUSBFS_Force(BYTE bState)	Forces a J, K, or SE0 State on the USB D+/D- pins. Normally used for remote wakeup.
	bState Parameters are:
	`USB_FORCE_SE0  0xC0`
	`USB_FORCE_J    0xA0`
	`USB_FORCE_K    0x80`
	`USB_FORCE_NONE 0x00`
	
	Note. When using this API Function and GPIO pins from Port 1 (P1.2-P1.7), the application uses the Port_1_Data_SHADE shadow register to ensure consistent data handling. From assembly language, access the Port_1_Data_SHADE RAM location directly. From C language, include an extern reference:
	`extern BYTE Port_1_Data_SHADE;`

Function	Description
BYTE BootLdrUSBFS_UpdateHIDTimer(BYTE bInterface)	Updates the HID Report timer for the specified interface and returns 1 if the timer expired and 0 if not. If the timer expired, it reloads the timer.
BYTE BootLdrUSBFS_bGetProtocol(BYTE bInterface)	Returns the protocol for the specified interface.

BootLdrUSBFS _Start (user defined application device)

BootLdrUSBFS _Start (bootloader device) 0xFF

Description:

Performs all required initialization for USBFS User Module. Either the foreground USB device or the bootloader specific USB device may be started using this command. Only one USB device configuration may be active at any time. To start the bootloader device set the value of bDevice to -1 (0xFF).

C Prototype:

```
void BootLdrUSBFS_Start(BYTE bDevice, BYTE bMode)
```

Assembly:

```
mov   A, 0xFF                ; The bootloader device descriptor
mov   A, 0                   ; Select application device descriptor
mov   X, USB_5V_OPERATION    ; Select the Voltage level
lcall  BootLdrUSBFS_Start     ; Call the Start Function
```

Parameters:

Register A: Contains the device number from the desired device descriptor set entered with the USBFS Setup Wizard.

Register X: Contains the operating voltage at which the chip runs. This determines whether the voltage regulator is enabled for 5V operation or the if pass through mode is used for 3.3V operation. Symbolic names are given in C and assembly, and their associated values are listed in the following table:

Mask	Value	Description
USB_3V_OPERATION	0x02	Disable voltage regulator and pass-through vcc for pull up
USB_5V_OPERATION	0x03	Enable voltage regulator and use regulator for pull up

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

BootLdrUSBFS _Stop

Description:

Performs all necessary shutdown task required for the USBFS User Module.

C Prototype:

```
void  BootLdrUSBFS_Stop(void)
```

Assembly:

```
lcall BootLdrUSBFS_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. Currently only the CUR_PP page pointer register is modified.

BootLdrUSBFS _bCheckActivity

Description:

Checks for USBFS Bus Activity.

C Prototype:

```
BYTE BootLdrUSBFS_bCheckActivity(void)
```

Assembly:

```
lcall BootLdrUSBFS_bCheckActivity
```

Parameters:

None

Return Value:

Returns 1 in A if the USB was active since the last check, otherwise returns 0.

Side Effects:

The A and X registers may be modified by this or future implementations of this function.

BootLdrUSBFS _bGetConfiguration

Description:

Gets the current configuration of the USB device.

C Prototype:

```
BYTE BootLdrUSBFS_bGetConfiguration(void)
```

Assembly:

```
lcall BootLdrUSBFS_bGetConfiguration
```

Parameters:

None

Return Value:

Returns the currently assigned configuration in A. Returns 0 if the device is not configured.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions. Currently only the CUR_PP page pointer register is modified.

BootLdrUSBFS _bGetEPState

Description:

Gets the endpoint state for the specified endpoint. The endpoint state describes, from the perspective of the foreground application, the endpoint status. The endpoint has one of three states, two of the states mean different things for IN and OUT endpoints. The following table outlines the possible states and their meaning for IN and OUT endpoints.

C Prototype:

```
BYTE BootLdrUSBFS_bGetEPState(BYTE bEPNumber)
```

Assembly:

```
MOV A, 1                            ; Select endpoint 1
lcall BootLdrUSBFS_bGetEPState
```

Parameters:

Register A contains the endpoint number.

Return Value:

Returns the current state of the specified USBFS endpoint. Symbolic names given in C and assembly, and their associated values are listed in the following table. Use these constants whenever you write code to change the state of the endpoints such as ISR code to handle data sent or received.

State	Value	Description
NO_EVENT_PENDING	0x00	Indicates that the endpoint is awaiting SIE action.
EVENT_PENDING	0x01	Indicates that the endpoint is awaiting CPU action.
NO_EVENT_ALLOWED	0x02	Indicates that the endpoint is locked from access.
IN_BUFFER_FULL	0x00	The IN endpoint is loaded and the mode is set to ACK IN.
IN_BUFFER_EMPTY	0x01	An IN transaction occurred and more data can be loaded.
OUT_BUFFER_EMPTY	0x00	The OUT endpoint is set to ACK OUT and is waiting for data.
OUT_BUFFER_FULL	0x01	An OUT transaction has occurred and data can be read.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP page pointer register is modified.

BootLdrUSBFS _bGetEPAckState

Description:

Determines whether or not an ACK transaction occurred on this endpoint by reading the ACK bit in the control register of the endpoint. This function does not clear the ACK bit.

C Prototype:

```
BYTE BootLdrUSBFS_bGetEPState(BYTE bEPNumber)
```

Assembly:

```
MOV A, 1                            ; Select endpoint 1
lcall BootLdrUSBFS_bGetEPState
```

Parameters:

Register A contains the endpoint number.

Return Value:

If an ACKed transaction occurred then this function returns a non-zero value. Otherwise, a zero is returned.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions.

BootLdrUSBFS _wGetEPCount

Description:

This functions returns the value of the endpoint count register. The Serial Interface Engine (SIE) includes two bytes of checksum data in the count. This function subtracts two from the count before returning the value. Call this function only for OUT endpoints after a call to USB_GetEPState returns EVENT_PENDING.

C Prototype:

```
WORD BootLdrUSBFS_wGetEPCount(BYTEbEPNumber)
```

Assembly:

```
MOV A, 1                            ; Select endpoint 1
lcall BootLdrUSBFS_bGetEPCount
```

Parameters:

Register A contains the endpoint number.

Return Value:

Returns the current byte count from the specified USBFS endpoint in A and X.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions.

BootLdrUSBFS _LoadInEP

BootLdrUSBFS _LoadInISOCEP

Description:

Loads and enables the specified USB endpoint for an IN interrupt or bulk transfer (.._LoadInEP) and isochronous transfer (..._LoadInISOCEP).

C Prototype:

```
void BootLdrUSBFS_LoadInEP(BYTE bEPNumber, BYTE * pData, WORD wLength, BYTE bToggle)
void BootLdrUSBFS_LoadInISOCEP(BYTE bEPNumber, BYTE * pData, WORD wLength, BYTE
bToggle)
```

Assembly:

```
mov A, USBFS_TOGGLE
push A
mov A, 0
```

```
push A
mov A, 32
push A
mov A, >pData
push A
mov A, <pData
push A
mov A, 1
push A
lcall BootLdrUSBFS_LoadInEP
```

Parameters:

bEPNumber – The endpoint Number between one and four.

pData – A pointer to a data array. Data for the endpoint is loaded from the data array specified by pData.

wLength – The number of bytes to transfer from the array as a result of an IN request. Valid values are between 0 and 256.

bToggle – A flag indicating whether or not the Data Toggle bit is toggled before setting it in the count register. For IN transactions toggle the data bit after every successful data transmission. This makes certain that the same packet is not repeated or lost. Symbolic names for the flag are given in C and assembly:

Mask	Value	Description
USB_NO_TOGGLE	0x00	The Data Toggle does not change
USB_TOGGLE	0x01	The Data bit is toggled before transmission

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

BootLdrUSBFS _bReadOutEP

Description:

Moves the specified number of bytes from endpoint RAM to data RAM. The number of bytes actually transferred from endpoint RAM to data RAM is the lesser of the actual number of bytes sent by the host and the number of bytes requested by the wCount argument.

C Prototype:

```
BYTE BootLdrUSBFS_bReadOutEP(BYTE bEPNumber, BYTE * pData, WORD wLength)
```

Assembly:

```
mov A, 0
```

```
push A
mov A, 32
push A
mov A, >pData
push A
mov A, <pData
push A
mov A, 1
push A
lcall BootLdrUSBFS_bReadOutEP
```

Parameters:

bEPNumber – The endpoint Number between one and four.

pData – The endpoint space is loaded from a data array specified by this pointer.

wLength – The number of bytes to transfer from the array and then send as a result of an IN request. Valid values are between 0 and 256. The function moves less than that if the number of bytes sent by the host are less requested.

Return Value:

Returns the number of bytes sent by the host to the USB device. This could be more or less than the number of bytes requested.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

BootLdrUSBFS _EnableOutEP

USBFS_EnableOutISOCEP

Description:

Enables the specified endpoint for OUT Bulk or Interrupt transfers (..._EnableOutEP) and Isochronous transfers (..._EnableOutISOCEP). Do not call these functions for IN endpoints.

C Prototype:

```
void USBFS_EnableOutEP(BYTE bEPNumber)
void USBFS_EnableOutISOCEP(BYTE bEPNumber)
```

Assembly:

```
MOV A, 1
lcall USBFS_EnableOutEP
```

Parameters:

Register A contains the endpoint number.

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. Currently only the IDX_PP page pointer register is modified.

BootLdrUSBFS _DisableOutEP

Description:

Disables the specified USBFS OUT endpoint. Do not call this function for IN endpoints.

C Prototype:

```
void BootLdrUSBFS_DisableEP(BYTE bEPNumber)
```

Assembly:

```
MOV A, 1                            ; Select endpoint 1
lcall BootLdrUSBFS_DisableEP
```

Parameters:

Register A contains the endpoint number.

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions.

BootLdrUSBFS _Force

Description:

Forces a USB J, K, or SE0 state on the D+/D- lines. This function gives the necessary mechanism for a USB device application to perform USB remote wakeup functionality. For more information, refer to the USB 2.0 Specification for details on Suspend and Resume functionality.

C Prototype:

```
void BootLdrUSBFS_Force(BYTE bState)
```

Assembly:

```
mov A, BootLdrUSB_FORCE_K
lcall BootLdrUSBFS_Force
```

Parameters:

bState is byte indicating which among four bus states to enable. Symbolic names are given in C and assembly code, and their associated values are listed in the following table:

State	Value	Description
USB_FORCE_SE0	0xC0	Force a Single Ended 0 onto the D+/D- lines
USB_FORCE_J	0xA0	Force a J State onto the D+/D- lines
USB_FORCE_K	0x80	Force a K State onto the D+/D- lines
USB_FORCE_NONE	0x00	Return bus to SIE control

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions.

BootLdrUSBFS _UpdateHIDTimer

Description:

Updates the HID Report Idle timer and returns the expiry status. Reloads the timer if it expires.

C Prototype:

```
BYTE BootLdrUSBFS_UpdateHIDTimer(BYTE bInterface)
```

Assembly:

```
MOV A, 1                            ; Select interface 1
lcall BootLdrUSBFS_UpdateHIDTimer
```

Parameters:

Register A contains the interface number.

Return Value:

The state of the HID timer is returned in A. Symbolic names are given in C and assembly code, and their associated values are listed in the following table:

State	Value	Description
USB_IDLE_TIMER_EXPIRED	0x01	The timer expired.
USB_IDLE_TIMER_RUNNING	0x02	The timer is running.
USB_IDLE_TIMER_IDEFINITE	0x00	Returned if the report is sent when data or state changes.

Side Effects:

The A and X registers may be modified by this or future implementations of this function.

BootLdrUSBFS _bGetProtocol

Description:

Returns the HID protocol value for the selected interface.

C Prototype:

```
BYTE BootLdrUSBFS_bGetProtocol(BYTE bInterface)
```

Assembly:

```
MOV A, 1                            ; Select interface 1
lcall BootLdrUSBFS_bGetProtocol
```

Parameters:

bInterface contains the interface number.

Return Value:

Register A contains the protocol value.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the large memory model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions.

BootLdrUSBFS_SetPowerStatus

Description:

Sets the current power status. Set the power status to one for self powered or zero for bus powered. The device will reply to USB GET_STATUS requests based on this value. This allows the device to properly report its status for USB Chapter 9 compliance. Devices may change their power source from self powered to bus powered at any time and report their current power source as part of the device status. Call this function any time your device changes from self powered to bus powered or vice versa, and set the status appropriately.

C Prototype:

```
void BootLdrUSBFS_SetPowerStatus(BYTE bPowerStaus);
```

Assembly:

```
MOV   A, 1                          ; Select self powered
lcall  BootLdrUSBFS_SetPowerStatus
```

Parameters:

bPowerStatus contains the desired power status, one for self powered or zero for bus powered. Symbolic names are given in C and assembly code, and their associated values are listed in the following table:

State	Value	Description
USB_DEVICE_STATUS_BUS_POWERED	0x00	Set the device to bus powered.
USB_DEVICE_STATUS_SELF_POWERED	0x01	Set the device to self powered.

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the Large Memory Model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions.

Sample Firmware Source Code

For both the C and assembly language example projects, leave the Bootloader User Module parameters in their default state (see Figure 3-4).

Ensure the correct pin is configured as a pull down to recognize the switch closure to enter the bootloader. The example depends on a button that you can press. This button pulls Port1_7 high and causes the program to re-enumerate as a bootloader. On some devices, the setting may be "pull down", and on other devices the correct setting may be "open-drain-low". You may need to consider the exact configuration of the test hardware used as a target for this example, and consult the relevant technical reference manual to correctly configure this project.

Port_1_4	P1[4]	StdCPU	High Z Analog	DisableInt
Port_1_5	P1[5]	StdCPU	High Z Analog	DisableInt
Port_1_6	P1[6]	StdCPU	High Z Analog	DisableInt
Port_1_7	P1[7]	StdCPU	Pull Down	DisableInt
Port_2_0	P2[0]	StdCPU	High Z Analog	DisableInt
Port_2_1	P2[1]	StdCPU	High Z Analog	DisableInt

Before you attempt to bootload an application, ensure that you modify the *flashsecurity.txt* file to make the application, checksum, and relocatable interrupt vector areas writeable. The example flashsecurity.txt file shown in the following figure is an example. Some devices look slightly different, but all follow the same basic pattern.

```
; to the end of the line.

; 0 40 80 C0 100 140 180 1C0 200 240 280 2C0 300 340 380 3C0 (+) Base Address

  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 0
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 400
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 800
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address C00
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  U  U  U ;    Base Address 1000
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 1400
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 1800
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 1C00
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 2000
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 2400
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 2800
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 2C00
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 3000
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 3400
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 3800
  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U  U ;    Base Address 3C00
; End 16K parts
```

Note that the included Host PC example project has a VID of 04b4 and a PID of E006. These are Cypress owned IDs and may be used for local debug, but cannot be released for production.

Here is an implementation of the USB Bootloader User Module written in C. This sample code was developed for a CY7C64215 device family. Other device families supported by this user module may have different available ports, pins, and drive modes. You may have to customize this code for it to work on your device.

```
//
//emulate a mouse that causes the cursor to move in a square
//

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all user modules

signed char bXInc = 0;  // X-Step Size
signed char bYInc = 0;  // Y-Step Size

#define USB_INIT     0   // Initialized state
#define USB_UNCONFIG 1   // Unconfigured state
#define USB_CONFIG   2   // Configured state

// Mouse movemet states
#define MOUSE_DOWN   0
#define MOUSE_LEFT   1
#define MOUSE_UP     2
#define MOUSE_RIGHT  3

#define POSMASK      0x03 // Mouse position state mask
#define BOX_SIZE     32   // Transfers per side of the box
#define bCursorStep  4    // Step size

BYTE bConfigState = 0;    // Configuration state
BYTE bDirState = 0;       // Mouse diretion state

BYTE abMouseData[3] = {0,0,0};  // Endpoint 1, mouse packet array
BYTE bButton;                   // Used for button
BYTE boxLoop = 0;               // Box loop counter

const char abDirection[4][6] = {"DOWN "};
extern const  USB_pAppChkSumBlk;
WORD blversion;
void main(void)
{
M8C_EnableGInt;                     //Enable Global Interrupts
USB_Start(0, USB_5V_OPERATION);     //Start USB Operation usgin device 0


PRT1DR = 0;

while(1) {                                  // Main loop
   if(PRT1DR & 0x80) {
   USB_Stop();
   while(PRT1DR & 0x80);
   USB_EnterBootloader();
   }
```

```
   switch(bConfigState) {                    // Check state
   case USB_INIT:                            // Initialize state
      bConfigState = USB_UNCONFIG;

      break;

case USB_UNCONFIG:                          // Unconfigured state
      if(USB_bGetConfiguration() != 0) {     // Check if configuration set
         bConfigState = USB_CONFIG;

         USB_LoadInEP(1, abMouseData, 3, USB_NO_TOGGLE);  // Load a dummy mouse packet
      }
      break;

case USB_CONFIG:                             // Configured state time to move the mouse
      if(USB_bGetEPAckState(1) != 0) {
         boxLoop++;
       if(boxLoop > BOX_SIZE) {              // Change mouse direction every 32 packets
            boxLoop = 0;
           bDirState++;    // Advance box state
            bDirState &= POSMASK;

         }

switch(bDirState) {                         // Determine current direction state

case MOUSE_DOWN:                            // Down
            bXInc = 0;
            bYInc = bCursorStep;
            //asm("nop");
            break;

case MOUSE_LEFT:                            // Left
            bXInc = -bCursorStep;
            bYInc = 0;
            break;

case MOUSE_UP:                              // up
            bXInc = 0;
            bYInc = -bCursorStep;
            break;

case MOUSE_RIGHT:                           // Right
            bXInc = bCursorStep;
            bYInc = 0;
            break;

}
         abMouseData[1] = bXInc;                    // Load the packet array
         abMouseData[2] = bYInc;
         abMouseData[0] = 0;   // No buttons pressed
         USB_LoadInEP(1, abMouseData, 3, USB_TOGGLE); // Load and cock Endpoint1


}  // End if Endpoint ready
```

```
      break;

}  // End Switch
}   // End While
}
```

Here is an implementation of the USB Bootloader User Module written in assembly code.

The assembly code illustrated here shows you how to use the BootLdrUSBFS User Module in a simple HID application. When connected to a PC host, the device enumerates as a 3 button mouse. When the code is run the mouse cursor zigzags from right to left. This code illustrates how the BootLdrUSBFS Setup Wizard configures the user module. This project is identical to the project in the USBFS User Module, except with the addition of a bootloader.

```
;-----------------------------------------------------------------------
; Assembly main line
;-----------------------------------------------------------------------

include "m8c.inc" ; part specific constants and macros
include "memory.inc" ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc" ; PSoC API definitions for all user modules



export _main
export i
export abMouseData

area bss(RAM) // inform assembler that variables follow
abMouseData: blk 3 // USBFS data variable
i: blk 1 // count variable

area text(ROM,REL) // inform assembler that program code follows

_main:
OR F,1
; Start USBFS Operation using device 0
PUSH X
MOV X,3
MOV A,0
LCALL BootLdrUSBFS_Start
POP X

; Wait for Device to enumerate
.no_device:
PUSH X
LCALL BootLdrUSBFS_bGetConfiguration
POP X
CMP A,0
JZ .no_device
; Enumeration is completed load endpoint 1. Do not toggle the first time
; BootLdrUSBFS_LoadInEP(1, abMouseData, 3, USB_TOGGLE);
PUSH X
MOV A,1
PUSH A
```

```
MOV A,0
PUSH A
MOV A,3
PUSH A
MOV A,>abMouseData
PUSH A
MOV A,<abMouseData
PUSH A
MOV A,1
PUSH A
LCALL BootLdrUSBFS_LoadInEP
ADD SP,250
POP X

.endless_loop:



;implement bootloader entry
mov reg[PRT1DR], 0   ;load reg[PRT1DR] with  0
;   if(PRT1DR & 0x80) {
;   USB_Stop();
;   while(PRT1DR & 0x80);
;   BootLdrUSBFS_EnterBootloader();
;   }
push A
mov A, reg[PRT1DR]
and A, 0x80
jz .Exit_BOOTLOAD_TEST
;****    IMPORTANT,      configure prt0.7 as a stdcpu/pulldown     IMPORTANT     *****
;if PRT1DR.7 is pulled high, (configure for a pull down, set data to zero)
; wait for the port pin to be released (back to zero) to debounce
; immeciately un-enumerate by releasing the D+ pullup
lcall BootLdrUSBFS_Stop
.wait_for_bit_low:
tst reg[PRT1DR], 0x80
jnz .wait_for_bit_low
; once it goes low enter the bootloader
pop A
ljmp BootLdrUSBFS_EnterBootloader ;;never returns
halt

.Exit_BOOTLOAD_TEST:
pop A


;;; mouse operations



PUSH X
MOV A,1
LCALL BootLdrUSBFS_bGetEPAckState
POP X
CMP A,0
```

```
JZ .endless_loop
; ACK has occurred, load the endpoint and toggle the data bit
; BootLdrUSBFS_LoadInEP(1, abMouseData, 3, USB_TOGGLE);
PUSH X
MOV A,1
PUSH A
MOV A,0
PUSH A
MOV A,3
PUSH A
MOV A,>abMouseData
PUSH A
MOV A,<abMouseData
PUSH A
MOV A,1
PUSH A
LCALL BootLdrUSBFS_LoadInEP
ADD SP,250
POP X

; When our count hits 128
CMP [i],128
JNZ .move_left
; Start moving the mouse to the right
MOV [abMouseData+1],5
JMP .increment_i
; When our counts hits 255
.move_left:
CMP [i],255
JNZ .increment_i
; Start moving the mouse to the left
MOV [abMouseData+1],251

.increment_i:
INC [i]

JMP .endless_loop

.terminate:
jmp .terminate
```

USBFS Setup Corresponding to the Example Code

1. Create a new project with a base part supported by the BootLdrUSBFS User Module (such as CY8C24894).
2. In the Device Editor, click **Protocols**. Add the BootLdrUSBFS User Module by double clicking the BootLdrUSBFS icon, or right clicking and choosing **Select**.
3. Select the Human Interface Device (HID) radio button. Optional step: rename the user module from BootLdrUSBFS_1 to BootLdrUSBFS to match the sample code by right clicking the module and selecting **Rename**.
4. Right click the USBFS User Module icon in the Device Editor to open the “Device: Application USB Setup Wizard".

- Click the Import HID Report Template operation and change the name to Import HID Report Template (italics) to show that it is a label.
- Select the 3 button mouse template.
- Click the Apply operation on the right side of the template.
- Select the Add String operation to add Manufacturer and Product strings.
- Edit the device attributes: Vendor ID, Product ID, and select strings.
- Edit the interface attributes: select HID for the Class field.
- Edit the HID class descriptor: select the 3 button mouse for the HID Report field.

5. Click **OK** to save the USB descriptor information.
6. Right click the USBFS User Module icon in the Device Editor to open the “Device: BootLoader USB Setup Wizard".
7. Enter the correct VID (Vendor ID) and PID (Product ID) into the wizard. Note that the VID and PID for the application and the bootloader cannot be identical.
8. Click **OK** to save the USB BootLoader descriptor information.
9. Generate the Application.
10. Copy the Sample code and paste it in the main.c.
11. Select **Rebuild all**.

Descriptor	Data
USB User Module descriptor root	Device name
Device descriptor	Device
Device attributes	
Vendor ID	Use company VID
Product ID	Use product PID
Device release (bcdDevice)	0000
Device class	Defined in interface descriptor
Subclass	No subclass
Manufacturer string	My company
Product string	My mouse
Serial number string	No string
Configuration descriptor	Configuration
Configuration attributes	
Configuration string	No string
Max power	100
Device power	Bus powered
Remote wakeup	Disabled

Descriptor	Data
Interface descriptor	Interface
Interface attributes	
Interface string	No string
Class	HID
Subclass	No subclass
HID class descriptor	
Descriptor type	Report
Country code	Not supported
HID report	3-button mouse
Endpoint descriptor	ENDPOINT_NAME
Endpoint attributes	
Endpoint number	1
Direction	IN
Transfer type	INT
Interval	10
Max packet size	8
String/LANGID	
String descriptors	USBFS
LANGID	
String	My company
String	My mouse
Descriptor	
HID report descriptor root	USBFS
HID report descriptor	USBFS

Appendix A – USBFS Topics

USB Standard Device Requests

The following section describes the requests supported by the USB User Module. If a request is not supported the USB User Module normally responds with a STALL, indicating a Request Error.

Standard Device Request	USB User Module Support Description	USB 2.0 Spec Section
CLEAR_FEATURE	Device:	9.4.1
	Interface: Not supported.	
	Endpoint	
GET_CONFIGURATION	Returns the current device configuration value.	9.4.2
GET_DESCRIPTOR	Returns the specified descriptor.	9.4.3
GET_INTERFACE	Returns the selected alternate interface setting for the specified interface.	9.4.4
GET_STATUS	Device:	9.4.5
	Interface:	
	Endpoint:	
SET_ADDRESS	Sets the device address for all future device accesses.	9.4.6
SET_CONFIGURATION	Sets the device configuration.	9.4.7
SET_DESCRIPTOR	This optional request is not supported.	9.4.8
SET_FEATURE	Device: DEVICE_REMOTE_WAKEUP support is selected by the bRemoteWakeUp User Module Parameter. TEST_MODE is not supported.	9.4.9
	Interface: Not supported.	
	Endpoint: The specified endpoint is halted.	
SET_INTERFACE	Not supported.	9.4.10
SYNCH_FRAME	Not supported. Future implementations of the user module adds support to this request to enable Isochronous transfers with repeating frame patterns.	9.4.11

HID Class Request

Class Request	USB User Module Support Description	Device Class Definition for HID - Section
GET_REPORT	Allows the host to receive a report by way of the Control pipe.	7.2.1
GET_IDLE	Reads the current idle rate for a particular Input report.	7.2.3
GET_PROTOCOL	Reads which protocol is currently active (either the boot or the report protocol).	7.2.5
SET_REPORT	Allows the host to send a report to the device, possibly setting the state of input, output, or feature controls.	7.2.2
SET_IDLE	Silences a particular report on the Interrupt In pipe until a new event occurs or the specified amount of time passes.	7.2.4
SET_PROTOCOL	Switches between the boot protocol and the report protocol (or vice versa).	7.2.6

USBFS Setup Wizard

This section details all the USBFS descriptors given by the USBFS User Module. The descriptions include the descriptor format and how user module parameters map into the descriptor data.

The USBFS Setup Wizard is a tool given by Cypress to assist engineers in designing USB devices. The setup wizard displays the device descriptor tree; when expanded, the following folders that are part of the standard USB descriptor definitions appear:

- Device attributes
- Configuration descriptor
- Interface descriptor
- HID Class descriptor
- Endpoint descriptor
- String/LANGID
- HID Descriptor

To access the setup wizard, right click the USB User Module icon in the device editor and click the USB Setup Wizard... menu item.

When the device descriptor tree is fully expanded, all the setup wizard options are visible. The left side displays the name of the descriptor, the center displays the data, and the left displays the operation available for a particular descriptor. In some instances, a descriptor has a pull down menu that presents available options.

Descriptor	Data		Operations
USBFS User Module descriptor root	“Device Name”		Add device
Device descriptor	DEVICE_1		Remove|Add configuration
Device attributes			
Vendor ID	FFFF		
Product ID	FFFF		
Device release (bcdDevice)	0000		
Device class	Undefined	pull down	
Subclass	No subclass	pull down	
Protocol	None	pull down	
Manufacturer string	No string	pull down	
Product string	No string	pull down	
Serial number string	No string	pull down	
Configuration descriptor	CONFIG_NAME		Remove|Add interface
Configuration attributes			
Configuration string	No string	pull down	
Max power	100		
Device power	Bus powered	pull down	
Remote wakeup	Disabled	pull down	
Interface descriptor	INTERFACE_NAME		Remove|Add endpoint
Interface attributes			
Interface string	No string	pull down	
Class	Vendor specific	pull down	
Subclass	No subclass	pull down	
HID class Descriptor			
Descriptor type	Report	pull down	
Country code	Not supported	pull down	
HID report	None	pull down	
Endpoint descriptor	ENDPOINT_NAME		Remove
Endpoint attributes			
Endpoint number	0		
Direction	IN	pull down	

Descriptor	Data		Operations
Transfer type	CNTRL	pull down	
Interval	10		
Max packet size	8		
String/LANGID			
String descriptors	Device name		Add string
LANGID		pull down	
String	Selected string name		Remove
Descriptor			
HID Descriptor	Device name		Import HID Report Template

Understanding the USB Setup Wizard

The USB Setup Wizard window is a table that presents three major areas for programming. The first area is the Descriptor USBFS User Module, the second is the String/LANGID, and the third is the Descriptor HID report. Use the two buttons below the table to perform the selected command.

The first section presents the Descriptor. The second section presents the String/LANGID; when a string ID is required, this area is used to input that string. To add a string for a USB device, click on the **Add String** operation. The software adds a row and prompts you to Edit your string here. Type the new string and click **Save/Generate**. After the string is saved, it is available for use in the Descriptor section from the pull down menus. If you close without saving, the string is lost.

The third area presents the HID Report Descriptor Root. From this area, you can add or import a HID Report for the selected device.

USB User Module Descriptor Root

The first column displays folders to expand and collapse. For the purpose of this discussion, you must fully expand the tree that all options are visible. The setup wizard allows you to enter data into the middle Data column; if there is a pull down menu, use it to select a different option. If there is no pull down menu, but there is data, use the cursor to highlight and select the data, then overwrite that data with another value or text option. All the values must meet the USB 2.0 Chapter 9 Specifications.

The first folder displayed at the top is the *USB User Module Descriptor Root*. It has the user module name in the Data column (this is the user module name given to it by the software. This user module is the one placed in the Interconnect View. The Add Device operation on the right hand column adds another USB device complete with all the different fields required for describing it. The new USB device descriptor is listed at the bottom after the endpoint Descriptor. Click **OK** to save. If you do not save the newly added device, it is not available for use.

Device Descriptor has DEVICE_NUMBER as the Data; it may be removed or a configuration added. All the information about a particular USB device may be entered by over writing the existing data or by using a pull down menu.

When the input of data is complete, either by using the pull down menus or by typing alphanumeric text in the appropriate spots, click **OK** to save.

USB Suspend, Resume, and Remote Wakeup

The USBFS User Module supports USB Suspend, Resume, and remote wakeup. Since these features are tightly coupled into the user application, the USBFS User Module gives a set of API functions.

Monitoring USFS Activity

The BootLdrUSBFS_bCheckActivity API function enables you to check if any USB bus activity has occurred. The application uses this function to determine if the conditions to enter USB Suspend are met.

USBFS Suspend

After the conditions to enter USB suspend are met, the application takes appropriate steps to reduce current consumption to meet the suspend current requirements. To put the USB SIE and transceiver into power down mode, the application calls the BootLdrUSBFS_Suspend API and BootLdrUSBFS_bCheckActivity API functions to detect USB activity. This function disables the USBFS block, but maintains the current USB address (in the USBIO_CR0 register). The device uses the sleep feature to reduce power consumption.

USBFS Resume

While the device is suspended, it periodically checks to determine if the conditions to leave the suspended state are met. One way to check resume conditions is to use the sleep timer to periodically wake the device. If the resume conditions are met, the application calls the BootLdrUSBFS_Resume API function. This function enables the USBFS SIE and Transceiver, bringing them out of power down mode. It does not change the USB address field of the USBCR register, maintaining the USB address previously assigned by the host.

USBFS Remote Wakeup

If the device supports remote wakeup, the application is able to determine if the host enabled remote wakeup with the BootLdrUSBFS_bRWUEnabled API function. When the device is suspended and it determines the conditions to initiate a remote wakeup are met, the application uses the BootLdrUSBFS_Force API function to force the appropriate J and K states onto the USB Bus, signaling a remote wakeup.

Creating Vendor Specific Device Requests and Overriding Existing Requests

The USBFS User Module supports vendor specific device requests by providing a dispatch routine for handling setup packet requests. You can also write your own routines that override any of the supplied standard and class specific routines, or enable unsupported request types.

Processing USBFS Device Requests

All control transfers, including vendor specific and overridden device requests, are composed of:

- A setup stage where request information is moved from host to device.
- A data stage consisting of zero or more data transactions with data send in the direction specified in the setup stage.
- A status stage that concludes the transfer.

In the BootLdrUSBFS User Module, all control transfers are handled by the endpoint 0 Interrupt Service Routine (BootLdrUSBFS_EP0_ISR).

The endpoint 0 Interrupt Service Routine transfers control of all setup packets to the dispatch routine, which routes the request to the appropriate handler based upon the bmRequestType field. The handler initializes specific user module data structures and transfers control back to the endpoint 0 ISR. A handler for vendor specific or override device request is given by the application. The user module handles the data and status stages of the transfer without involving your application. After the transfer is completed, the user module updates a completion status block. The status block is monitored by the application to determine if the vendor specific device request is complete.

All setup packets enter the BootLdrUSBFS_EP0_ISR, which routes the setup packet to the BootLdrUSBFS_bmRequestType_Dispatch routine. From here all the standard device requests and vendor specific device requests are dispatched. The device request handlers must prepare the application to receive data for control writes or prepare the data for transmission to the host for control reads. For no-data control transfers, the handler extracts information from the setup packet itself.

The USBFS User Module processes the data and status stages exactly the same way for all requests. For data stages, the data is copied to or from the control endpoint buffer (registers EP0DATA0-EP0DATA7) depending upon the direction of the transaction.

Vendor Specific Device Request Dispatch Routines

Depending upon the application requirements, the USBFS User Module dispatches up to eight types of vendor specific device requests based upon the bmRequestType field of the setup packet. Refer to section 9.3 of the USB 2.0 specification for a discussion of USB device requests and the bmRequestType field. The eight types of vendor specific device requests the USBFS User Module dispatches are listed in the Table 1:

Table 1. Vendor Specific Request Dispatch Routine Names

Direction	Recipient	Dispatch Routine Entry Point	Enable Flag
Host to Device (Control Write)	Device	USB_DT_h2d_vnd_dev_Dispatch	USB_CB_h2d_vnd_dev
	Interface	USB_DT_h2d_vnd_ifc_Dispatch	USB_CB_h2d_vnd_ifc
	Endpoint	USB_DT_h2d_vnd_ep_Dispatch	USB_CB_h2d_vnd_ep
	Other	USB_DT_h2d_vnd_oth_Dispatch	USB_CB_h2d_vnd_oth
Device to Host (Control Read)	Device	USB_DT_d2h_vnd_dev_Dispatch	USB_CB_d2h_vnd_dev
	Interface	USB_DT_d2h_vnd_ifc_Dispatch	USB_CB_d2h_vnd_ifc
	Endpoint	USB_DT_d2h_vnd_ep_Dispatch	USB_CB_d2h_vnd_ep
	Other	USB_DT_d2h_vnd_oth_Dispatch	USB_CB_d2h_vnd_oth

You must follow these steps for an application to give an assembly language dispatch routine for the vendor specific device request.

1. In the *USBFS.inc* file, enable support for the vendor specific dispatch routine. Find the dispatch routine enable flag and set EQU to 1.
2. Write an appropriately named assembly language routine to handle the device request. Use the entry points listed in Table 1.

Override Existing Request Routines

To override a standard or class specific device request, or enable an unsupported device request, you must do the following:

1. In the *USBFS.inc* file, redefine the specific device request as USB_APP_SUPPLIED.
2. Write an appropriately named assembly language function to handle the device request. The name of the assembly language function is APP_ plus the device name.

For example, to override the supplied HID class Set Report request, USB_CB_SRC_h2d_cls_ifc_09, enable the routine with these changes to *USBFS.inc*:

```
;@PSoC_UserCode_BODY_1@ (Do not change this line.)
   ;---------------------------------------------------
   ; Insert your custom code below this banner
   ;---------------------------------------------------
   ;   NOTE: interrupt service routines must preserve
   ;   the values of the A and X CPU registers.

; Enable an override of the HID class Set Report request.
USB_CB_SRC_h2d_cls_ifc_09: EQU USB_APP_SUPPLIED

   ;---------------------------------------------------
   ; Insert your custom code above this banner
   ;---------------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)
```

Then, write an assembly language routine named APP_USB_CB_SRC_h2d_cls_ifc_09. Device request names are derived from the USB bmRequestType and bRequest values (USB specification Table 9-2).

This code is a stub for the assembly routine for the previous example:

```
export APP_USB_CB_SRC_h2d_cls_ifc_09
APP_USB_CB_SRC_h2d_cls_ifc_09:

;Add your code here.

; Long jump to the appropriate return entry point for your application.
LJMP BootLdrUSBFS_InitControlWrite
```

Appendix B – Bootloader Topics

The following section contains additional information that you may find useful when creating a USB bootloader.

Dispatch and Override Routine Requirements

At a minimum, the dispatch or override routine must return control back to the endpoint 0 ISR by a LJMP to one of the endpoint 0 ISR Return Points listed in Table 2. The routine may destroy the A and X registers, but the Stack Pointer (SP) and any other relevant context must be restored before returning control to the ISR.

Table 2. Endpoint 0 ISR Return Points

Return Entry Point	Required Data Items	Description
USB_Not_Supported	Use this return point when the request is not supported. It STALLs the request.	
	Data Items: None	
USB_InitControlRead	This return point is used to initiate a Control Read transfer.	
	BootLdrUSBFS_DataSource (BYTE)	The data source is RAM or ROM (USB_DS_RAM or USB_DS_ROM). This is necessary since different instructions are used to move the data from the source ROMX or MOV.
	BootLdrUSBFS_TransferSize (WORD)	The number of data bytes to transfer.
	BootLdrUSBFS_DataPtr (WORD)	RAM or ROM address of the data.
	BootLdrUSBFS_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USB_XFER_STATUS_BLOCK macro.
BootLdrUSBFS_InitControlWrite	This return point is used to initiate a Control Write transfer.	
	BootLdrUSBFS_DataSource (BYTE)	USB_DS_RAM (the destination for control writes must RAM).
	BootLdrUSBFS_TransferSize (WORD)	Size of the application buffer to receive the data
	BootLdrUSBFS_DataPtr (WORD)	RAM address of the application buffer to receive the data
	BootLdrUSBFS_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USB_XFER_STATUS_BLOCK macro.
USB_InitNoDataControlTransfer	This return point is used to initiate a No Data Control transfer.	
	BootLdrUSBFS_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USB_XFER_STATUS_BLOCK macro.

Status Completion Block

The status completion block contains two data items, a one byte completion status code and a two byte transfer length. The “main" application monitors the completion status to determine how to proceed. Completion status codes are found in the following table. The transfer length is the actual number of data bytes transferred.

Table 3. USBFS Transfer Completion Codes

Completion Code	Description
USB_XFER_IDLE (0x00)	USB_XFER_IDLE indicates that the associated data buffer does not have valid data and the application should not use the buffer. The actual data transfer takes place while the completion code is USB_XFER_IDLE, although it does not indicate a transfer is in progress.
USB_XFER_STATUS_ACK (0x01)	USB_XFER_STATUS_ACK indicates the control transfer status stage completed successfully. At this time, the application uses the associated data buffer and its contents.
USB_XFER_PREMATURE (0x02)	USB_XFER_PREMATURE indicates that the control transfer was interrupted by the SETUP of a subsequent control transfer. For control writes, the contents of the associated data buffer contains the data up to the premature completion.
USB_XFER_ERROR (0x03)	USB_XFER_ERROR indicates that the expected status stage token was not received.

Customizing HID Class Report Storage Area

If you enable optional HID class support, the Setup Wizard creates a fixed-size report storage area for data reports from the HID class device. It creates separate report areas for IN, OUT, and FEATURE reports. This area is sufficient for the case where no Report ID item tags are present in the Report descriptor and therefore only one Input, Output, and Feature report structure exists. If you want better control over the report storage size or want to support multiple report IDs, you will need to do the following:

1. Use the wizard to specify your device description, endpoints, and HID reports then generate the application.
2. Disable the wizard defined report storage area in *USB_descr.asm*.
3. Copy the wizard created code that defines the report storage area.
4. Paste it into the protected user code area in *USB_descr.asm* or a separate assembly language file.
5. Customize the code to define the report storage area.

Specify Your Device and Generate Application

Use the USB setup wizard to specify your device description, endpoints, and HID reports. Click the **Generate Application** button in PSoC Designer.

Disable Wizard Defined Report Storage Area

In the *USB_descr.asm* file, disable the wizard defined storage area by uncommenting the WIZARD_DEFINED_REPORT_STORAGE line in the custom code area, as shown in the following code:

```
WIZARD: equ 1
WIZARD_DEFINED_REPORT_STORAGE: equ 1
   ;-------------------------------------------------
;@PSoC_UserCode_BODY_1@ (Do not change this line.)
```

```
   ;-------------------------------------------------
   ; Insert your custom code below this banner
  ;-------------------------------------------------
   ; Redefine the WIZARD equate to 0 below by
; uncommenting the WIZARD: equ 0 line
; to allow your custom descriptor to take effect
   ;-------------------------------------------------

; WIZARD: equ 0
  WIZARD_DEFINED_REPORT_STORAGE: equ 0
   ;-------------------------------------------------
   ; Insert your custom code above this banner
   ;-------------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)
```

Copy the Wizard Created Code

Find this code in *USB_descr.asm*.

```
;-----------------------------------------------------------------------
; HID IN Report Transfer Descriptor Table for ()
;-----------------------------------------------------------------------
IF WIZARD_DEFINED_REPORT_STORAGE
AREA  UserModules     (ROM,REL,CON)
.LITERAL
USB_D0_C1_I0_IN_RPTS:
TD_START_TABLE 1                                ; Only 1 Transfer Descriptor
  TD_ENTRY       USB_DS_RAM, USB_HID_RPT_3_IN_RPT_SIZE, USB_INTERFACE_0_IN_RPT_DATA,
NULL_PTR
.ENDLITERAL
ENDIF ; WIZARD_DEFINED_REPORT_STORAGE
```

There are three sections, one each for the IN, OUT, and FEATURE reports. Copy all three sections.

Paste Code Into Protected User Code Area

You can paste the code into the protected user code area of *USB_descr.asm* shown or a separate assembly language file:

```
   ;-------------------------------------------------
   ;@PSoC_UserCode_BODY_2@ (Do not change this line.)
   ;-------------------------------------------------
   ; Redefine your descriptor table below. You might
   ; cut and paste code from the WIZARD descriptor
   ; above and then make your changes.
```

```
   ;-------------------------------------------------

   ;-------------------------------------------------
   ; Insert your custom code above this banner
   ;-------------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)
; End of File USB_descr.asm
```

Customize Code to Define Report Storage Area

To define the report storage area, you need to write your own transfer descriptor table entries. The table contains entries to define storage space for the required data items. Each transfer descriptor entry in the table creates a new Report ID. IDs are numbered consecutively, starting with zero. Report ID 0 is reserved in the USB spec; you cannot use Report ID of 0, but the transfer descriptor entry specified for the ID 0 is used when no Report IDs are present in the Report descriptor. For code efficiency, you must use Report IDs in order starting with ID 1.

Table 4. Transfer Descriptor Table Entries

Table Entry	Required Data Items	Description
TD_START_TABLE	USB_NumberOfTableEntries	Number of Report IDs defined. IDs are numbered consecutively from 0. Report ID 0 is not used.
TD_ENTRY		
	USB_DataSource	The data source is RAM or ROM (USB_DS_RAM or USB_DS_ROM).
	USB_TransferSize	Size of the data transfer in bytes. The first byte is the Report ID.
	USB_DataPtr	RAM or ROM address of the data transfer.
	USB_StatusBlockPtr	Address of a status block allocated with the USB_XFER_STATUS_BLOCK macro.

The following example sets up the unused Report ID 0, and two other IN reports with different sizes. Note Conditional assembly statements are only necessary if you place the code in the protected user code area of *USB_descr.asm.*

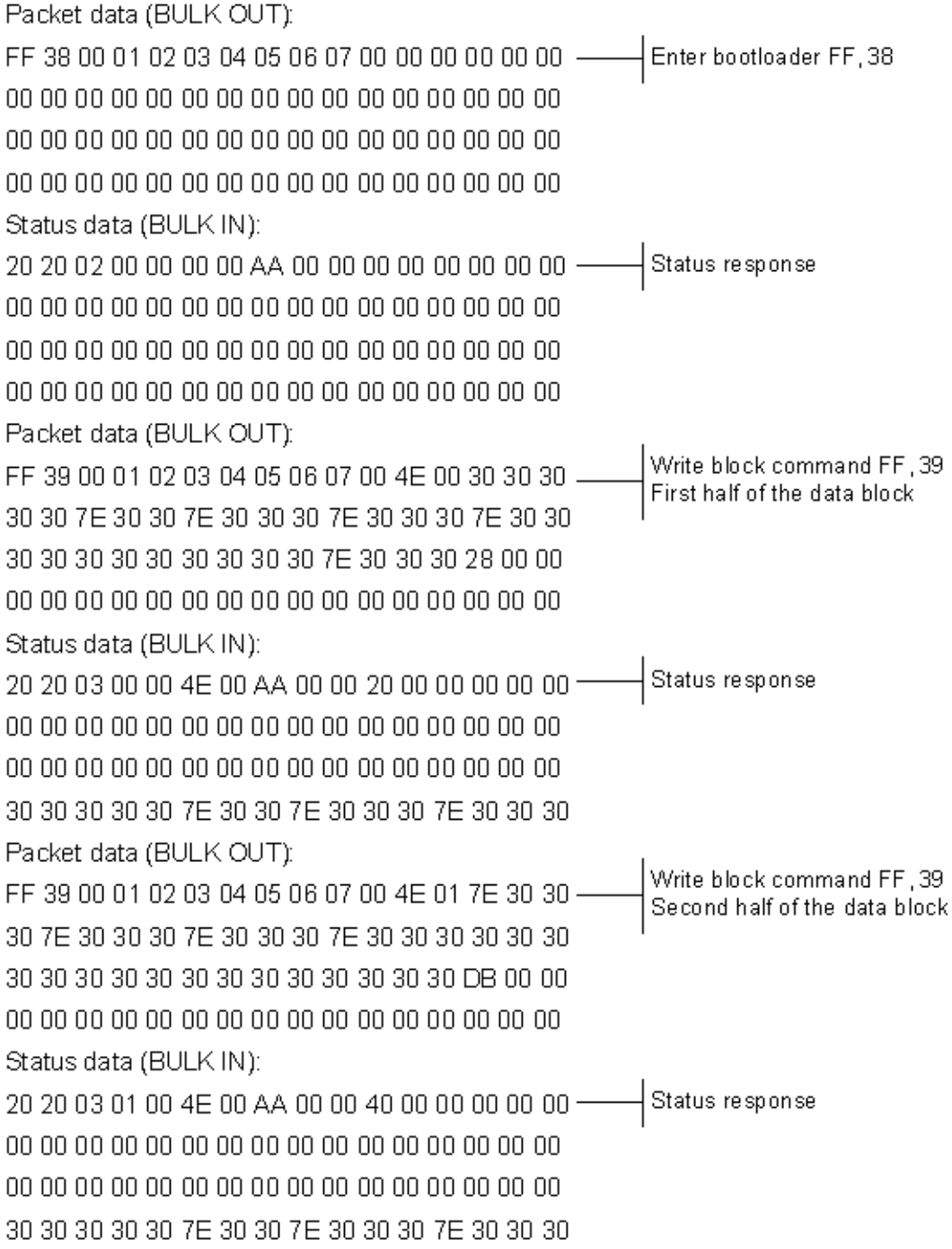
```
;-------------------------------------------------------------------------
; HID IN Report Transfer Descriptor Table for ()
;-------------------------------------------------------------------------
IF WIZARD_DEFINED_REPORT_STORAGE
ELSE

_ID0_RPT_SIZE:   EQU 0       ; 7 data bytes + report ID = 8 bytes (unused)
_SM_RPT_SIZE:    EQU 3       ; 2 data bytes + report ID = 3 bytes
_LG_RPT_SIZE:    EQU 5       ; 4 data bytes + report ID = 5 bytes
```

```
AREA data (RAM, REL, CON)

EXPORT _ID0_RPT_PTR
_ID0_RPT_PTR: BLK 0         ; Allocates space for report ID0 (unused)
EXPORT _SM_RPT_PTR
_SM_RPT_PTR:    BLK 3       ; Allocates space for report ID1
EXPORT _LG_RPT_PTR
_LG_RPT_PTR:    BLK 5       ; Allocates space for report ID2

AREA bss (RAM, REL, CON)

EXPORT _SM_RPT_STS_PTR
_SM_RPT_STS_PTR: USB_XFER_STATUS_BLOCK
EXPORT _LG_RPT_STS_PTR
_LG_RPT_STS_PTR: USB_XFER_STATUS_BLOCK

AREA  UserModules     (ROM,REL,CON)
.LITERAL
EXPORT USB_D0_C1_I0_IN_RPTS:
TD_START_TABLE 3
TD_ENTRY  USB_DS_RAM, _ID0_RPT_SIZE, _ID0_RPT_PTR, NULL_PTR ; ID0 unused
TD_ENTRY  USB_DS_RAM, _SM_RPT_SIZE, _SM_RPT_PTR, _SM_RPT_STS_PTR ; ID1
TD_ENTRY  USB_DS_RAM, _LG_RPT_SIZE, _LG_RPT_PTR, _LG_RPT_STS_PTR ; ID2
.ENDLITERAL

ENDIF ; WIZARD_DEFINED_REPORT_STORAGE
```

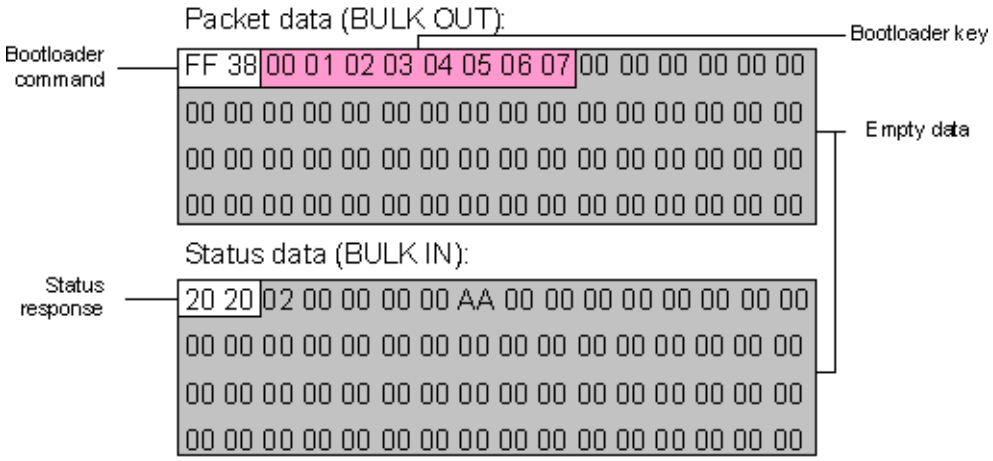
Bootloader USB Download Protocol

Two sample download records are shown in the following figures – the first and last. These records consist of actual data that would be transmitted between the USB master and a slave to be bootloaded. The format of the records is shown here:

Figure 5. The Enter Bootloader Command and the First Data Block



Each command to the bootloader is followed by a response from the bootloader. The following figure shows the format of the Enter Bootloader command:



The first line begins with the bootloader command FF38, enter bootloader. This command is followed by the bootloader key. All bootloader commands must be sent with the bootloader key. The bootloader ignores commands that are not sent with the proper key and sets the “invalid bootloader key” bit in the status response byte. You can set the bootloader key with the BootloaderKey parameter. Other bootloader commands are:

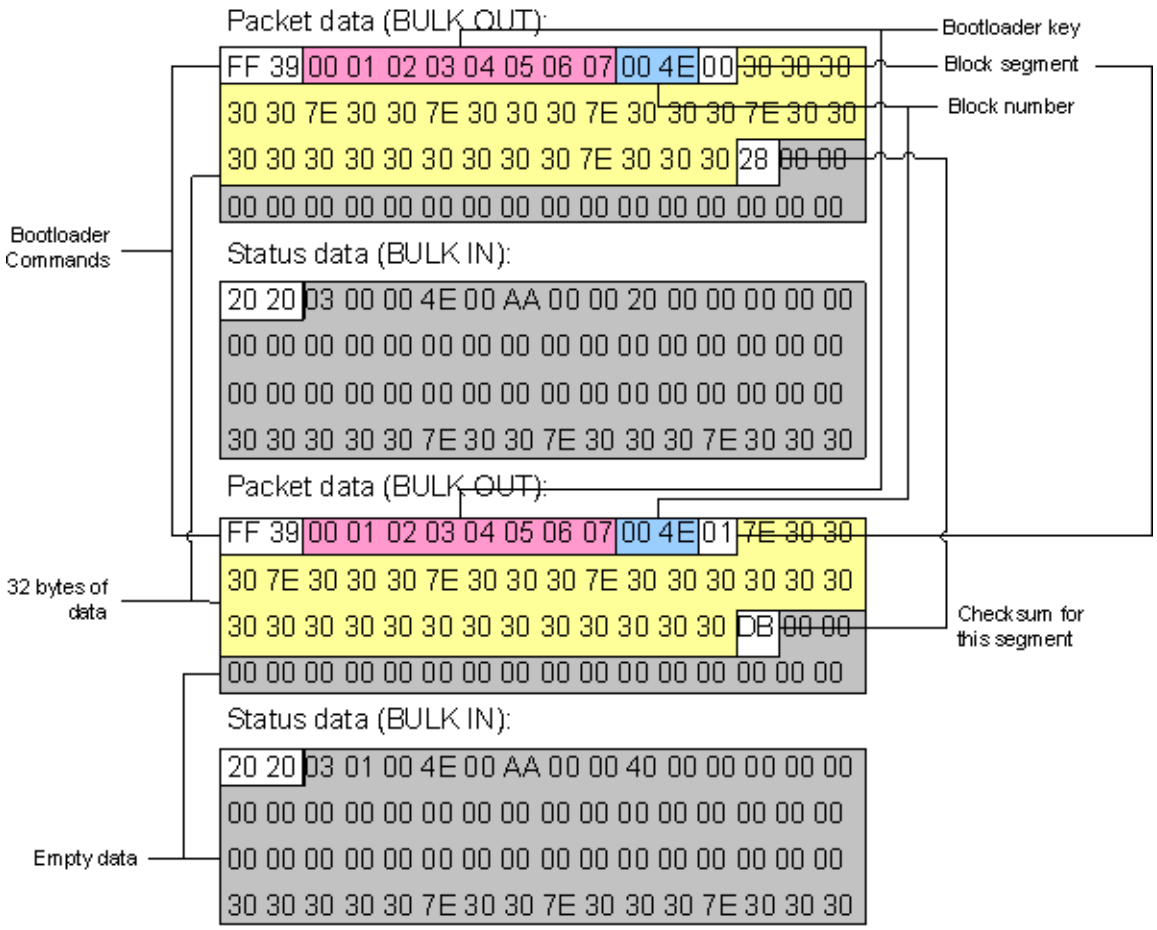
Command	Meaning
FF38	Enter bootloader
FF39	Write block
FF3A	Verify flash
FF3B	Exit bootloader

The command is followed by a status response from the bootloader. The code sent, 0x20, indicates that the bootloader has successfully started. Other status responses are listed in the following table. For details, see the BootLoader operation flowcharts at the end of this document.

Code	Meaning
0x20	Bootload mode (Success). Received ‘Enter bootloader’ command with valid bootLoader Key.
0x01	Boot completed OK. Received ‘Exit bootloader’ command. The checksum of application and relocatable interrupt vector areas calculated by the bootloader matches the checksum received from Host.
0x02	Image verify error. Received ‘Exit bootloader’ command. The checksum of application and relocatable interrupt vector areas calculated by the bootloader does not match the checksum received from the Host.
0x04	Flash checksum error. The flash block content does not match the data received from Host.
0x08	Flash protection error. Flash block cannot be rewritten because its flash protection level does not allow this.
0x10	Comm checksum error. Received a packet with incorrect checksum.
0x40	Invalid bootloader key. A packet with the incorrect bootLoaderKey value was received.
0x80	Invalid command error. Unknown command was received.

Bootloader Write Block Command

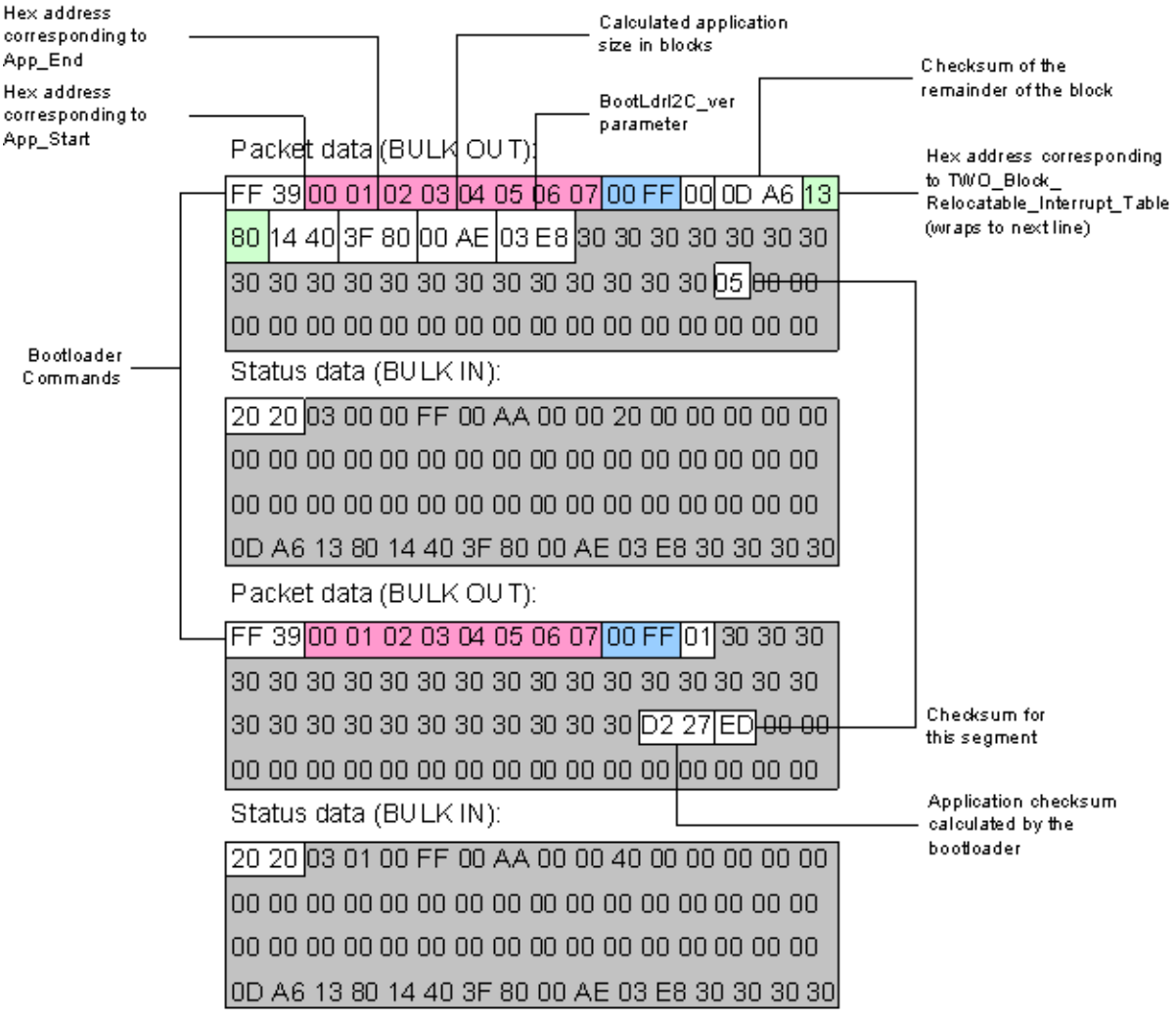
Most of the commands sent to the bootloader are write block commands. The format of each of the write block commands is identical. Each flash block is broken up into 32-byte packets. Each command requires a status response from the slave. The transmission of a 64-byte block is shown in the following figure:



The first line of the first packet consists of a write block command and the bootloader key followed by the block number being transmitted. Since each block is broken, the block number is followed by the block segment number, either 0x00 for the first segment or 0x01 for the second. The last three bytes of the first line, all sixteen bytes of the second line, and the first 13 bytes of the third line represent the 32 bytes of valid data, followed by a checksum for the segment data. The remainder of the block is empty data to pad the segment to 64 bytes.

The status response consists of the status byte transmitted twice and 62 bytes of empty data to pad the segment to 64 bytes.

The format of the second segment of the block is exactly the same as the first. All transmitted data blocks follow this format, except the last block. The last block contains checksums and other necessary data for bootloader operation. The format of the last data block is shown in the following figure.



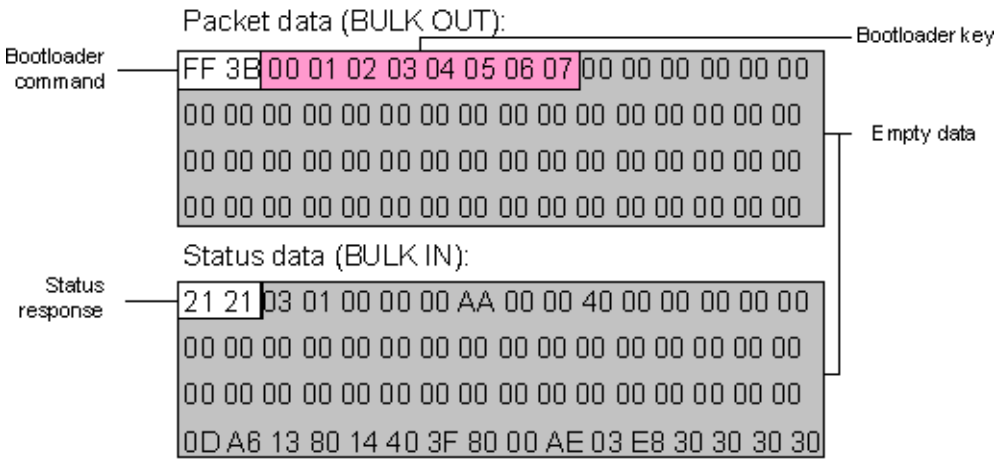
The last record contains the checksum block for this example (note that the block number for this example is 0x00ff but the last block does not have to be 0x00ff).

The first line contains the bootloader write block command, the bootloader key, the block number, and block segment just as the other records did. The next two bytes contain the checksum for the remainder of the block, 0x0DA6 in this case. The last byte of the first line and the first byte of the second line contain the hex address calculated from block 0x4E for the TWO_Block_Relocatable_Interrupt_Table parameter.

The second line then contains a two byte value that represents the hex address of the App_Start user module parameter calculated from block 0x45. The next two bytes are the hex address of the App_End user module parameter calculated from block 0xFE. This is followed by two bytes that are the application size in blocks. The final two bytes of real data value on this line is the bootloader version number from the BootLdrI2C_ver parameter. The remainder of the line and most of the next line is empty data space. The checksum for the segment occupies the same place in the packet that it did for the other packets. The remainder of the packet is empty space.

The second packet of the checksum block begins as all other packets but the only data that it contains is the application checksum and the segment checksum in line three.

The checksum block is followed immediately by the Bootloader Exit command:



The bootloader exit command consists of the bootloader exit command 0xFF3B, and the bootloader key.

The last packet is a final status response.

Figure 6. BootLdrUSBFS User Module Operations Flowchart (64-byte Flash Devices)

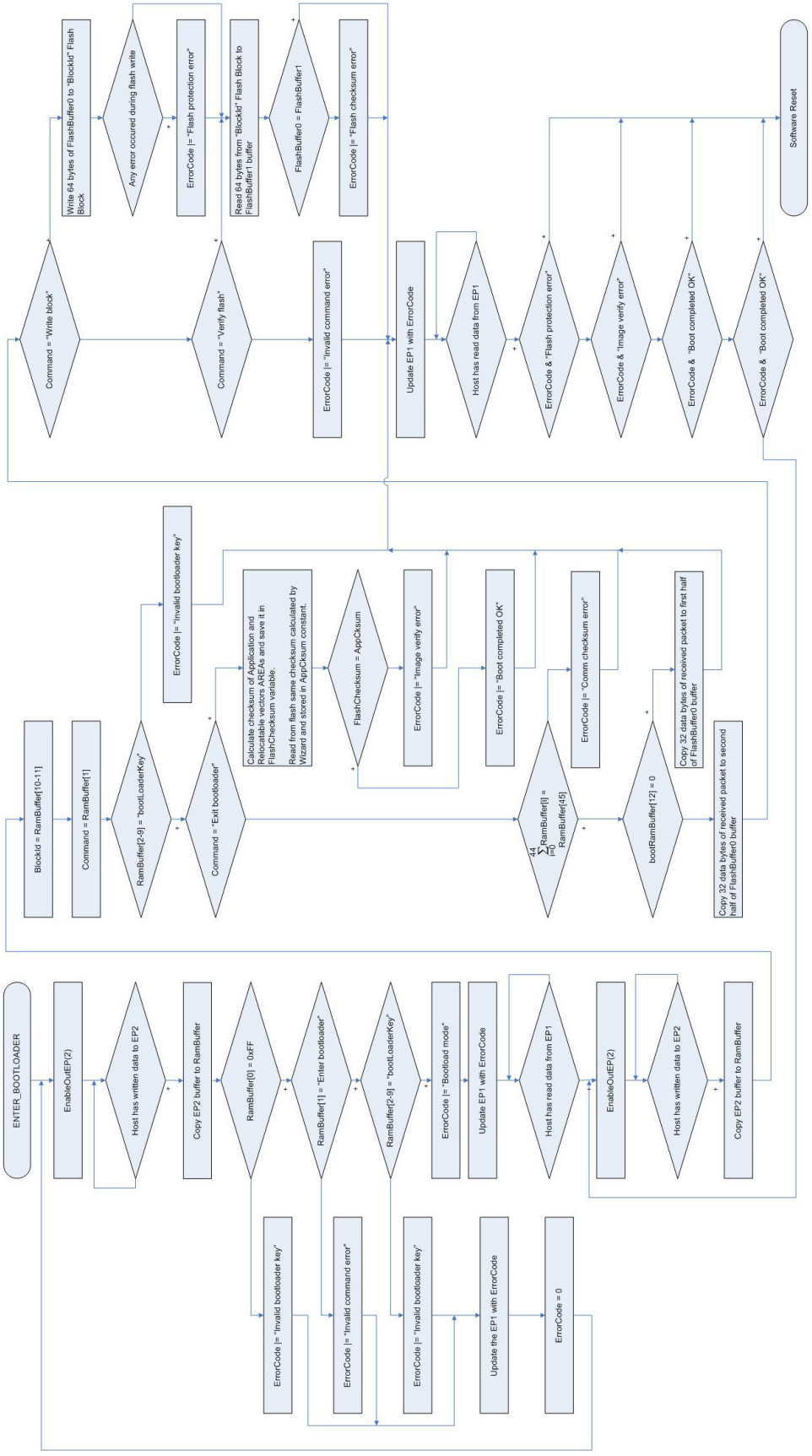
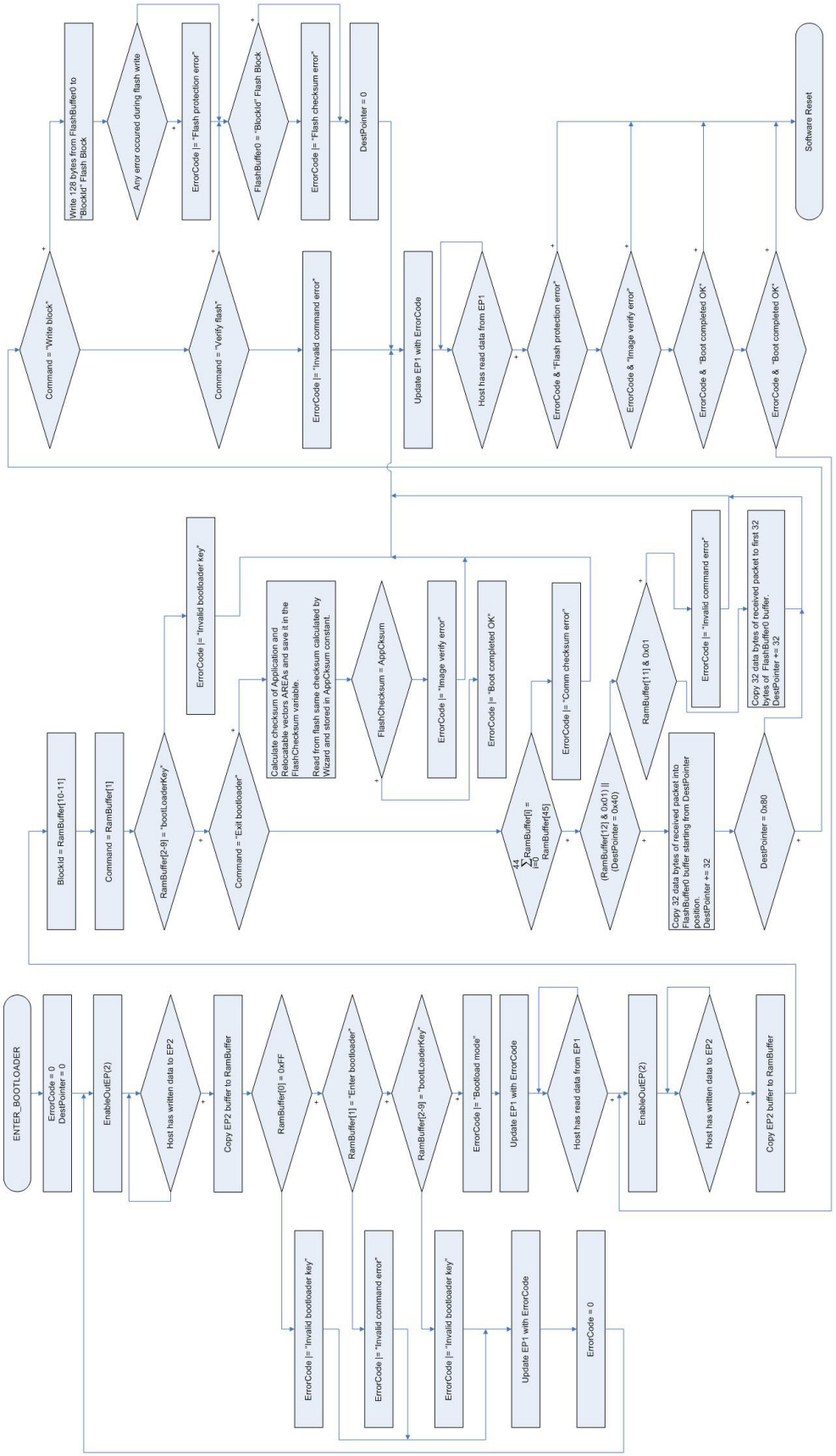


Figure 7. BootLdrUSBFS User Module Operations Flowchart (128-byte Flash Devices)



Version History

Version	Originator	Description
1.2	DHA	Added Version History.
1.30	DHA	1. Removed .Literal/.Endliteral directives around jmp instructions. 2. Removed Directives .SECTION and .ENDSECTION for USBFS_bGetProtocol and USBFS_UpdateHIDTimer functions in the USB_cls_hid.asm.
1.40	DHA	1. Added wizard help file. 2. Added verification of the writing EP0_CR. 3. Added verification of the SIE MODEs and ACK bit into the EP0 ISR. 4. Increased Bootloader area. 5. Changed `UM Name`_MAXMEM to `UM Name`_MemMax in USB_Bt_loader.asm file.
1.50	DHA	1. Moved Application Descriptors from "AREA UserModules" to "AREA lit". 2. Added initialization of USB_Protocol variable to comply with HID specification. 3. Added support to display bootloader output files in Workspace Explorer. 4. Corrected the Large Memory Model mode directives to address compile errors. 5. Added the automatic frequency locking of the internal oscillator to the USB traffic into bootResetIsr() function for Encore III devices. 6. Added description in the "Improper Settings in Flashsecurity.txt" section.
1.60	DHA	1. Updated area declarations to support Imagecraft optimization. 2. Added Javascript updates to enhance wizard performance. 3. Fixed issues related to coexistence with the watchdog timer. 4. Corrected the first data packet PID value to address an issue with the EnterBootloader command.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.