USBFS Bootloader Data Sheet

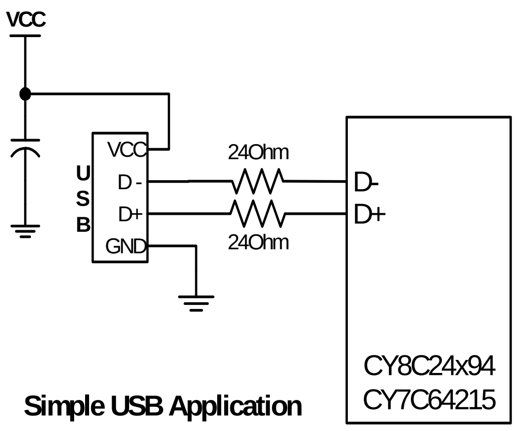
BootLdrUSBFS vX.Y

Resources	PSoC® Blocks			API Memory (Bytes)		Pins (from Single Port)
	Digital	Analog CT	Analog SC	Flash	RAM	
CY8C24x94, CY8CLED04, CY7C64215 HID support	0	0	0	4982	46	2
CY8C24x94, CY8CLED04	0	0	0	4516-4982	46	2

Note. Expect an expansion of Flash and RAM when adding additional interfaces, HID classes, and other USBFS extensions. When the bootloader is in actual operation it makes use of a large amount of RAM to download program data but then frees it up upon exit. Since operation of the bootloader precludes 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 an additional 2K-bytes above the normal requirement of about 1.9K bytes of code used by USB itself.

Features and Overview

- Flexible memory map
- Device reprogramming without engineering tools
- Product resident reprogramability
- 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



USBFS Device Block Diagrams

Functional Description

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

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

The USBFS User Module provides:

- 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. If you want to change your choice after initial selection, you must delete the existing instance of the USBFS User Module and then add a new instance.

The bootloader portion of the user module provides 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 will be 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 below.

Once 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 provided. A built in set of tools can be accessed by right-clicking on the bootloader icon in the device manager view. Additionally a host application (including source code) is provided 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, resources examples, and forums regarding use of USB can be found at www.usb.org.

Quick Start

- Review this data sheet. A successful implementation of a bootloader project requires an understanding of this information.
- Add the user module to a project.
- Place the user module, selecting either a HID or NON-HID class application.
- Right-click on the user module icon. Select **Boot Loader Tools**. Select **Get Files**. When finished, the boot.tpl, custom.lkp, and flashsecurity.example files should be in the project root directory.
- 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. There should be one present by default, if not add at least a one character string. select **OK**.
- Right-click on the user module icon. Select **Device: Bootloader USB Setup Wizard…** It is not necessary to make any modifications. Select **OK**.
- From the **Project > Settings > Linker** dialog box, set the **Relocatable code start address** to 0x13c0 to avoid attempting to create application code in the bootloader ROM area. If these settings leave unused ROM areas the settings can be optimized later. The linker gives rather un-helpful messages when it encounters memory overlap errors. Initial project development will be less frustrating in the long run if linker problems can be kept to a minimum.
- Generate source code and compile the project.
- Review the output file <project>.mp to and <project>.hex to see how the project has been built.
- After creating a project that compiles without errors go to the Sample Firmware Code section. Modify and adapt the code provided in the sample.

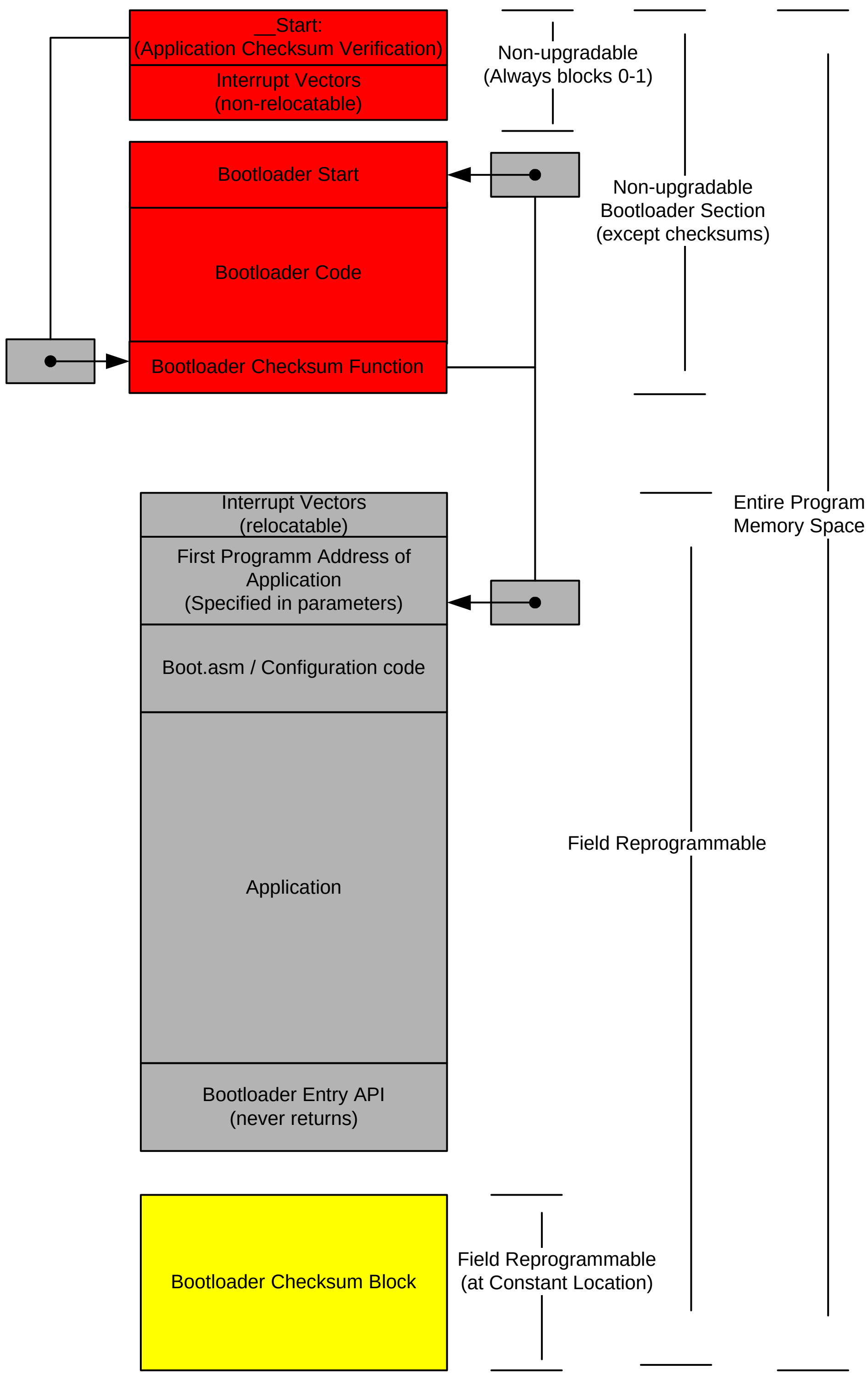
Theory of Operation

Creating a project with a bootloader requires several non-standard modifications to the PSoC Designer standard model. To facilitate this, the bootloader user module provides 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 provided as part of the user module, a host application example is provided as part of 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 this user module.

```
<install_path>\Cypress MicroSystems\PSoC Designer\Examples\BootLdrUS-
BFS\USB_BootLoaderHostApp/…
```

Use of 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 above.

The included driver.inf file should be modified to corrrectly 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.



Bootloader Memory
Organization

USBFS Bootloader Memory Organization

Overview

PSoC Designer makes use of standardized files, built in data about the part family, and attributes of specific devices to create compilable projects 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 will be maintained throughout the life of the design. While a project without the requirements of a bootloader simply allows the compiler and linker to allocate RAM and ROM, a bootloader must group RAM and ROM in specific areas so that the program will not crash while a new application is being loaded.

In the memory layout above there are four 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 re-programmed. The first two blocks of ROM should 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 but once the project or device containing the bootloader is deployed, this area is not re-programmable and cannot be field upgraded.
- The third 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 re-written. Consider the problems that a program might encounter if executing code were to be changed during execution. This requirement can easily be met by turning off all application interrupts during bootload. This is automatically done when the bootloader is started. In addition to interrupt vectors, the application area also contains most of the device bootcode and all of the foreground runtime application.
- The fourth 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 always be operational, including during a bootload process, the design of the Bootloader User Module can allow sufficient customization to accommodate this. The best way to accomplish this is to add this code to the bootloader ROM area using the assembler AREA directive. Any RAM used by your code during the bootload process needs to be added to the RAM area defined for the bootloader.

Definition of Memory Areas in the User Module Parameters

The parameters available in the USBFS Bootloader User Module allow you to customize where major program elements are placed in ROM. The defaults in the user module should provide a working initial setup. You should use these settings until a complete project is successfully compiled. Once you have a compiled 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 provides a complete utility that coexists with a your foreground application. When the device is started or reset, the bootloader utility is always invoked. Once invoked at system startup the bootloader will validate the foreground application by calculating a checksum on the foreground application ROM area. The calculated checksum will be compared to the one stored in the checksum block (which is created with the application). If the two checksums are equal, the bootloader utility will allow 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 susbsystem to allow the host to transmit data. When the host system observes this interface is enabled, it may choose to execute its own set of applications. Although a default USB descriptor is provided that will run successfully with the examples provided, 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 provided in the installation directory for this user module:

```
<install_path>\Cypress MicroSystems\PSoC Designer\Examples\BootLdrUS-
BFS\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 and custom.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 – Once your project builds correctly you can use the bootloader tools to create and auto-validate checksums. Upon entry into the bootloader tools selection screen, 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 will be reported but no error debug information will be 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 will be necessary to use the conventional build and generate process external to the bootloader tools menu.

Generate dld file – This tool item will derive a download file from the hex project output file. This file contains only the hex blocks that will be 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.

Checksum Semiautomatic Generation

Once your project can be 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, as soon as 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 prior to a successful compile. Once a checksum is created, it must be integrated into the compiled files. This requires a second compile. Last, to validate the block of code that the checksum is stored in, a checksum on that block itself is provided. This requires a third compile. Multiple compiles are required because the built-in checksum parameters are pre-compile elements that can only be calculated post-compile.

Special Files Provided

Two important files are provided by accessing the **Bootloader Tools** menu and selecting **Get Files**. A device specific boot.tpl file is placed in the main project directory along with a file called custom.lkp and a pre-defined flashsecurity.txt file. A brief description of the purpose of each file follows: The original version of each of these files is placed in the project backup directory.

Boot.tpl. – This file contains a relocatable and non-relocatable 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 auto-generated 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 re-written under control of the code generation software. Changes made within or below the last three lines will either cause an error or simply be lost. Changes may be made to the rest of the custom.lkp file. For the purposes of debugging the memory allocation of the project, it is possible to comment out all three lines mentioned above by inserting a semicolon in the first space. This will allow the linker to place code automatically and may be helpful in determining application code size requirements.

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 this file may need to be modified by hand according the final memory may 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. The provided file, flashsecurity.example, may be edited and renamed as necessary.

Flashsecurity.txt – This is a default file provided 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 will 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 an additional 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. Again, this is to overcome the problem of executing code that is being re-written (never 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 will be 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_CNTL1) to control the D+ and D- pins.

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 developers may not use VIDs and PID's assigned to Cypress or Cypress Microsystems.

Block Entry of Parameters

All memory parameters are entered in the bootloader in blocks numbered from 0x00 through 0xFF for 16K devices; 0x00 through 0x7F for 8K devices; and 0x00 through 0x1FF for 32K devices. Although this is not the most convenient format to enter memory addressees, it does prevent accidental assignment of partial block addresses to different sections of the memory map. The PSoC devices in question are only capable of storing 64byte flash blocks (128byte 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 bootloader built in may be difficult to debug. Because of this there are additional adjustments that can be made 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 a user to lengthen, shorten, or make infinite, code that will timeout if no communication is received from a host once 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 and CY7C64215 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.

Parameters

The USBFS Bootloader consists of a bootloader utility integrated with a fully functional USBFS user module.

Parameters defined for the bootloader allow the developer to define where major program blocks will be located when the program is compiled and linked.

TWO_Block Relocatable_Interrupt_Table

Two blocks of the ROM area are used to define a set of interrupt tables that match the ones that are always present in blocks 0 and 1 of PSoC ROM. These tables must remain located at the same point in memory since the block 0 and 1 interrupt tables are re-directed 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.

The default for this value is located immediately after the Bootloader application at block 0x4B/0x12C0.

ApplicationCode_Start_Block

This is the first block of code assigned to the user application. This code will be 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.

The default value is 0x4D/0x1340.

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. Once the application is completed this value may be reduced to save processing time during the checksum operation and to 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

The default value is 0xFE/0x3F80.

Application_Checksum_Block

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

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

The default value is 0xFF/0x3FC0.

Bootloader_Start_Block

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

The default value is 0x2/0x0080.

Bootloader_Size

This is the size in blocks for the bootloader application. Normally this size will not require adjustment. If a user wished to add code to the bootloader he or she could increase this size to accommodate the extra code. Other blocks below would have to be similarly adjusted.

The default value is 0x4A blocks in length.

Bootloader_Key

This is the key value prepended to the transactions sent to the bootloader application representing an extra verification step to make sure the bootloader upgrade utility is not accidentally.

The default value "0001020304050607".

Flash_Program_Temperature_Deg_C

This is the typical programming temperature expected when the device is re-programmed. Programming the device at 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 result in reduced memory retention. For this reason, you should take precautions to ensure that the bootloader is never operated more than 20 degrees C 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 will be 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. It may be set by the user and used to verify the correct version of bootloader executable code.

Common Problems

Internal use of the Watch Dog Timer

Coordination with the watchdog timer is linked to the global parameter WATCHDOG_ENABLE, contained in the file globalparams.inc.If the project uses a watchdog timer, conditionally compiles code linked to the global parameter and automatically pets the watchdog during bootload checksum and download operations. CPU clock speed will effect how fast the watchdog timer is updated. A practial minimum setting for the watchdog timer is about 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 provided in the file “Flashsecurity.example”. Flashsecurity.example is provided with the BootLoader Tools - Get Files user module menu item. The map must allow flash write at all the locations that will eventually be bootloaded. One strategy would be to make all blocks writeable. Another would be to take a moment to layout your memory map now and edit this file accordingly. No matter which strategy is chosen, taking action at the beginning of the project will be 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.

Incorrect Relocatable Code Start Address (Linker Parameter)

Since the memory map for a bootloader project is considerably different than that for a conventional project, the relocatable code start address will usually need to be altered. This is a comon 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 changes 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 un-used area of ROM. For the USB version of the bootloader, setting this value to 0x1440 should be adequate (if the default values fo the other parameters are used).

Memory Overlap

To correct the relocatable code start address (see above), use a leading semi-colon to comment out the last three lines of the custom.lkp file, and attempt to build the file again and examine the resulting memory map. Memory overlap problems are difficult to diagnose since they prevent output files from being generated. Modifying the custom.lkp file may allow the linker to place object blocks which will then provide a starting point for correcting the memory overlap root cause.

Power Stability

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 power levels changing when flash programming is taking place. One example is some sort of status write to flash at power up. You should take care to 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.

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 especially easy to do this unintentionally. For example, a main{} function with a simple while(1); loop will never return and never enter the bootloader. Therefore, it cannot be reprogrammed once 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 could be configured to enter the bootloader upon reset and exiting to the foreground application when the timeout expires.
2. Set a test at some point in the code that will cause the device to enter the bootloader. This could 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 will drop off the USB bus briefly, and when it returns it shold enumerate as a bootloader.
4. Make use of the watch dog timer to reset the device if it is not serviced regularly. This could be combined with one of the above strategies to allow a WDT interrupt to initiate a bootloadable state. Upon reset from a watch dog 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 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 re-programmable applications must be identical. All bootloader parameters should be identical, 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. Once 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.

Application Programming Interface

A discussion of the APIs for the bootloader and the USB functionality contained within the bootloader follows. The Bootloader contains a very limited set of APIs since the main purpose of the Bootloader is to completely remove and replace the user application. Since the Bootloader is not reprogrammed.

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:
```
call  ENTER_BOOTLOADER ; Call the Start Function
```

Alternately:

```
GenericHardDefinition: equ (0x7C)
call  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:
```
call  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:
```
call bootLoaderVerify             ; Call the Start Function
```
Parameters:

None.

USBFS APIs

The application programming interface (API) routines in this section allow programmatic control of the USBFS User Module. The sections that follow describe descriptor generation and integration, and list the basic and device specific API functions. As a developer 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 USBFS_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 In all user module APIs, the values of the A and X registesr may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X prior to 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 will do so in the future.

Note The API routines for the USB user modules are not reentrant. 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 USBFS_Start(BYTE bDevice, BYTE bMode)	Activate the user module for use with the device and specific voltage mode.
void USBFS_Stop(void)	Disable user module.
BYTE USBFS_bCheckActivity(void)	Checks and clears the USB bus activity flag. Returns 1 if the USB was active since the last check, otherwise returns 0.
BYTE USBFS_bGetConfiguration(void)	Returns the currently assigned configuration. Returns 0 if the device is not configured.
BYTE USBFS_bGetEPState(BYTE bEPNumber)	Returns the current state of the specified USBFS endpoint. 2 = NO_EVENT_ALLOWED 1 = EVENT PENDING 0 = NO_EVENT_PENDING
BYTE USBFS_bGetEPAckState(BYTE bEPNumber)	Identifies whether ACK was set by returning a nonzero value.
BYTE USBFS_wGetEPCount(BYTE bEPNumber)	Returns the current byte count from the specified USBFS endpoint.
void USBFS_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 USBFS_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 USBFS_DisableOutEP(BYTE bEPNumber)	Disables the specified USB endpoint to NAK OUT transfers

USBFS_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 USBFS_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 USBFS_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 USBFS_Start(BYTE bDevice, BYTE bMode)
```

Assembly:

```
mov   A, 0xFF                ; The bootloader device descriptor
mov   A, 0                   ; Select application device descriptor
mov   X, USBFS_5V_OPERATION  ; Select the Voltage level
call  USBFS_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 provided in C and assembly, and their associated values are given in the following table.

Mask	Value	Description
USBFS_3V_OPERATION	0x02	Disable voltage regulator and pass-through vcc for pull-up
USBFS_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:

```
call 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:

```
call 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:

call USBFS_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 modellarge 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 table below 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
call BootLdrUSBFS_bGetEPState
```

Parameters:

Register A contains the endpoint number.

Return Value:

Returns the current state of the specified USBFS endpoint. Symbolic names provided in C and assembly, and their associated values are given in the following table. Use these constants whenever the user writes 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 enpoint 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
call 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
call 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
call 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 provided 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 transmis-sion

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
call 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
call 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
call 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 provides 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
call BootLdrUSBFS_Force
```

Parameters:

bState is byte indicating which among four bus states to enable. Symbolic names provided in C and assembly, and their associated values are listed here:.

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
call BootLdrUSBFS_UpdateHIDTimer
```

Parameters:

Register A contains the interface number.

Return Value:

The state of the HID timer is returned in A. Symbolic names provided in C and assembly, and their associated values are given here:

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
call BootLdrUSBFS_bGetProtocol
```

Parameters:

bInterface contains the interface number.

Return Value:

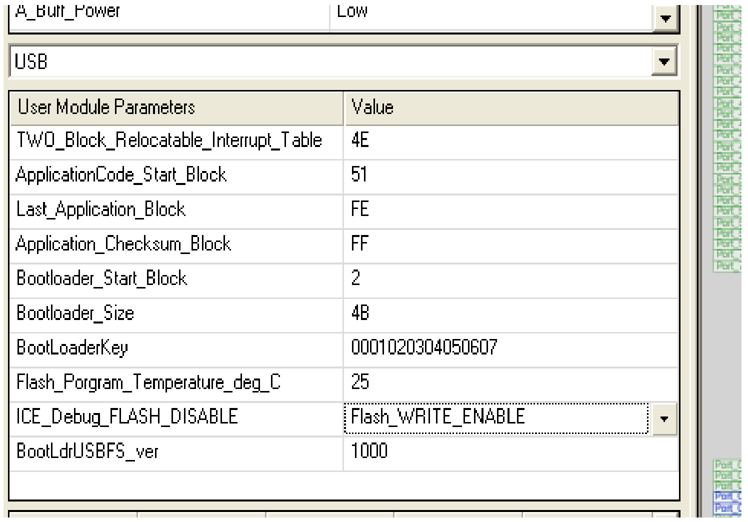
Register A contains the protocol value.

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.

Sample Code

For both the C and assembly language example projects configure the bootloader user module as shown below.

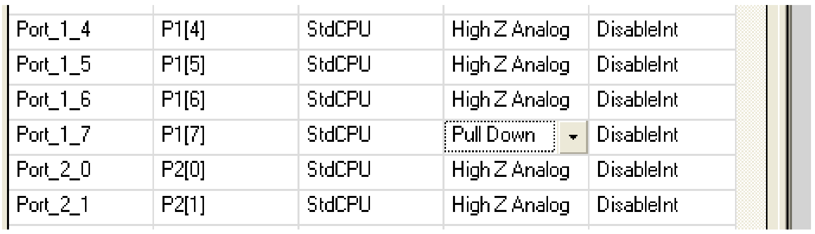


A_Buff_Power	Low

USB

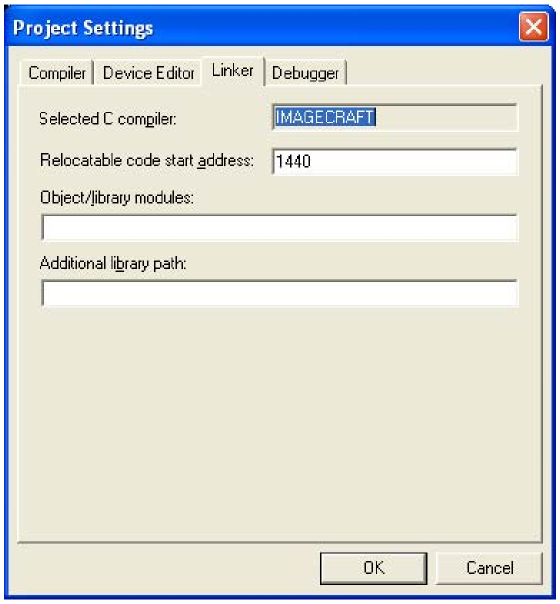
User Module Parameters	Value
TWO_Block_Relocatable_Interrupt_Table	4E
ApplicationCode_Start_Block	51
Last_Application_Block	FE
Application_Checksum_Block	FF
Bootloader_Start_Block	2
Bootloader_Size	4B
BootLoaderKey	0001020304050607
Flash_Porgram_Temperature_deg_C	25
ICE_Debug_FLASH_DISABLE	Flash_WRITE_ENABLE
BootLdrUSBFS_ver	1000

Make sure the correct pin is configured as a pull down to recognize the switch closure to enter the bootloader.



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

Make sure the code start address is set correctly.



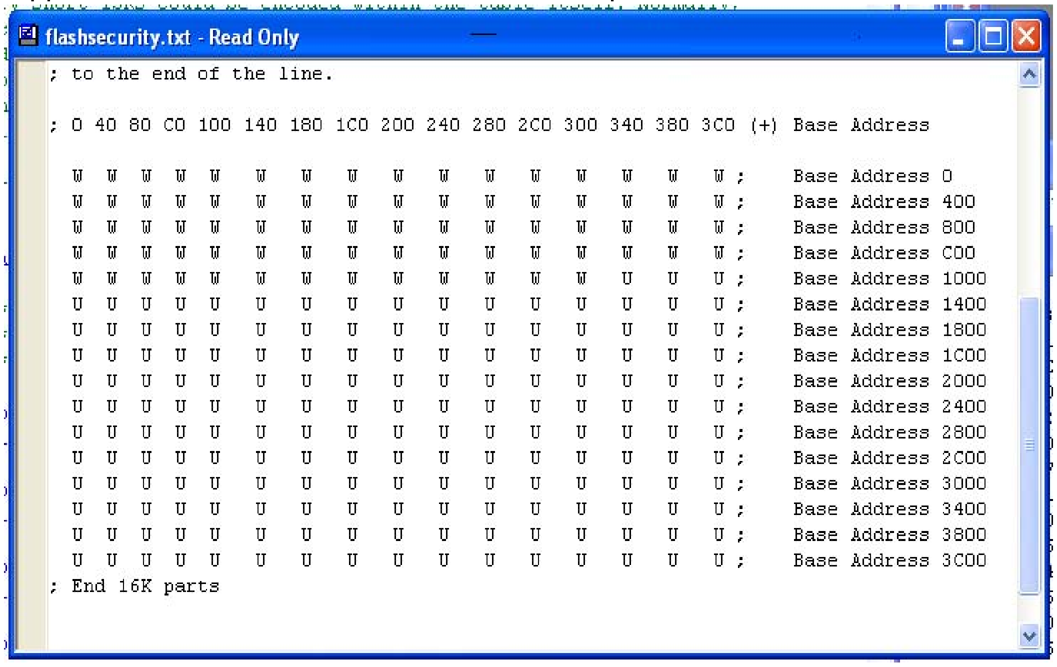
Selected C compiler: IMAGECRAFT

Relocatable code start address: 1440

Object/library modules:

Additional library path:

Make sure that before you attempt to bootload an application you modify the flashsecurity.txt file to make the application, checksum, and relocatable interrupt vector areas writeable.



```
; 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   U   U   U ;    Base Address 1000
  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 ;    Base Address 1800
  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 ;    Base Address 2000
  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 ;    Base Address 2800
  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 ;    Base Address 3000
  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 ;    Base Address 3800
  U  U  U  U  U   U   U   U   U   U   U   U   U   U   U   U ;    Base Address 3C00
; End 16K parts
```

Please note that the included Host PC example project expect to see a VID of 04b4 and a PID of E006. These are Cypress owned IDs and may be used for local debug but may not be released for production.

Below is an implementation of the USB Bootloader User Module written in C.

```
//
//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()
{
   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 direc-
tion 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
}
```

Below 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. Once 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 the how the BootLdrUSBFS Setup Wizard configures the user module. This project is identical to that in the USBFS user module 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 USBFS_Start
POP X

; Wait for Device to enumerate
.no_device:
PUSH X
LCALL USBFS_bGetConfiguration
POP X
CMP A,0
JZ .no_device
; Enumeration is completed load endpoint 1. Do not toggle the first time
; USBFS_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 USBFS_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);
;                 USBFS_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 USBFS_Stop
.wait_for_bit_low:
tst reg[PRT1DR], 0x80
jnz .wait_for_bit_low
; once it goes low enter the bootloader
pop A
ljmp USBFS_EnterBootloader ;;never returns
halt

.Exit_BOOTLOAD_TEST:
pop A


;;; mouse operations



PUSH X
MOV A,1
LCALL USBFS_bGetEPAckState
POP X
CMP A,0
JZ .endless_loop
; ACK has occurred, load the endpoint and toggle the data bit
; USBFS_LoadInEP(1, abMouseData, 3, USBFS_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 USBFS_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. For the purpose of the examples above, the user module has been renamed “USB” in the first C-based example and “USBFS” in the assembly based example. Both examples use the BootLdrUSBFS user module.
3. In the Device Editor, click **Protocols** and add the BootLdrUSBFS User Module by double-clicking the BootLdrUSBFS icon or right-clicking and choosing **Select**.
4. Select the Human Interface Device (HID) radio button.
   Optional step: rename the user module from BootLdrUSBFS_1 to USBFS to match the sample code by right-clicking the module and selecting **Rename**.
5. 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 make the name 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.
6. Click **OK** to save the USB descriptor information.
7. Right click the USBFS User Module icon in the Device Editor to open the “Device: BootLoader USB Setup Wizard”.
8. 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.
9. Click **OK** to save the USB BootLoader descriptor information.
10. Generate the Application.
11. Copy the Sample code and paste it in the main.c.

12. Do a **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
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

Descriptor	Data
String	My mouse
Descriptor	
HID report descriptor root	USBFS
HID report descriptor	USBFS

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 will add 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

USB Standard Device Requests

This section describes the requests supported by the USBFS user module. If a request is not supported the USBFS 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

Standard Device Request	USB User Module Support Description	USB 2.0 Spec Section
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 will add support to this request to enable Isochronous transfers with repeating frame patterns.	9.4.11

HID Class Request

Class Request	USBFS 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 provided 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 provided by Cypress to assist engineers in the designing of 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, you see all the setup wizard options. 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	

Descriptor	Data		Operations
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	
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 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 then click **Save/Generate**. Once 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 here you add or import an 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 permits the entering of 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 provides a set of API functions.

USFS Activity Monitoring

The USBFS_bCheckActivity API function provides a means to check if any USB bus activity occurred. The application uses the function to determine if the conditions to enter USB Suspend were met.

USBFS Suspend

Once 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 USBFS_Suspend API function and the USBFS_bCheckActivity API to detect USB activity. This function disables the USBFS block, but maintains the current USB address (in the USBCR 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 were met. One way to check resume conditions is to use the sleep timer to periodically wake the device. If the resume conditions were met, the application calls the USBFS_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 USBFS_bRWUEnabled API function. When the device is suspended and it determines the conditions to initiate a remote wakeup are met, the application uses the USBFS_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 of USBFS Device Requests

All control transfers, including vendor specific and overriden 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 USBFS User Module, all control transfers are handled by the endpoint 0 Interrupt Service Routine (USBFS_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 provided by the application. The user module handles the data and status stages of the transfer without any involvement of the user application. Upon completion of the transfer, 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 USBFS_EP0_ISR, which routes the setup packet to the USBFS_bmRequestType_Dispatch routine. From here all the standard device requests as well as the 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 Vendor Specific Request Dispatch Routine Names.

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 provide 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 the table above.

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 USBFS_InitControlWrite
```

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 the following table. The routine may destroy the A and X registers, but the Stack Pointer (SP) and any other relevant context must be restored prior to returning control to the ISR.

Endpoint 0 ISR Return Points

Return Entry Point	Required Data Items	Description
USBFS_Not_Supported	Use this return point when the request is not supported. It STALLs the request.	
	Data Items: None	
USBFS_InitControlRead	This return point is used to initiate a Control Read transfer.	
	USBFS_DataSource (BYTE)	The data source is RAM or ROM (USBFS_DS_RAM or USBFS_DS_ROM). This is necessary since different instructions are used to move the data from the source ROMX or MOV.
	USBFS_TransferSize (WORD)	The number of data bytes to transfer.
	USBFS_DataPtr (WORD)	RAM or ROM address of the data.
	USBFS_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USBFS_XFER_STATUS_BLOCK macro.
USBFS_InitControlWrite	This return point is used to initiate a Control Write transfer.	
	USBFS_DataSource (BYTE)	USBFS_DS_RAM (the destination for control writes must RAM).
	USBFS_TransferSize (WORD)	Size of the application buffer to receive the data
	USBFS_DataPtr (WORD)	RAM address of the application buffer to receive the data
	USBFS_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USBFS_XFER_STATUS_BLOCK macro.
USB_InitNoDataControlTransfer	This return point is used to initiate a No Data Control transfer.	
	USBFS_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USBFS_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.

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 com-pleted 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 the 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 the 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.

```
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  func_lit     (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 the Code Into the 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 the Code to Define the Report Storage Area

To define the report storage area, you will write your own transfer descriptor table entries. The table contains entries to define storage space for the reqired 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 will be used when no Report IDs are present in the Report descriptor. For the sake of code effeciency, you should use Report IDs in order starting with ID 1.

Transfer Descriptor Table Entries

Table Entry	Required Data Items	Description
TD_START_TABLE	USBFS_NumberOfTableEntries	Number of Report IDs defined. IDs are numbered consecutively from 0. Report ID 0 is not used.
TD_ENTRY		
	USBFS_DataSource	The data source is RAM or ROM (USBFS_DS_RAM or USBFS_DS_ROM).
	USBFS_TransferSize	Size of the data transfer in bytes. The first byte is the Report ID.
	USBFS_DataPtr	RAM or ROM address of the data transfer.
	USBFS_StatusBlockPtr	Address of a status block allocated with the USBFS_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: USBFS_XFER_STATUS_BLOCK
EXPORT _LG_RPT_STS_PTR
_LG_RPT_STS_PTR: USBFS_XFER_STATUS_BLOCK

AREA  func_lit     (ROM,REL,CON)
.LITERAL
```

```
EXPORT USB_D0_C1_I0_IN_RPTS:
  TD_START_TABLE 3
  TD_ENTRY  USBFS_DS_RAM, _ID0_RPT_SIZE, _ID0_RPT_PTR, NULL_PTR ; ID0
unused
  TD_ENTRY  USBFS_DS_RAM, _SM_RPT_SIZE, _SM_RPT_PTR, _SM_RPT_STS_PTR ; ID1
  TD_ENTRY  USBFS_DS_RAM, _LG_RPT_SIZE, _LG_RPT_PTR, _LG_RPT_STS_PTR ; ID2
.ENDLITERAL

ENDIF ; WIZARD_DEFINED_REPORT_STORAGE
```

Appendix

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

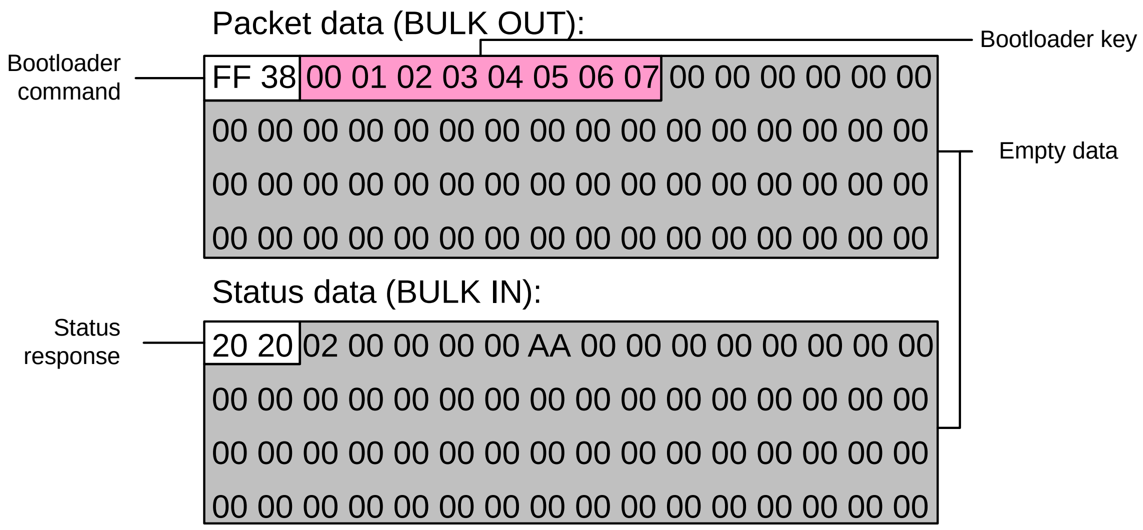
Bootloader USB Download Protocol

Two sample download records are shown below – the first and the 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 detailed below.

```
Packet data (BULK OUT):
FF 38 00 01 02 03 04 05 06 07 00 00 00 00 00 00  Enter bootloader FF, 38
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Status data (BULK IN):
20 20 02 00 00 00 00 AA 00 00 00 00 00 00 00 00  Status response
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Packet data (BULK OUT):
FF 39 00 01 02 03 04 05 06 07 00 4E 00 30 30 30  Write block command FF, 39
                                                 First half of the data block
30 30 7E 30 30 7E 30 30 30 7E 30 30 30 7E 30 30
30 30 30 30 30 30 30 30 30 7E 30 30 30 28 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Status data (BULK IN):
20 20 03 00 00 4E 00 AA 00 00 20 00 00 00 00 00  Status response
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 30 30 30 30 7E 30 30 7E 30 30 30 7E 30 30 30
Packet data (BULK OUT):
FF 39 00 01 02 03 04 05 06 07 00 4E 01 7E 30 30  Write block command FF, 39
                                                 Second half of the data block
30 7E 30 30 30 7E 30 30 30 7E 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 DB 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Status data (BULK IN):
20 20 03 01 00 4E 00 AA 00 00 40 00 00 00 00 00  Status response
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 30 30 30 30 7E 30 30 7E 30 30 30 7E 30 30 30
```

The Enter Bootloader Command and the First Data Block

Each command to the bootloader is followed by a response from the bootloader. The following illustration details 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 will ignore commands that are not sent with the proper key. You set the bootloader key with the Bootloader_Key parameter. Other bootloader commands are:

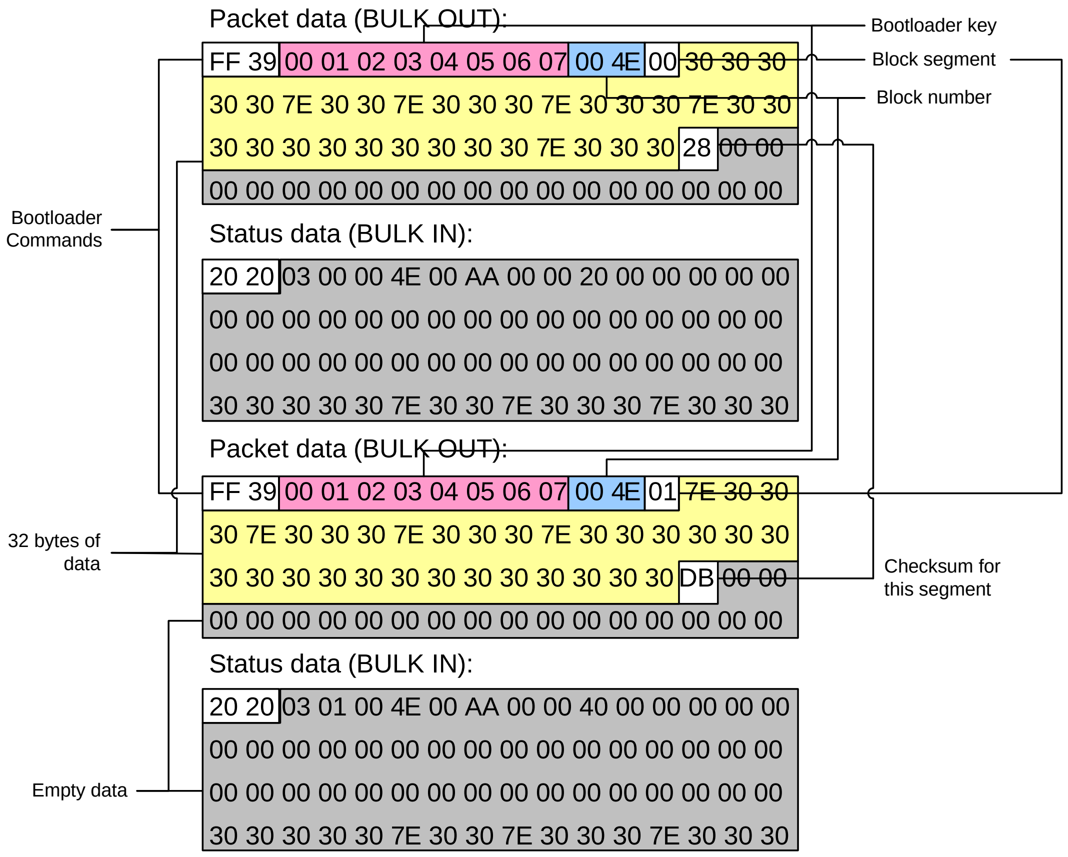
Command	Meaning
FF38	Enter bootloader
FF39	Write block
FF3A	Verify flash
FF3B	Exit bootloader
FF3C	Update checksum

The command is followed by a status response from the bootloader. The code sent, 0x20, indicates that the bootloader successfully started. Other status responses are:

Code	Meaning
0x20	Bootload mode (Success)
0x01	Boot completed OK
0x02	Image verify error
0x04	Flash checksum error
0x08	Flash protection error
0x10	Comm checksum error
0x40	Invalid bootloader key
0x80	Invalid command error

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 64-byte block is broken up into two 32-byte packets. Each command requires a status response from the slave, so transmission of a 64-byte block is as follows:



Packet data (BULK OUT):

```
FF 39 00 01 02 03 04 05 06 07 00 4E 00 30 30 30
30 30 7E 30 30 7E 30 30 30 7E 30 30 30 7E 30 30
30 30 30 30 30 30 30 30 30 7E 30 30 30 28 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Status data (BULK IN):

```
20 20 03 00 00 4E 00 AA 00 00 20 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 30 30 30 30 7E 30 30 7E 30 30 30 7E 30 30 30
```

Packet data (BULK OUT):

```
FF 39 00 01 02 03 04 05 06 07 00 4E 01 7E 30 30
30 7E 30 30 30 7E 30 30 30 7E 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 DB 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

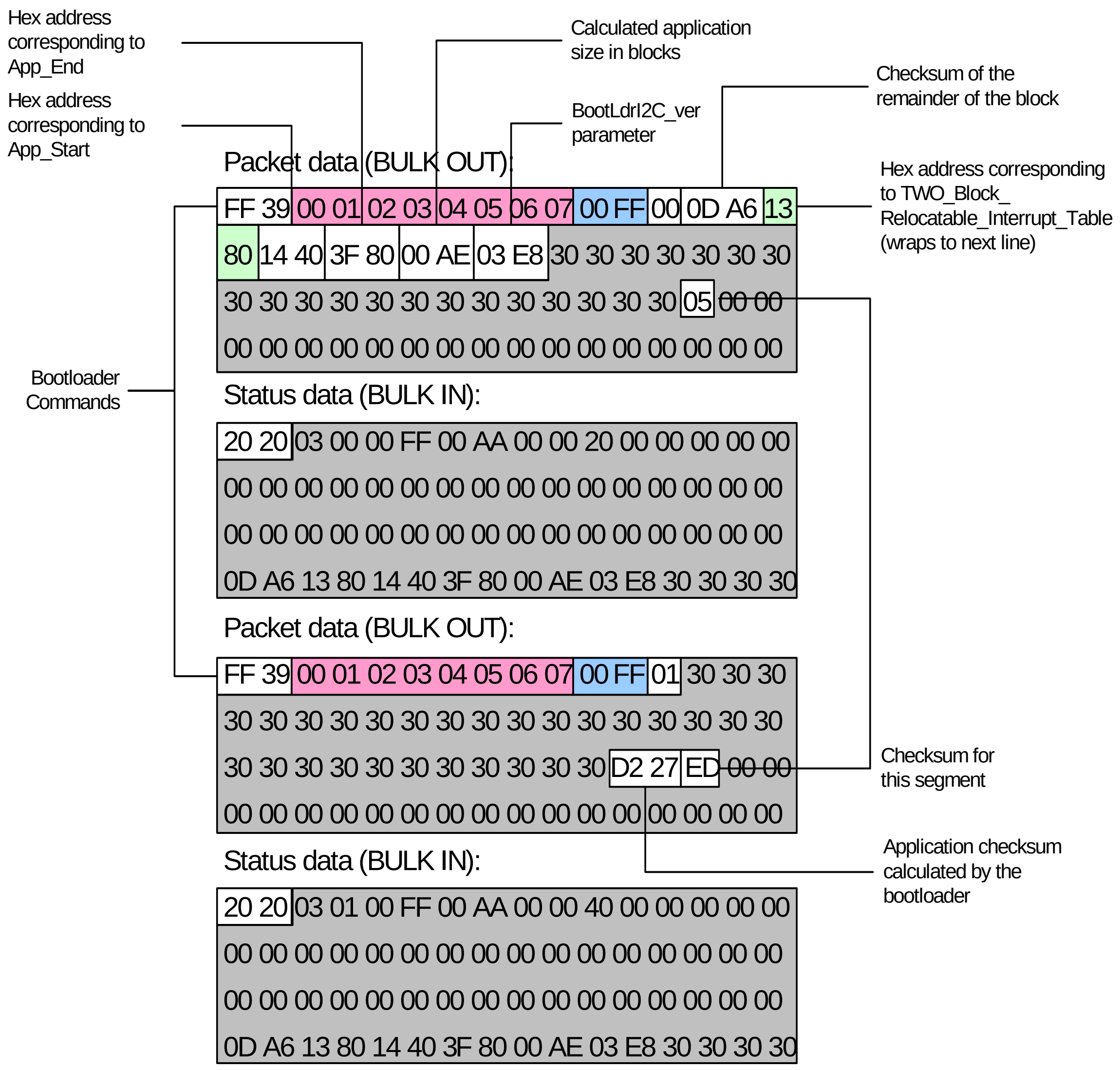
Status data (BULK IN):

```
20 20 03 01 00 4E 00 AA 00 00 40 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 30 30 30 30 7E 30 30 7E 30 30 30 7E 30 30 30
```

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 in two, 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 same 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 below:


Hex address corresponding to App_End
Hex address corresponding to App_Start
Calculated application size in blocks
BootLdrI2C_ver parameter
Checksum of the remainder of the block
Hex address corresponding to TWO_Block_ Relocatable_Interrupt_Table (wraps to next line)
Bootloader Commands
Checksum for this segment
Application checksum calculated by the bootloader

```
Packet data (BULK OUT):
FF 39 00 01 02 03 04 05 06 07 00 FF 00 0D A6 13
80 14 40 3F 80 00 AE 03 E8 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 05 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Status data (BULK IN):
20 20 03 00 00 FF 00 AA 00 00 20 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0D A6 13 80 14 40 3F 80 00 AE 03 E8 30 30 30 30
Packet data (BULK OUT):
FF 39 00 01 02 03 04 05 06 07 00 FF 01 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 D2 27 ED 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Status data (BULK IN):
20 20 03 01 00 FF 00 AA 00 00 40 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0D A6 13 80 14 40 3F 80 00 AE 03 E8 30 30 30 30
```

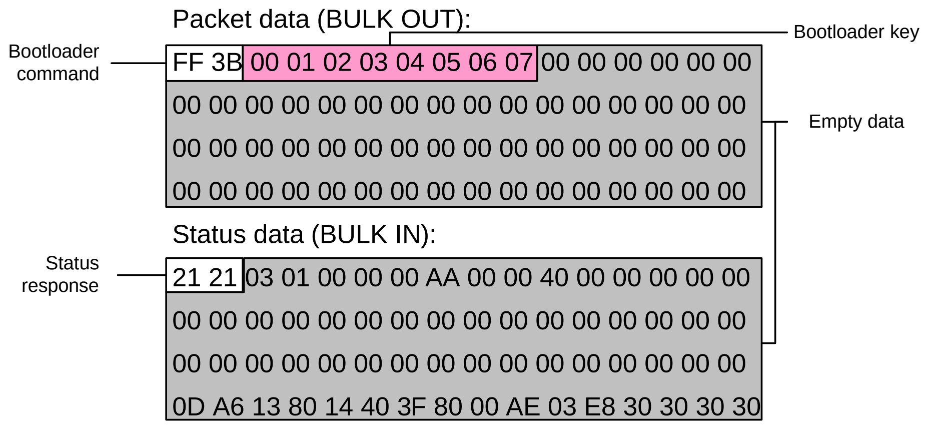
The last record contains the checksum block for this example (note the block number for this example is 0x00ff but the last block does not have to be 0x00ff).

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 Bootloader Exit command:



```
FF 3B 00 01 02 03 04 05 06 07 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
21 21 03 01 00 00 00 AA 00 00 40 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0D A6 13 80 14 40 3F 80 00 AE 03 E8 30 30 30 30
```

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

The last packet is a final status response.

[+] Feedback