I2C Bootloader Datasheet BootLdrI2C V 3.20

Copyright © 2008-2014 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks				API Memory (Bytes)		Pins
	CapSense®	I2C/SPI	Timer	Comparator	Flash	RAM	
CY7C604xx, CY7C643xx, CY8C20xx6/A/AN/AS/L/H, CY8C20xx7/S, CY8CTST200, CY8CTMG200, CY8CTMA300, CYONS2xxx, CYONSCN2xxx, CYONSFN2xxx, CYONSKN2xxx, CYONSTB2xxx, CYONSTN2xxx, CYRF89x35, CY8C20055/65, CY8C24x93, CY7C69xxx, CY8C20075							
Slave (full API support) 12 Bytes	–	1	–	–	3000	12 Bytes	2
Slave (no API support) 5 Bytes	–	1	–	–	2500	12 Bytes	2
SW-Only Slave (no API support) No I^2C resource usage	–	–	–	–	2400	9 Bytes	–

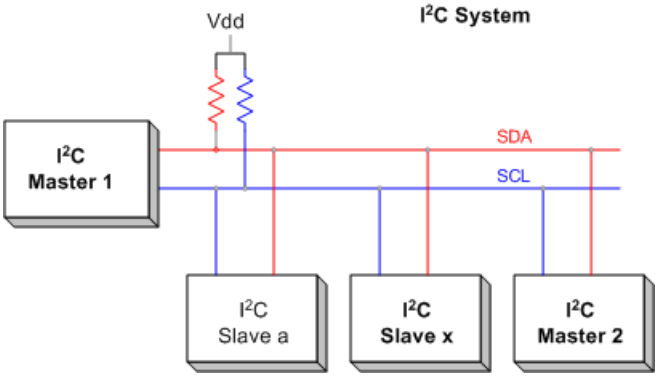
Features and Overview

- Industry standard Philips I^2C bus compatible interface.
- Enables you to reprogram a PSoC® device using the I^2C system bus instead of in-system programming pins.

The I^2C Bootloader User Module implements a bootloader that can reprogram the PSoC device over the I^2C interface. The PSoC device already gives an in-system serial programming interface (ISSP) that allows downloading new code into the device. However, the bootloader allows a code update to occur through an industry standard communication interface, such as I^2C. This user module can be useful for any device that has to be reprogrammed in the field. The bootloading information can be sent through an I^2C master device, such as a CY3240 (USB to I^2C bridge) or an in-system host processor.

The I^2C bootloader requires the I^2C Hardware User Module. It does not prevent the use of the I^2C bus for other functions within the PSoC device. The I^2C bootloader uses a separate I^2C address for its associated functions. All of the code for the I^2C bootloader is programmed in a protected area of EEPROM and cannot be accidentally overwritten.

Figure 1. I²C Block Diagram



Available Configurations

Three configurations are available. Two of the configurations use the available I²C resource, and this prevents adding a different user module in the application that also uses the I²C resource. The third configuration is for a Software-only Bootloader that uses no interrupts and does not consume the I²C resource. This offers more flexibility when you design an application. The operation architecture of the software-only user module has undergone other changes; these changes are summarized at the end of this user module datasheet.

Quick Start

1. Review this user module datasheet.
2. To operate properly, this user module alters memory locations and the way the overall project is handled by the compiler. It is important that you understand the information given in this user module datasheet before implementing a bootloader project.
3. Add the user module to a project.
4. Place the user module. Select the I²C for Bootloader Only or Full I²C API support with Bootloader.
5. In the menu bar, open the **Project > Settings** dialog box and click **OK** to save the project parameters.
6. Right-click on the user module icon and select **Boot Loader Tools**.
7. Click **Get Files**. The files *boot.tpl*, *custom.lkp*, and *flashsecurity.example* files are placed in the project root directory.
8. Close the **Boot Loader Tools** wizard.
9. Generate the source code and compile the project.
10. Review the output file *<project>.mp* and *<project>.hex* to see how the project has been built.
11. After creating a project that compiles without errors, go to the Sample Firmware Source Code section. Modify and adapt the sample code given in this section.
12. A detailed tutorial is available in PSoC Designer™ 5.1 and greater. To access the Bootloader tutorial, go to the menu bar, and click **Help > Documentation > Supporting Documents**.

Functional Description

The bootloader is located in a section of flash memory that you can define using user module parameters. This memory space is write-protected to prevent any accidental modification or corruption. The reset vector is modified so that when the processor is reset, the bootloader is executed.

The following operations are carried out by the bootloader:

1. Upon reset, the bootloader calculates a checksum for the flash user code and verifies it with a checksum that is written to the last two bytes of flash memory. If the checksum matches, the previous programming attempt has been successful and the bootloader branches to the beginning of the user code. The user code can execute.
2. If the checksum does not match, then the bootloader executes a customizable user code to perform system critical tasks (such as turning on a fan) and then enters the bootloader mode, where it waits for a 8-byte bootloader key from the master. If the previous bootloading has failed (for example, if there was a power transient), the program enters the bootloader mode due to a checksum mismatch.
3. Upon receiving a valid bootloader key from the master, the bootloader responds with a status byte informing the master that it is ready to receive the flash image.
4. The master sends the updated user code in variable length packets with some encoding bytes.
5. The bootloader writes the user code to the flash. When all the flash pages are written successfully, the bootloader performs a flash verify operation and then a software reset to start the user code.

   Note The I^2C master must wait 100 ms after each block write before reading the block status byte to allow the flash block write operation. While the device is writing to flash, it is unable to respond to interrupts. Devices with 128 byte flash blocks accumulate packets and verify the correct packet organization until a complete flash block is accumulated.

The bootloader portion of the user module gives a method to organize the memory map and major code functional blocks into areas that are compatible with device reprogramming. The memory organization of the project is considerably different from that of a conventional PSoC Designer™ project. Modifications to the memory map are necessary to meet the minimum device functionality requirements while the device application is being reprogrammed. Effectively, a project incorporating a bootloader contains two independent programs supporting different functions. Figure 2 on page 4 shows the Bootloader memory map.

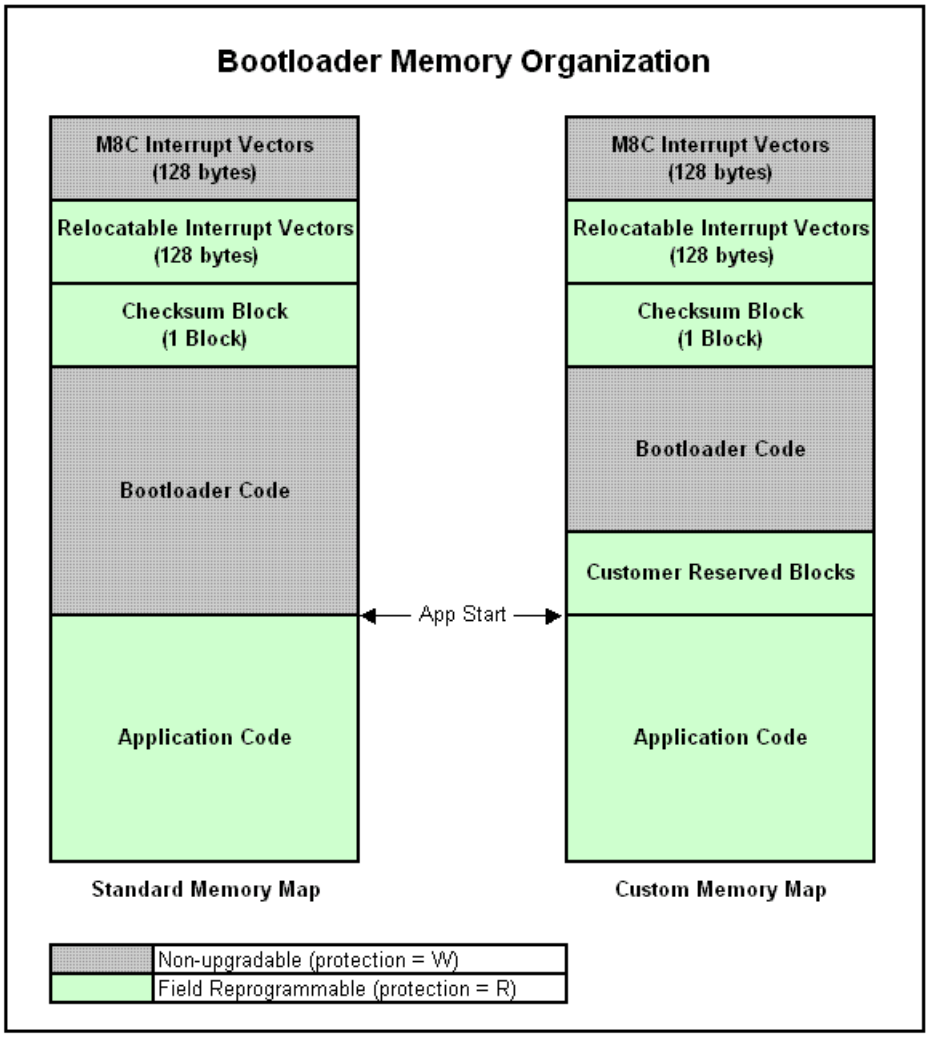
After a project incorporating a bootloader is deployed, the memory locations highlighted in gray are never reprogrammed. The memory locations highlighted in green can be altered by running the bootloader.

Theory of Operation

Creating a project with a bootloader requires several nonstandard modifications to the PSoC Designer standard model. To facilitate this, the BootLdrI2C User Module gives customized files and specialized tools to assist you in bootloader project development. The special tools are accessed by switching to the Device Editor view and right-clicking the BootLdrI2C User Module icon. In addition to the tools and files given as part of the user module, a host application example is also given in the user module installation to demonstrate the basic capability of the bootloader. This PC-based application and the source code for Microsoft Visual Studio® 2005 are available in a .zip file in the installation directory of PSoC Programmer 3:

[Install path]\Cypress\Programmer\3.xx\Bootloaders\BootLdrI2C\BootLoaderHostApp\…

Figure 2. Bootloader Memory Map



Overview

PSoC Designer uses 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 is maintained throughout the life of the design. A project without the requirements of a bootloader allows the compiler and linker to allocate RAM and ROM. However, a bootloader must group the RAM and ROM in specific areas so that the program does not crash while loading a new application.

The memory layout (see Figure 2) shows six key areas of ROM that are managed:

- The first memory area is blocks 0 and 1 of ROM. These blocks contain critical interrupt vectors and restart vectors. Because it is nearly impossible to control the read access to these blocks by any operating device, they are never erased and reprogrammed. The first two blocks of ROM must not be modified or placed in any other location.
- The second memory area is the relocatable interrupt table. This table may consist of one or two blocks, depending on the architecture of the device. This area contains interrupt and general purpose

vectors to give a jump table for interrupts or code entries that may be altered when a new application is loaded using the bootloader.

For example, this area contains the application start address. The bootloader can use this address to start the new application after the checksum has been validated at power up. This area is placed at blocks 2 and 3. After the application and bootloader are deployed, the contents in this area can be rewritten, but its location must not be modified. The characteristics of this area are similar to the checksum area described in the next section.

- The third area of the ROM defined is the checksum area. This area is placed at block 4 and it contains important data used by the bootloader software 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. 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 can be modified as long as the data used by the bootloader utility is not changed or repositioned within the block.
- The fourth area of memory to be defined is the area containing the bootloader code itself. This area starts at block 5. After the project or device containing the bootloader is deployed, this area is not reprogrammable and cannot be field upgraded.
- The fifth area is reserved for your data. This may contain configuration data that must persist through an upgrade using bootload. As shown in the memory map in Figure 2 on page 4, this area is optional. You can use a Standard Memory Map if you do not have any custom data.
- The sixth memory area is the application area. This area holds the application image. Because the code size of the "Bootloader Code" area is expandable, the starting address of this area can be adjusted. The "Application_Start_Block" parameter (in the Properties window) enables you to set the application starting address accordingly. This area must occupy all remaining memory.

If your application has code that must always be operational, even during a bootload process, the design of the BootLdrI2C User Module can allow sufficient customization to accommodate this. The best way to accomplish this is to add the code to the bootloader ROM area using the assembler AREA directive. Any RAM used by your code during the bootload process must be added to the RAM area defined for the bootloader.

Definition of Memory Areas in User Module Parameters

The parameters available in the BootLdrI2C User Module enable you to customize where major program elements are placed in ROM. The defaults in the user module should give a working initial setup. You must use these settings until a complete project is successfully compiled. After you have a compiled project, see 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 BootLdrI2C User Module gives a complete subsystem application that coexists with a foreground (primary) application. When the device is started or reset, the bootloader application is always invoked. After it is invoked at system startup, the bootloader validates the foreground application by calculating a checksum on the foreground application ROM area. The calculated checksum is compared to the one stored in the checksum block (that is created with the application). If the two checksums are equal, the bootloader validation function allows the foreground application to execute. If the two checksums are not equal, the bootloader enters a wait loop for a host application to download a valid application. It also enables its own I^2C subsystem to enable the host to transmit data. When the host system observes this

interface enabled, it may choose to execute its own set of applications. The user module tools automatically support two file formats for downloading an application to a target. The first format is an output file named <project_name>.txt and the second is an output file named <project_name>.dld. Each of these download file formats is supported by a different demonstration tool.

Download Methods

1. The .txt file can be downloaded using the CY3240 USB-I^{2}C bridge kit. The associated tool can be used to download the .txt file format as explained in AN45683. For more information on how to use the CY3240 USB-I^{2}C bridge, see the application note “PSoC1 Communication - I2C-USB Bridge Usage” AN2352. This download method is also discussed in the Appendix.

2. A second method of downloading an application is also supported. This method is more complicated than the previous method, but may give more information to develop a custom I2C download application for the finished product. A brief discussion of this host application is given here. An example application and source code is given in the installation directory of PSoC Programmer 3:

[Install path]\Cypress\Programmer\3.xx\Bootloaders\BootLdrI2C\BootLoaderHostApp\…

Two applications are necessary to demonstrate the I^{2}C bootloader. The first is a PSoC 27000-based application (a complete PSoC project is included) that is capable of translating RS-232 communication containing embedded bootloader download records to I^{2}C packets sent to the Bootloader application. Source code is given for this PSoC project and it should be easily adaptable to other PSoC device architectures.

The second application is a Microsoft Visual Studio application, I^{2}C Bootloader Host, given with an installer and source code for modification. It is capable of reading and parsing a download file <ProjectName>.dld and transmitting it to the PSoC project mentioned earlier. The source code is also given for this application for use with Microsoft Visual Studio 2005. This application is given for demonstration purposes only and is not intended for production use or resale.

Note In some cases, to use this application you must install the file mscomm32.ocx on your computer. You can download this file from the Microsoft website and install it using the Windows/ accessories/command prompt window and the following command:

```
> regsvr32 mscomm32.ocx
```

The file *regsvr32.exe* is in the Windows installation folder windows/system32 (winXP).

To download the file *mscomm32.ocx* from the Microsoft website, search for the filename "mscomm32.ocx".

Bootloader Tools

Several tools are available from the shortcut menu 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 the BootLdrI2C User Module is removed, the option to restore the default boot files is no longer available from the user module icon. It can be accessed from the tools tab in the main menu of PSoC Designer.

Generate Checksum – After your project builds correctly, you can use the bootloader tools to create and auto-validate checksums. When you enter the bootloader tools selection screen, the project code is generated and the entire project is compiled. 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 want. If build or compile

errors occur in the automated generate and build invoked by the Bootloader Tools and no hex file is successfully created, an error is reported but error debug information is not 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, you must use the conventional build and generate process external to the bootloader tools menu.

Generate dld file – This tool item derives a download file from the hex project output file. Three types of download files are generated: (<project_name>.txt, <project_name>.dld, and <project_name>.iic). The <project_name>.txt and <project_name>.dld are generated for I2C for Bootloader Only or Full I2C API Support option. The <project_name>.iic is generated for the Software-only Bootloader option. These files contain only the hex blocks that are reprogrammed by the bootloader, including the checksum block. The Host Demonstration application can read this file and download it to a working project incorporating a bootloader. Either of these download files can be deployed to a field application to upgrade a PSoC device.

The dld, txt, and iic download files are generated by the BootLdrI2C User Module tool and are listed in “Output Files” in the Workspace Explorer.

Note The <project_name>.txt file is now supported for the devices covered by this user module data-sheet only (unless specified in another datasheet).

Checksum Semiautomatic Generation

After your project is built and compiled without errors, the application checksum must be generated. To create the application checksum using one of the available utilities, right-click on the I^2C Bootloader User Module icon in the Device Editor view and select **Bootloader Tools.** An application checksum (previously calculated or default) is stored as a hidden user module parameter. When the Bootloader Tools menu page is invoked, any previous checksum is validated against the checksum calculated on the current output\<prj_name>.hex file. The checksum cannot be generated before a successful compile. After a checksum is created, it must be integrated into the compiled files. This requires a second compile.

Special Files Given

You can access several important files by selecting the **Bootloader Tools** menu and clicking **Get Files**. A device specific boot.tpl file is placed in the main project directory with a file called custom.lkp and a predefined flashsecurity.txt file. The original versions of these files are placed in the project backup directory. A brief description of each file follows:

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 following code blocks in the custom.lkp file:

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

During code generation, each of the last three lines of the custom.lkp file are rewritten under the control of the code generation software. Changes made within or below the last three lines either cause an error or are lost. You can change the rest of the custom.lkp file. To debug the memory allocation of the project, it is

possible to comment out all three lines mentioned earlier by inserting a semicolon in the first space. This allows the linker to place code automatically and may be helpful in determining the 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. To create the final project, you may have to manually modify this file according the final memory map and application size for the deployed device and firmware. Note that this is not the file that is directly used by PSoC Designer. If for some reason the project is updated or tagged for out of data files, it is inconvenient to have the flashsecurity file overwritten which would then require repeated modification by the developer. You can edit and rename the flashsecurity.example file as necessary.

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

I^2C Interrupt Processing

The standard BootLdrI2C User Module optionally gives a foreground copy of the I^2C interrupt processing module. This code is placed in upgradable memory with APIs to operate a fully featured I^2C slave. The bootloader itself maintains an internal utility that processes I^2C data addressed to the bootloader. This is to overcome the problem of executing code that is being rewritten (which is never a good programming practice).

Block Entry of Parameters

All memory parameters are entered in the bootloader in Blocks numbered from 0x00 through 0xFF (the maximum number of blocks varies depending on the device family from 0x4f to 0x1FF). Although this is not the most convenient format to enter memory addressees, it prevents the accidental assignment of partial block addresses to different sections of the memory map. Some PSoC devices are only capable of storing 64 byte flash blocks, while some use 128 byte blocks. Using blocks is a simple way to maintain the boundaries between the different sections of the project code correctly.

Most PSoC parts have 64 byte block sizes. The PSoC devices covered by this user module datasheet have 128 byte blocks. A couple of key facts are:

1. Any Bootloader needs to write to flash.

2. PSoC can only write to flash "Block" by "Block".

So for bootloader applications it is more useful to think of memory as a group of "Blocks" to be written.

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

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

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

DC and AC Electrical Characteristics

See the device datasheet of your PSoC device for the electrical characteristics of this device.

Placement

The user module consumes the I2C/SPI communication block. When placing the user module, you have the choice of **I2C for Bootloader Only** or **Full I2C API Support with Bootloader**. If you do not plan to use the I^2C interface for any purpose other than bootloading the device, choose **I2C for Bootloader Only**. The API is reduced to only those functions used by the bootloader, reducing both flash and RAM use.

Parameters and Resources

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

Figure 3. Default Parameters – Full API

Slave_Addr_HEX	0x0
Boot_Loader_Addr_HEX	0x0
Read_Buffer_Types	RAM ONLY
Communication_Service_Type	Interrupt
ApplicationCode_Start_Block	0x14
Bootload_when_CKSUM_fails	ENABLE_(deployment)
BootLoaderKey	0001020304050607
Flash_Program_Temperature_deg_C	-40C
Ignore_N_I2C_Prefix_Bytes	2
BootLdrI2C_ver	0x1000
I2C Clock	100K Standard
I2C_Pin	P[1]5-P[1]7

Figure 4. Default Parameters – I2C Bootloader only

Boot_Loader_Addr_HEX	0x0
Communication_Service_Type	Interrupt
ApplicationCode_Start_Block	0x14
Bootload_when_CKSUM_fails	ENABLE_(deployment)
BootLoaderKey	0001020304050607
Flash_Program_Temperature_deg_C	-40C
Ignore_N_I2C_Prefix_Bytes	2
BootLdrI2C_ver	0x1000
I2C Clock	100K Standard
I2C_Pin	P[1]5-P[1]7

Figure 5. Default Parameters – Software only Bootloader

Boot_Loader_Addr_HEX	0x0
I2C Clock	100kHz
I2C_Pin	P1[5]-P1[7]
Interrupt_out	Disable
Interrupt_out_Port	None
Interrupt_out_Pin	None
ApplicationCode_Start_Block	0x14
Bootload_when_CKSUM_fails	ENABLE_(deployment)
Run_App_if_CKSUM_OK	DISABLE
BootLoaderKey	0001020304050607
Flash_Program_Temperature_deg_C	-20C
Ignore_N_I2C_Prefix_Bytes	2
BootLdrI2C_ver	0x1000

Default parameters are shown here as an example. These parameters may be periodically updated in the source code of the user module and may differ from this example.

All buffer names are written to describe their use by an I²C master. For example, the I2Cs_pRead_Buf refers to the location in RAM containing data to be read by the I²C master.

Slave_Addr_HEX

This is a Slave and MultiMasterSlave parameter. It selects the 7-bit slave address that is used by the I²C master to address the slave or the MultiMasterSlave when it is in slave mode. Valid selections are from 0x00-0x7F. Because this is the upper 7 bits of the address, the actual address appears to be doubled inside the code.

Boot_Loader_Addr_HEX

Selects the 7-bit slave address that is used by the I2C master to address the I2C BootLoader slave device. Valid selections are from 0 - 7Fh. Because this is the upper 7 bits of the address, the actual address appears to be doubled inside the code. The parameter value must differ from the Slave_Addr_HEX parameter value.

Read_Buffer_Types

Selects what types of buffers are supported for data reads. Two selections are available: RAM ONLY or RAM OR FLASH. Selection of RAM ONLY removes code and variables required to support direct flash-ROM reads. Select RAM OR FLASH to give code and variable support for reading either RAM buffers or flash-ROM buffers for data to be transmitted to the master. Select RAM or FLASH to enable support for API calls to select a RAM- or flash-read buffer for use.

Interrupt_out/Interrupt_out_Port/Interrupt_out_Pin

Note Only available on Software-only Bootloader.

A configurable output pulse is enabled while the bootloader is waiting for a "enter-bootloader" or "run-app" command. After the bootloader is entered, the pulse is Low during the time that the flash is being written after each block has been transferred. The duty-cycle and the period of the pulse are configurable through the include file. See the description in this user module datasheet that discusses the SW-only Bootloader.

Communication_Service_Type

Note Not available in Software-only Bootloader.

This parameter allows you to select between an interrupt based data processing strategy and a polled strategy. In the interrupt based strategy a transfer is initiated against a predefined buffer. Data is then moved in or out of the buffer as quickly as possible in the background. An interrupt service routine (ISR) routine is included which handles data movement. When you select the polled data processing strategy, you are in control of when data movement takes place. To implement a polled strategy you must periodically call the function BootLdrI2C_Poll() (see the I2C.h files for the exact instance name). Each time the polling function is called a single byte is transferred. Other I²C functions are used identically. The polled communication strategy may be used in a situation where interrupt latency is critically important (and asynchronous communication interrupts may cause problems). Another use is when you desire absolute control of when data is transferred. A drawback of polling is that when the I²C state machine is enabled the bus is stalled automatically after each byte until the polling function is called. The polling function is available only for the Slave and MultiMasterSlave implementations of I²C. The Single Master implementation offers API functions to support byte-wise data transfers.

I2C Clock

Specifies the desired clock speed at which to run the I²C interface. There are three clock rates available:

- 50K Standard
- 100K Standard
- 400K Fast (when CPU_Clk_speed is greater than 6 MHz)

When an invalid checksum is detected at power up the default clock speed for the bootloader is 100 kHz. This can be customized by the developer by making a permanent modification to the bootloader asm (search for the tag "UserCode_BODY6").

I2C_Pin

Selects the pins from Port 1 to be used for I^2C signals. There is no need to select the proper drive mode for these pins, PSoC Designer does this automatically.

ApplicationCode_Start_Block

This is the first block of code assigned to the User Application. This code must be bootloadable/writeable. 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 shown in Figure 3 on page 10.

Bootloader_when_CKSUM_fails

For development purposes it may be convenient to disable the checksum feature while the application is being debugged. This parameter can be used to disable the feature that automatically enters the bootloader after power up if the application checksum does not match that stored in the checksum block.

The default value is Enable_(Deployment).

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 invoked.

The default value "0001020304050607".

Flash_Program_Temperature_Deg_C

This is the typical programming temperature expected when the device is reprogrammed. Programming the device at 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.

Run_App_if_CKSUM_OK

Note Available in the Software-only Bootloader.

This parameter allows the application to run automatically if the checksum is validated. When this feature is enabled, the only way to enter the bootloader is through the BootLdrI2C_bootLoaderStart() function from the application. See the associated flow chart of Software-only bootloader.

Ignore_N_I2C_Prefix_Bytes

Note The RS-232 to I²C translator project sends 2 prefix bytes to the device. Therefore the correct setting for this parameter when using the given demonstration applications is 2. This is also used for the case of certain SM-bus protocols based on I²C. This parameter allows you to configure the Bootloader to ignore a variable number of prefix bytes.

When used with the Software-only bootloader, this parameter takes on special meaning. The SW only bootloader uses two of these byte as protocol bytes itself. This means that host protocol bytes should be added to the two bytes used by the SW only bootloader. The SW only bootloader allows not LESS than 2 bytes of prefix.

BootLdrI2C_ver

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

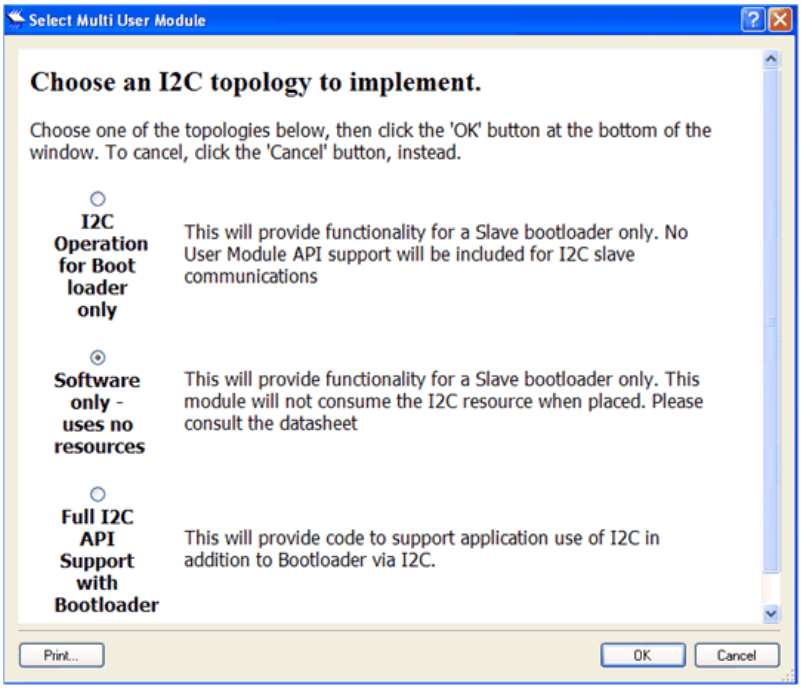
I²C Topology Selection Options

When placing a Bootloader User Module, you must decide which I²C topology to implement for the bootloader project.

I²C operation for Bootloader only – this option gives I²C communication for bootloader only. No user module API support will be included for I2C slave communications. Select this option if your application does not use I²C communication for anything else besides bootloading.

Software-only (uses no resources) – this option also gives I²C communication for bootloader only. No user module API support will be included for I²C slave communication. Unlike the first option, this option will not consume the I²C resource. Select this option if you plan to use I²C communication for bootloading only and I²C resource for other purposes.

Full I²C API Support with Bootloader – this option gives code to support application use of I²C in addition to Bootloader through I²C. Select this option if you plan to use I²C communication in your application for other purposes besides bootloading.



Common Problems

Updating Bootloader Projects, Service Pack Upgrades, and Compilers

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

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

Internal Use of the Watchdog Timer

Coordination with the watchdog timer (WDT) is linked to the global parameter: WATCHDOG_ENABLE in the file globalparams.inc. If the project uses a watchdog timer, conditionally compiled code linked to the global parameter automatically sets the watchdog during bootload checksum and download operations. The CPU clock speed affects how fast the watchdog timer is updated. A practical minimum setting for the watchdog timer is about 0.125s.

Improper Settings in flashsecurity.txt

The default settings for this file are set when the project is created. An example configuration is given in the file “flashsecurity.example”. flashsecurity.example is given with the ‘BootLoader Tools > Get Files’ user module menu item. The map must allow flash write at all the locations that are eventually bootloaded. One strategy is to make all blocks writable. Another strategy is 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 is quicker than debugging it later. It is your responsibility to write protect the areas of code used by the bootloader executable. Failure to correctly map flash security can be a contributing factor in a broken system and an extremely difficult debug task.

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

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

Since the memory map for a bootloader project is considerably different than that of a conventional project, the relocatable code start address almost certainly needs to be altered. This is a common source of the errors generated by the linker when it attempts to link more than one block code at the same address. This parameter is found under the main menu tab, **Project > Settings > Linker**. Calculate the absolute hex start address to be a little bigger than the highest block used by the bootloader code, or to

occupy an unused area of ROM. For the I²C version of the bootloader, setting this value to 0x8C0 or 0x900 should be adequate (if the default values for the other parameters are used).

Note When unplacing the Bootloader UM, the Relocatable Code Start Address does not reset to its original value. The user needs to change it back manually to save ROM space.

Memory Overlap

To correct the relocatable code start address, use a leading semicolon 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, because they prevent output files from being generated. Modifying the custom.lkp file may allow the linker to place object blocks, which then give a starting point for correcting the memory overlap root cause.

Power Stability

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

Application is Not Completely Stopped during Bootload Process

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

Using the I²C interface allows multiple addresses to be recognized. It is possible to begin talking to the bootloader while the application is incompletely turned off. A method to explicitly enter the bootloader should be implemented that includes a complete shutdown of the running application.

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 never returns and never enters the bootloader. As a result, it cannot be reprogrammed after it begins executing (as long as it has a correct checksum). There are multiple strategies to address this problem. 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 exit to the foreground application when the timeout expires.
2. Set a test at some point in the code that causes the device to enter the bootloader. This could be switch closure or holding a port pin low or high.
3. Enable the I²C application resource and create an I²C command that causes the device to enter the bootloader. Generally, if I²C is enabled by the main routine, the bootloader address can be used to cause the device to enter the bootloader
4. Use the WDT to reset the device if it is not serviced regularly. This could be combined with one of the previous strategies to allow a WDT interrupt to initiate a bootloadable state. Upon reset from a WDT

reset condition, it is possible to monitor a status bit associated with the WDT to detect that this is the cause of the reset condition. Refer to the Technical Reference Manual.

5. Two projects have been developed and the bootloader in each is different in some subtle way. Remember that bootloading implies that programming *part* of a device is taking place. This implies that the implementation of the bootloader for each of two mutually reprogrammable applications must be identical. All bootloader parameters should be identical, relocatable code start addresses should also be identical (this is different from the first application block). Debug strategies for this problem include comparison of the two hex files in question paying particular attention to the areas of hex code used by the bootloader. Another method is to compare the <project>.lst files. The bootloader makes use of some redirect vectors to allow certain application address parameters to change. All of these jump vectors must match for an application and a bootloader. After a bootloader is deployed to a field application, there is no way to alter the code within it. A future application must still ‘agree’ about where mutually used jump vectors are stored.
6. The PSoC based translator app hangs when a bootload operation fails. There is an if-def based timeout that may be configured to allow the PSoC base translator to drop a communication attempt after a variable software loop. For debugging purposes, you may wish to turn this software switch on or off. Examine the source code of the project for the timeout switch.
7. Power noise, glitching, brownout, and slow power ramp poor connections. All of these power problems can cause difficult to diagnose problems with flash programming. Program execution is rapid with respect to power ramps and in some cases a part may still have power levels changing when flash programming is taking place. One example is a status write to flash at power up. You must evaluate your use model and the potential for changing power supply conditions during flash operations. Poor power stability may contribute to non functional parts and may appear to be the result of poor flash retention.

Application Programming Interface

The Application Programming Interface (API) firmware gives high level commands that support sending and receiving multi-byte transfers. Read buffers may be set up in RAM or flash memory. Write buffers may only be set up in RAM.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns BootLdrI2C_1 to the first instance of this user module in a given project. It can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of every global function name, variable and constant symbol. In the following descriptions the instance name has been shortened to BootLdrI2C for simplicity.

Note

In this, as in all user module APIs, the values of the A and X register may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This "registers are volatile" policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. 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 functions may leave A and X unchanged, there is no guarantee they may do so in the future.

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

ENTER_BOOTLOADER

Description:

Routine to completely setup the bootloader and prepare to download a new application program. Once called this routine does not return unless a timeout or reset occurs.

The GenericBootloaderEntry function name is always located at the same physical ROM address so an application compiled at a later date can still use this function call to enter the bootloader.

C Prototype:

```
void ENTER_BOOTLOADER(void);
```

Assembler:

```
lcall ENTER_BOOTLOADER
```

Parameters:

None

Return Value:

None

Side Effects:

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

BootLdrI2C_Start

Description:

Empty routine included for compatibility with other Cypress user module APIs.

C Prototype:

```
void  BootLdrI2C_Start(void);
```

Assembler:

```
lcall BootLdrI2C_Start
```

Parameters:

None

Return Value:

None

Side Effects:

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

BootLdrI2C_DisableInt

Description:

Disables the I^2C slave by disabling the serial DATA (SDA) interrupt. Performs the same action as I2Cs_Stop.

C Prototype:

```
void  BootLdrI2C_DisableInt(void);
```

Assembler:

```
lcall BootLdrI2C_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

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

BootLdrI2C_EnableInt

Description:

Enables I^2C interrupt allowing start condition detection. Remember to call the global interrupt enable function by using the macro: M8C_EnableGInt.

C Prototype:

```
void  BootLdrI2C_EnableInt(void);
```

Assembler:

```
lcall BootLdrI2C_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

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

BootLdrI2C_Poll() and BootLdrI2C_BootLdr_Poll()

Description:

Used when the Communication_ Service_Type parameter is set to Polled. This function gives a user controlled entry into the I/O processing routine. If Communication_Service_Type parameter is set to Interrupt, the function does nothing.

C Prototype:

```
void BootLdrI2C_Poll(void);
void BootLdrI2C_BootLdr_Poll(void);
```

Assembler:

```
lcall BootLdrI2C_Poll
lcall BootLdrI2C_BootLdr_Poll
```

Parameters:

None

Return Value:

None

Side Effects:

One I^2C event is processed each time this routine is called and status variables are updated. An event constitutes either an error condition, an I/O byte, or in certain cases, a stop condition. There are three possible results from calling this routine:

1. No action if no data was available.
2. Reception or transmission of an address or data byte if one was available.
3. Processing of a stop ‘event’ when an external master has completed its write operation. When a stop state is processed at the end of a write operation, only status variables are updated. If an I^2C byte is pending when a stop state is processed, the I2CHW_Poll function must be called again to process it.
   The I2CHW_Poll() function has no effect if Communication_Service_Type is set to Interrupt. When a start/restart condition and an address is detected on the bus the bus is stalled until the I2CHW_Poll() function is called. If the address is NAKed, subsequent bytes transferred for that transaction are ignored until another start/restart and address is detected, otherwise the I2C bus is stalled for each data byte until the I2CHW_Poll() function is called.
   The I2C data is stalled by the I2C hardware until this function is called if the Communication_Service_Type is set to Polled. For received data the bus is stalled at the end of the byte and before an ACK/NAK is generated by holding the SCL (clock) line low. For transmitted data the bus is stalled immediately after the ACK/NAK bit is generated externally.
   These two functions are compatible with the following restrictions. If the Bootloader is active and may be entered by an external application I2C command, the API BootLdrI2C_BootLdr_Poll() should be used. If an I2C address that is not the Bootloader address is detected this address is tested and passed on to the foreground I2C interrupt process, which also tests the address. If the Bootloader is inactive, use of the BootLdrI2C_Poll() API does not give the I2C address to the internal bootloader data process routine. The address and subsequent data are instead passed directly to the foreground routine

BootLdrI2C_Stop

Description:

Disables the I2CHW by disabling the I^2C interrupt.

C Prototype:

```
void  BootLdrI2C_Stop(void);
```

Assembler:

```
lcall BootLdrI2C_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

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

BootLdrI2C_EnableSlave

Description:

Enable the I²C Slave function for the I²C HW block by setting the Enable Slave bit in the I2C_CFG register.

C Prototype:

```
void  BootLdrI2C_EnableSlave(void);
```

Assembler:

```
lcall BootLdrI2C_EnableSlave
```

Parameters:

None

Return Value:

None

Side Effects:

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

BootLdrI2C_DisableSlave

Description:

Disables the I²C Slave function by clearing the Enable Slave bit in the I2C_CFG register.

C Prototype:

```
void  BootLdrI2C_DisableSlave(void);
```

Assembler:

```
lcall BootLdrI2C_DisableSlave
```

Parameters:

None

Return Value:

None

Side Effects:

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

BootLdrI2C_InitWrite

Description:

API available only for FullAPISupport.

The BootLdrI2C_InitWrite routine initializes a data buffer pointer for the slave to use to deposit data, and zeroes the value of a count byte for the same buffer.

C Prototype:

```
void  BootLdrI2C_InitWrite(BYTE * pBootLdrI2C_WriteBuf, BYTE  BootLdrI2C_Write_Count);
```

Assembler:

```
mov A, Write_Count
push A
move A, >pWriteBuf
push A
mov A, <pWriteBuf
push A
lcall BootLdrI2C_InitWrite
```

Parameters:

pWriteBuf: A pointer to a RAM buffer location. Write_Count: The length of the write buffer.

Return Value:

None

Side Effects:

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

BootLdrI2C_InitRamRead

Description:

API available only for FullAPISupport.

The BootLdrI2C_InitRamRead routine initializes a data buffer pointer for the slave to use to retrieve data from and zeroes the value of a count byte for the same buffer.

C Prototype:

```
void BootLdrI2C_InitRamRead(BYTE * pBootLdrI2C_ReadBuf, BYTE  BootLdrI2C_Read_Count);
```

Assembler:

```
mov A, Read_Count
push A
move A, >pReadBuf
push A
mov A, <pReadBuf
push A
lcall BootLdrI2C_InitRamRead
```

Parameters:

pReadBuf: A pointer to a RAM buffer location. Read_Count: The length of the read buffer.

Return Value:

None

Side Effects:

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

BootLdrI2C_InitFlashRead

Description:

API available only for FullAPISupport

The BootLdrI2C_InitFlashRead routine initializes a flash data buffer pointer for the slave to use to retrieve data from, and zeroes the value of a count byte for the same buffer.

C Prototype:

```
void  BootLdrI2C_InitFlashRead(const BYTE * pBootLdrI2C_flashaddr,  unsigned int
BootLdrI2C_Read_CountHI);
```

Assembler:

```
mov A, >Read_Count
push A
mov A, <Read_Count
push A
move A, >pflashaddr
push A
mov A, <pflashaddr
push A
lcall BootLdrI2C_InitFlashRead
```

Parameters:

pflashaddr: A pointer to a flash data buffer location. Read_Count: The length of the read buffer.

Return Value:

None

Side Effects:

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

Read status bits are cleared.

BootLdrI2C_bReadI2CStatus

Description:

API is available only for FullAPISupport.

Returns the value in the I2CStatus variable.

C Prototype:

```
BYTE  BootLdrI2C_bReadI2CStatus(void);
```

Assembler:

```
lcall BootLdrI2C_bReadI2CStatus ; Accumulator contains the status on return
```

Parameters:

None

Return Value:

bI2CStatus - Status data

Constant	Value	Description
I2CHW_RD_NOERR	01h	Data read by the master, normal ISR exit
I2CHW_RD_ OVERFLOW	02h	More data bytes were read by the master than were available
I2CHW_RD_ COMPLETE	04h	A read was initiated and is complete.
I2CHW_READFLASH	08h	The next read is from a flash location

Constant	Value	Description
I2CHW_WR_NOERR	10h	Data was written successfully by the master
I2CHW_WR_OVERFLOW	20h	The master wrote too many bytes for the write buffer
I2CHW_WR_COMPLETE, I2CHW_ISR_NEW_ADDR	40h	A master write was completed by a new address or stop
I2CHW_ISR_ACTIVE	80h	The I^2C ISR has not yet exited and is active

Side Effects:

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

BootLdrI2C_ClrRdStatus

Description:

API available only for FullAPISupport

Clears the status bits in the Control/Status register but does not alter buffer addresses, counts, or the flash/Ram Read bit.

C Prototype:

```
void  BootLdrI2C_ClrRdStatus(void);
```

Assembler:

```
lcall BootLdrI2C_ClrRdStatus
```

Parameters:

None.

Return Value:

None

Side Effects:

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

BootLdrI2C_ClrWrStatus

Description:

API available only for FullAPISupport

Clears the status bits in the Control/Status register but does not alter buffer addresses, counts, or the flash/Ram Read bit.

C Prototype:

```
void  BootLdrI2C_ClrWrStatus(void);
```

Assembler:

```
lcall BootLdrI2C_ClrWrStatus
```

Parameters:

None.

Return Value:

None

Side Effects:

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

Software-only Bootloader (uses no I²C resources or interrupts)

This new version of the I²C bootloader is based entirely on software. It does not use interrupts while it operates, and it does not consume the I²C resource when it is placed. This frees the resource for use within the application.

The use model of the Software-only Bootloader should be considered as: the bootloader exists as a separate application that runs at reset and allows loading a new application program. When the application is running, the bootloader is almost completely invisible.

Features Supported by the Software-only Bootloader

Communication format matches the EZI2C protocol: For a WRITE transaction

<I2C_WRaddr>, <EZI2CSubaddr>, <data written starting at EZI2CSubaddr>,<stop condition>

For a READ transaction

<I2C_RDaddr>, <data read starting at EZI2CSubaddr specified in previous write>,<stop condition>

No I²C resource is used by the bootloader, which allows the application to use I²C more easily.

Optional ability to run the application automatically if the checksum is OK.

Optional ability to erase the contents of the application when the bootloader bootload operation is started.

Optional ability to speed the checksum process. After the checksum is verified, a value is saved in flash indicating that it does not need to be run on each startup. This allows startup to take place much faster.

A configurable output pulse is enabled while the bootloader is waiting for a command. After the bootloader is entered, the pulse is High when the flash is being written after each block is transferred. This pulse is duty-cycle, and period configurable.

A hardware I²C buffer is used to eliminate software latency when acknowledging the I²C address and Data.

At initial boot, a status response is preloaded for initial communication with the host.

Restrictions (Software-only Bootloader)

Multiple instances of the Software-only Bootloader are not supported. The Software-only Bootloader User Module is not guaranteed to work properly when there are multiple instances.

After writing any data to the HW I²C buffer when the bootloader is active, the host must wait to allow the HW buffer to be read before requesting status (approximately 10 ms).

The Software-only Bootloader offers only one API function.

Initial STATUS (16 bytes)

When the bootloader is entered, a status response is preloaded for the host to query. This status consists of 16 bytes. The response may be tailored by the end user by modifying values in the bootloader.inc file.

The status can be obtained by the first writing a subaddress of 0 to the bootloader address, then reading the data from the I2C buffer.

These values are initialized in the following table:

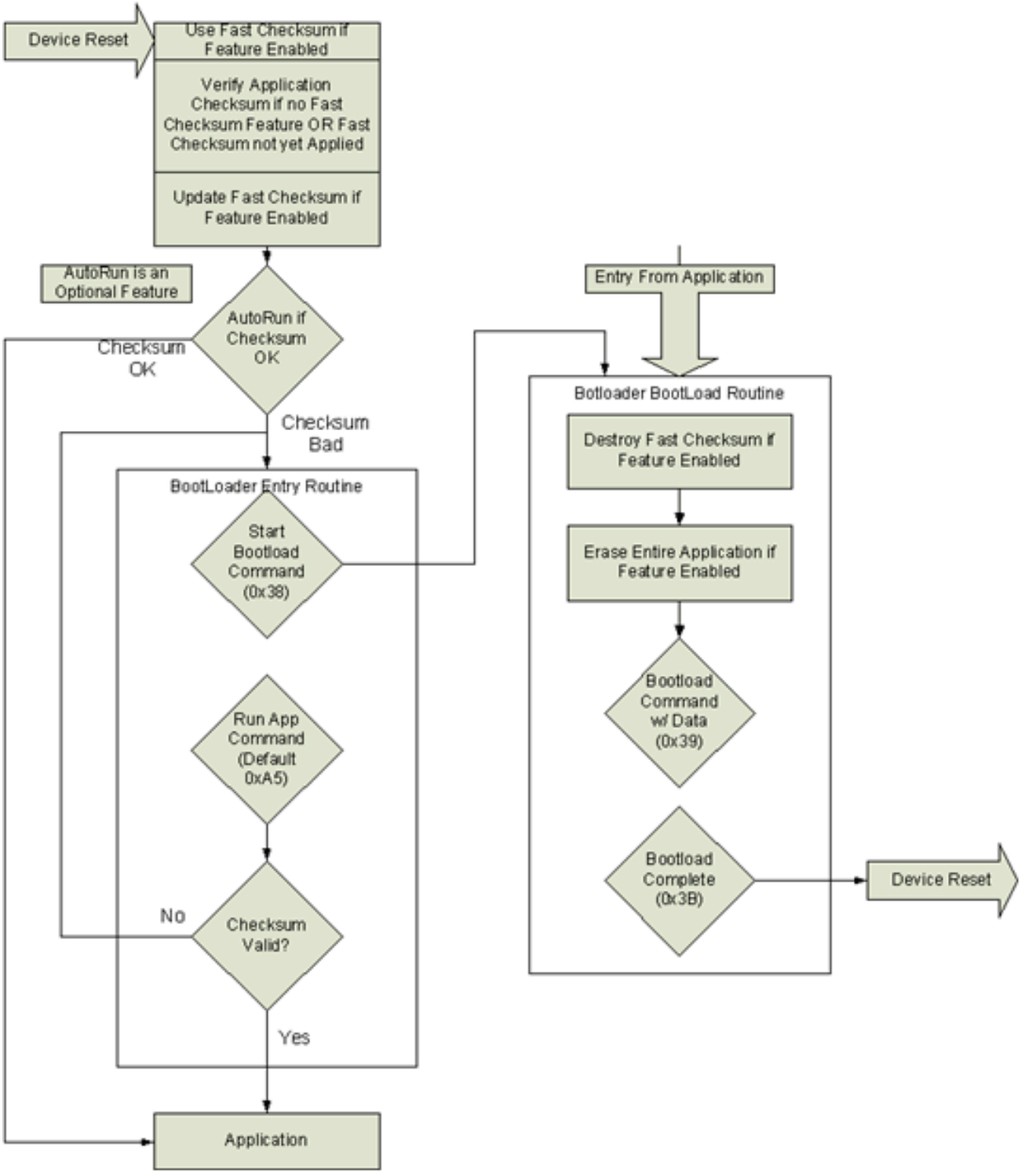
Address	Name / Bit Fields Description	Value	Comment	Master/PSoC Access
0	**HST_MODE** Bit 7-0 – Always 0	0x00	Defined later by host	W/R
1	**BootLoaderStatus** Bit 7- 5 - Always 0 Bit4- bootloader has entered Bit 3-2 - Always 0 Bit1- Watchdog reset has occurred Bit0 - Flash Checksum is valid	0x10 or 0x11 or 0x13	Bits 0-1 are used if the bootloader is loaded on startup; else, these bits are cleared	R/W
2	**bootErrorCode** Bit7-6- Always 0 Bit5- bootloader has entered Bit4-2 Always 0 Bit1- Flash checksum Error Bit0- Always 0	0x00 or 0x02 or 0x20	Bit 1 is used if bootloader is loaded on startup; else, the byte is cleared	R/W
3	**BL_rev_BIH** - Bootloader Version Defined by Bootloader (MSB)	0x10	MSB of "BootLdrI2C_ver" UM parameter	R/W
4	**BL_rev_BIL** - Bootloader Version Defined by Bootloader (LSB)	0x00	LSB of "BootLdrI2C_ver" UM parameter	R/W

Address	Name / Bit Fields Description	Value	Comment	Master/PSoC Access
5	**BL_rev_AppH** - Bootloader Version Defined by Application (MSB)	0xFF	Can be overridden by the user in the "bootloader.inc" file	R/W
6	**BL_rev_AppL** - Bootloader Version Defined by Application (LSB)	0xFF		R/W
7	**Design_IP_revH** - Design IP Version (MSB)	0xFF		R/W
8	**Design_IP_revL** - Design IP Version (LSB)	0xFF		R/W
9	**APP_IdH** - Application ID (MSB)	0xFF		R/W
A	**APP_IdL** - Application ID (LSB)	0xFF		R/W
B	**App_verH** - Application Version (MSB)	0xFF		R/W
C	**App_verL** - Application Version (LSB)	0xFF		R/W
D	**Custom_ID0**	0xC1		R/W
E	**Custom_ID1**	0xC2		R/W
F	**Custom_ID2**	0xC3		R/W

An end-user can modify the values of "BL_rev_App*", "Design_IP_rev*", "APP_Id*", and "App_ver*" "Custom_ID*" in the "bootloader.inc" file. However, in the "bootloader.inc" file, if you change the "BOOT_LOADER_DEFAULTS" to 0x00, you can set these values by replacing the "TO_DO_Developer" variables.

Software-only Bootloader Operation

Figure 6. Software-only Bootloader Operation Flow Chart



Customizing the Software-only Bootloader

PSoC Designer allows protecting blocks of code from update by the code generator. The entire bootloader is protected. For this reason, you can update the bootloader code and your changes are not overwritten by the code generator. This approach is an imperfect compromise to attempt to mitigate past problems.

1. A deployed bootloader must not be upgraded with a new version. It may not be possible to mix versions of bootloader in the field.
2. You may choose to fix a bug, and add an enhancement or other customization. Preserve this code wherever it is in the code.

Include File Configurations

Some configuration changes are made by changing permanent EQU values inside the file UM_Name_bootloader.inc.

The Bootloader.inc file can be used to configure some operations of the Software-only Bootloader. There is a protected area where you can make configuration changes that are not altered when code is regenerated.

Bootloader RUN_APP

The value of this command is set to a default of 0xA5 in the booloader.inc file. If needed, you can modify the value of the command by altering the definition in the include file.

PULSE_COUNT: EQU 0xA00 (Default)empirical adjustment for the pulse period for the "I'm alive" interrupt out pulse (legal values, 0-65535).

PULSE_COMPARE:EQU 0x10 (Default)empirical adjustment for the pulse width for the "I'm alive" interrupt out pulse (legal values 0-255).

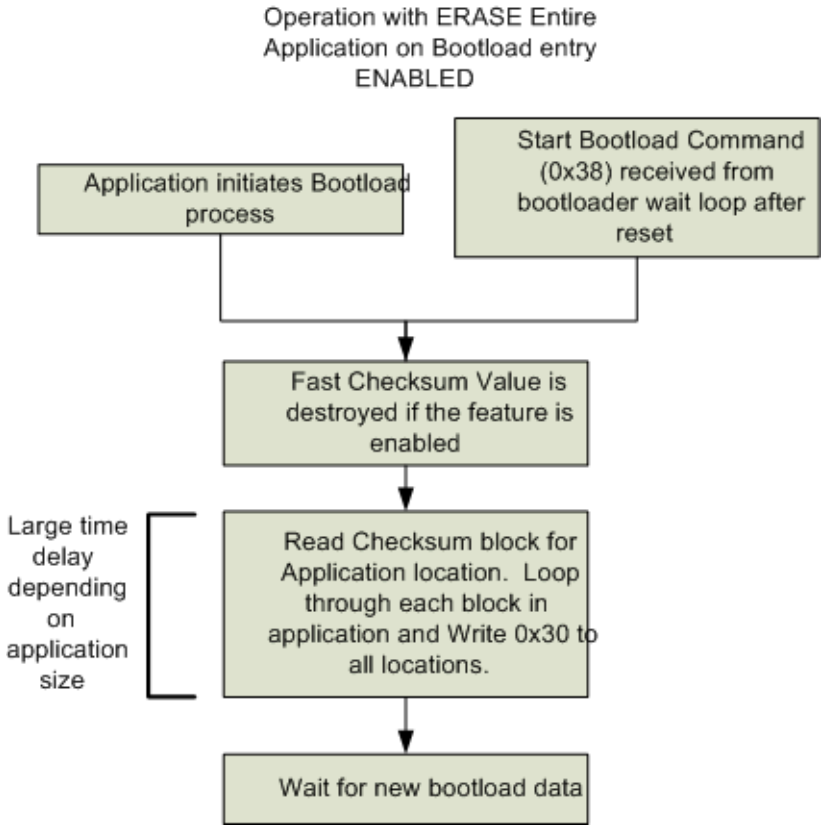
The pulse is driven through a software timing loop. Therefore, for the previous two values the pulse is on 0x10 loops of 0xA00 loops (32/320).

BOOT_TIMEOUT:EQU 00 (default 0x40) At certain points in the bootloader loop, a counter is initialized and decremented to prevent infinite loops.

If the counter decrements to zero the loop is exited. Setting this value to zero prevents the loop from exiting. This is useful for debugging and stepping through the bootload process.

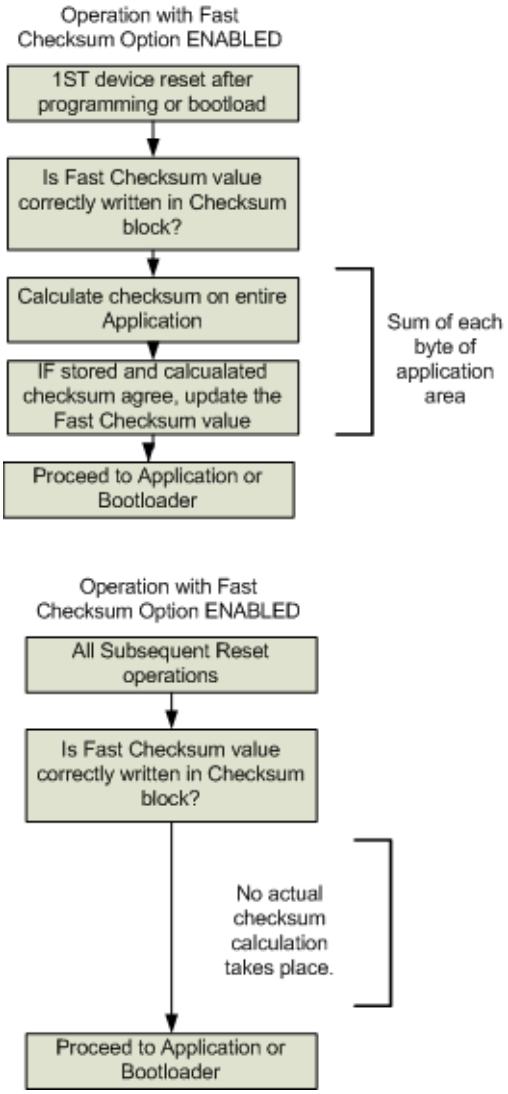
ERASE_APP_BEFORE_BOOTLOAD: EQU 0 (default 0/off) When the start bootload command is received, the entire application is erased. This involves entering a loop to first erase the block, then initialize each block defined as application space with the value 0x30 (halt). This operation may take several seconds. During the period that the application is being erased, the device may not always respond to I^2C traffic. The I2C_Interrupt_out pin is asserted when the device is writing to flash. Using this feature with the USB-I2C bridge is especially difficult, because the bridge is ONLY capable of delaying for fixed periods of time. The time that might be spent on an application erase is highly variable. The best development approach may be to time the erase period for the specific application size and customize the delay used to bootload the device. If an intelligent host is used to transfer bootload data, it is possible to wait for a pin toggle before continuing with the process.

Figure 7. Erase Application on Bootload



CHECKSUM_FAST_VERIFICATION: EQU 1 (Default 1/on) verifying the checksum takes approximately 1.3 seconds with a 3 MHz clock pulse, and approximately 30K of ROM space to checksum. When this feature is enabled after the first checksum verification, a value is updated in the checksum block that indicates that the checksum has passed. Subsequent checksum verifications only look to see if this value has been set. This operation reduces the verification time to about 260 uSec at a 3 MHz instruction clock rate.

Figure 8. Fast Checksum Option



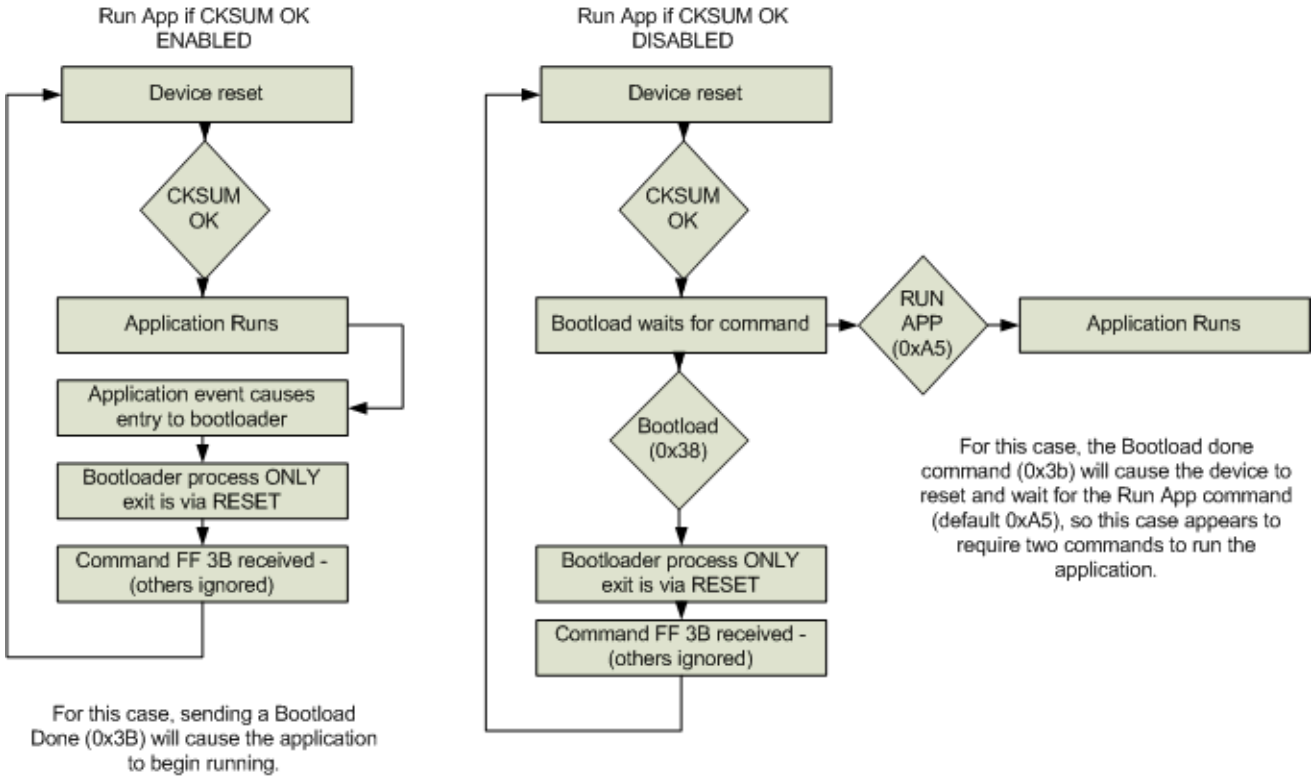
Run_App_if_CKSUM_OK: EQU 1 (Default 1/on)

Enabling this feature allows running the application automatically on power up, if the checksum is correct in the part. This feature may be used in conjunction with the Fast Checksum verification feature to allow rapid startup of the device. The bootloader can only be entered using the EnterBootloader() API described earlier. Using this feature slightly alters the command sequence to exit the bootloader and start the application after a bootload. See the following figure for additional description of the operation made available by this feature.

Consider the flowchart describing the Software bootloader operation.

The left case assumes that the optional “Auto Run App on CKSUM OK” is enabled. The right case assumes that the Run App if CKSUM OK parameter is disabled.

Figure 9. Device Operation Flow with Run App if CKSUM OK Enabled and Disabled



Download File Created for the Software-only Bootloader

A new format of download file has been created for the Software-only Bootloader. This file is similar to the .txt file output. The Software-only bootloader output file has the extension project_name.iic.

Figure 10. Example Format for project.iic

```
w 38 00 00 FF 38 00 01 02 03 04 05 06 07 p
w 38 [delay=300] 00 p
r 38 x x x p
w 38 00 00 FF 39 00 01 02 03 04 05 06 07 00 34 30 30 30 30 p
w 38 00 10 30 7E 30 30 7E 30 30 30 7E 30 30 30 7D 0F 75 7E p
w 38 00 20 7D 12 A3 7E 7E 30 30 30 7D 0F DD 7E 7E 30 30 30 p
w 38 00 30 7E 30 30 30 7E 30 30 30 7E 30 30 30 7E 30 30 30 p
w 38 00 40 7E 30 30 30 7E 30 30 30 7E 30 30 30 7E 84 p
w 38 [delay=100] 00 p
r 38 x x x p
w 38 00 00 FF 39 00 01 02 03 04 05 06 07 00 35 7E 30 30 30 p
w 38 00 10 7E 30 30 30 7E 30 30 30 7E 30 30 30 7E 30 30 30 p
w 38 00 20 7E 30 30 30 7E 30 30 30 7E 30 30 30 30 30 30 30 p
w 38 00 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 p
w 38 00 40 30 30 30 30 30 30 30 30 7D 11 15 30 83 8F p
w 38 [delay=100] 00 p
r 38 x x x p
   <further records>
w 38 00 00 FF 3B 00 01 02 03 04 05 06 07 p
```

Action | I2C Addr | HW Buf Sub Addr | 78byte buf sub addr | Command | Security Key | Flash addr/0x40

Sample Firmware Source Code

SW Only Bootloader Example

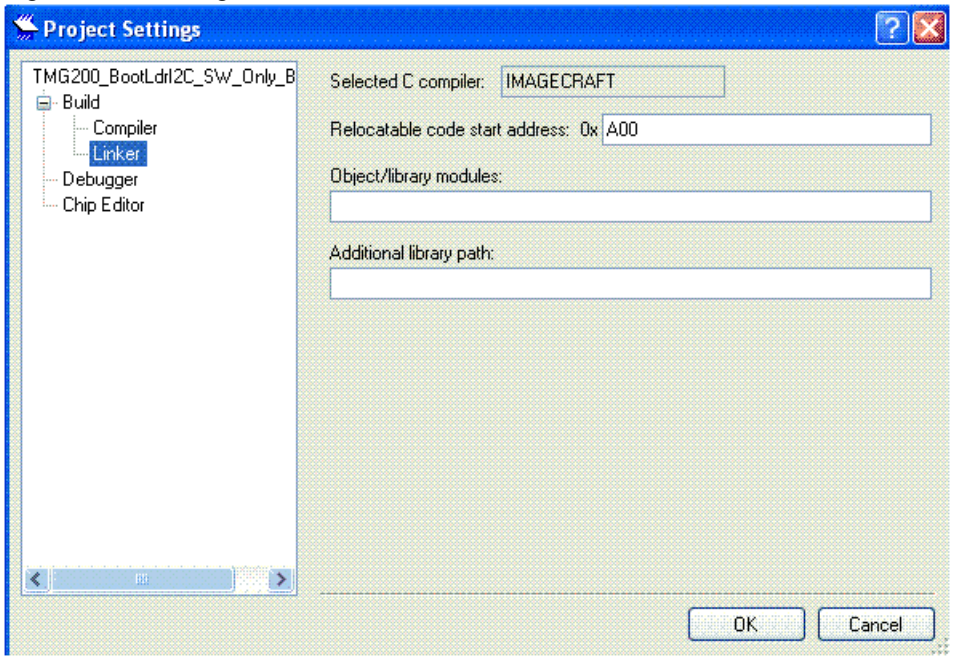
Configure the user module parameters using the values shown in the following figure for both the assembly language and C examples:

Figure 11. Software-only Parameter Setting

Boot_Loader_Addr_HEX	0x0
I2C Clock	100kHz
I2C_Pin	P1[0]-P1[1]
Interrupt_out	Disable
Interrupt_out_Port	None
Interrupt_out_Pin	None
ApplicationCode_Start_Block	0x14
Bootload_when_CKSUM_fails	ENABLE_(deployment)
Run_App_if_CKSUM_OK	ENABLE
BootLoaderKey	0001020304050607
Flash_Program_Temperature_deg_C	-20C
Ignore_N_I2C_Prefix_Bytes	2
BootLdrI2C_ver	0x1000

Ensure the code start address is set correctly (IMAGECRAFT Compiler only).

Figure 12. Setting Code Start Address



The following two examples demonstrate that the Software-only BootLoader User Module is working properly. For this test, two LEDs are connected to P0[0] and P0[1] respectively. First, program the device with example 1 to turn on LED on P0[0] pin. Next, bootload (using the USBtoIIC Bridge GUI) the device with example 2 to turn on LED on P0[1] pin. If the BootLoader User Module works properly, then you will see that LED on P0[1] is on and LED on P0[0] is off. This assumes you have followed the Quick Start section in this datasheet to place the Bootloader User Module in the project.

```
//-------------------------------------------------------------------
// Example 1
//-------------------------------------------------------------------

#include <m8c.h>        // part specific constants and macros
```

```
#include "PSoCAPI.h"    // PSoC API definitions for all user modules


void main(void)
{
    PRT0DM0 = 0xff;
    PRT0DM1 = 0x00;

    while(1)
    {
        PRT0DR = 0x01;
    }
}


//-------------------------------------------------------------------------
// Example 2
//-------------------------------------------------------------------------

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


void main(void)
{
    PRT0DM0 = 0xff;
    PRT0DM1 = 0x00;

    while(1)
    {
        PRT0DR = 0x02;
    }
}
```

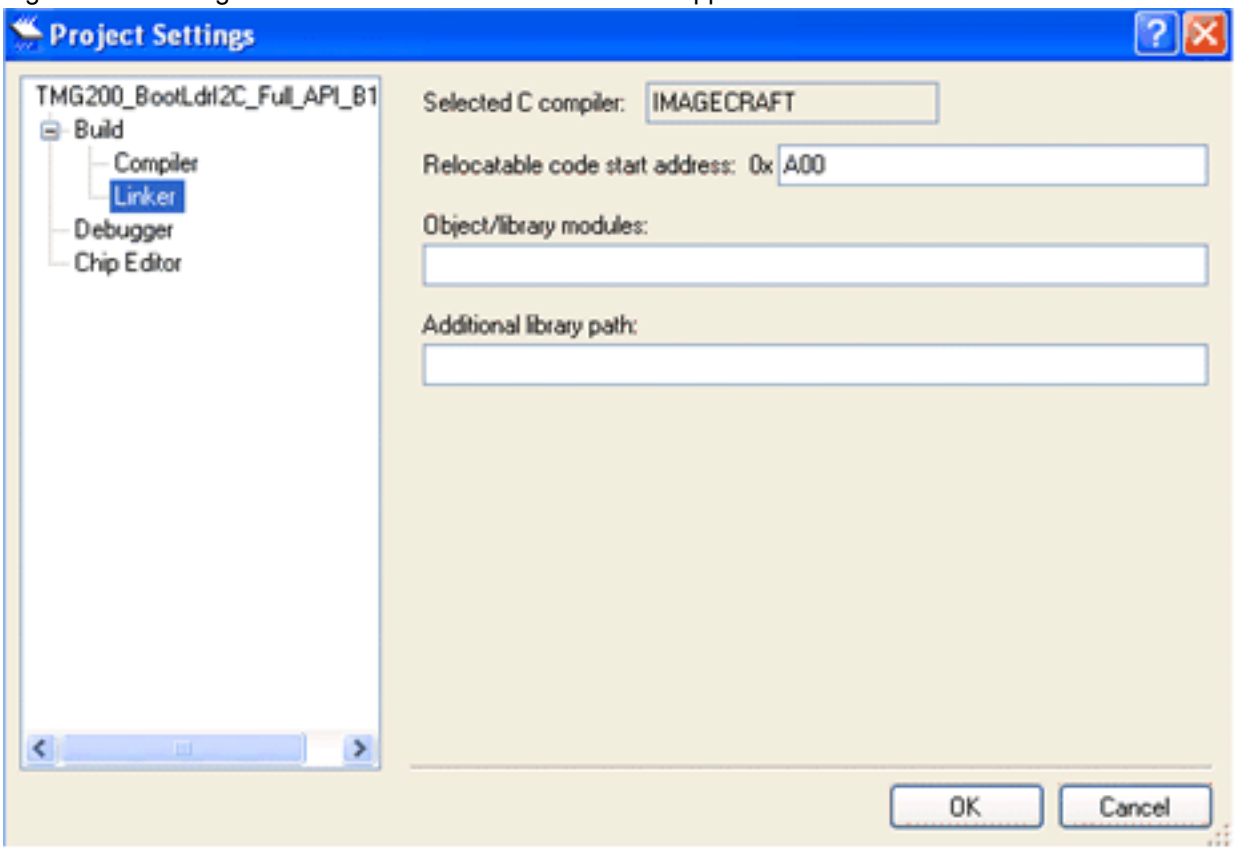
Full I²C API Support with Bootloader Example

Here is an implementation of an I²C BootLoader User Module written in C for "I²C Operation for Bootloader only" and the "Full I²C API Support with Bootloader" options.

Figure 13. Parameters setting for Full I²C API Support

Slave_Addr_HEX	0x1
Boot_Loader_Addr_HEX	0x0
Read_Buffer_Types	RAM ONLY
Communication_Service_Type	Interrupt
ApplicationCode_Start_Block	0x14
Bootload_when_CKSUM_fails	ENABLE_(deployment)
BootLoaderKey	0001020304050607
Flash_Program_Temperature_deg_C	-40C
Ignore_N_I2C_Prefix_Bytes	2
BootLdrI2C_ver	0x1000
I2C Clock	100K Standard
I2C_Pin	P[1]0-P[1]1

Figure 14. Setting Code Start Address for Full I²C API Support




```
//-------------------------------------------------------------------
// C main line
//-------------------------------------------------------------------
#include <m8c.h> // part specific constants and macros
#include "PSoCAPI.h" // PSoC API definitions for all user modules
BYTE result;
WORD wAddr, wByteCount, cTemperature, wByteReadCount;
BYTE pbDataDest[10], pbData[10];
```

```
void main(void)
{
    //example application consists of an EEPROM UM, an LED UM,
    //and a 16-bit timer UM.
    //the EEPROM demonstrates that the EEPROM UM can co-exist
    //with the bootloader, the timer sets a duty cycle and the
    //LED blinks at the set duty cycle.
    //The bootloader UM provides the capability to modify
    //the project (LED duty cycles are conveniently visible.)
    //and use the bootloader to download the modified project.
    //by the time the main() function is executed the project
    //has already been checksummed and verified by the bootloader.
    //init EEPROM data
    wAddr = 0;
    wByteCount = 64;
    wByteReadCount = 10;
    cTemperature = 25;

    //start the bootloader running in the background
    BootLdrI2C_1_Start();
    BootLdrI2C_1_EnableSlave();
    BootLdrI2C_1_EnableInt();

    //start blinking the LED
    Counter24_1_Start();
    Counter24_1_EnableInt();
    LED_1_Start();
    M8C_EnableGInt;

#define INCLUDE_LIB_API
#ifdef INCLUDE_LIB_API
    E2PROM_1_Start();
    result = E2PROM_1_bE2Write( wAddr, pbData, wByteCount, cTemperature);
    E2PROM_1_E2Read( wAddr, pbDataDest, wByteReadCount);
    // Insert your main routine code here.
#endif

#define BUSMODE 0xf5
    while(1)
    {
        asm("nop");
    }
}
```

Here is an implementation of the I²C BootLoader User Module written in assembly language.

```
;-----------------------------------------------------------------------
; 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
_main:
    ; Insert your main assembly code here.
```

```
    lcall BootLdrI2C_1_Start
    lcall BootLdrI2C_1_EnableSlave
    lcall BootLdrI2C_1_EnableInt

    //start blinking the LED
    lcall Counter24_1_Start
    lcall Counter24_1_EnableInt
    lcall LED_1_Start
    M8C_EnableGInt
.terminate:
    jmp .terminate
```

Configuration Registers

This section describes the PSoC Resource Registers used or modified by the I^{2}C Bootloader User Module.

Table 1. Resource I2C_CFG: Bank 0 reg[D6] Configuration Register

Bit	7	6	5	4	3	2	1	0
Value	Reserved	PinSelect	Bus Error IE	Stop IE	Clock Rate[1]	Clock Rate[0]	Enable Master	Enable Slave

Pin Select: Selects either SCL and SDA as P1[5]/P1[7] or P1[0]/P1[1].

Bus Error Interrupt Enable: Enable I^{2}C interrupt generation on a Bus Error.

Stop Error Interrupt Enable: Enable an I^{2}C interrupt on an I^{2}C Stop condition.

Clock Rate[1,0]: Select from 3 valid Clock rates; 50, 100, and 400 Kbps (400 Kbps when CPU_Clk_speed is greater than 6 MHz).

Enable Master: Enable the I^{2}C HW block as a bus Master.

Enable Slave: Enable the I^{2}C HW block as a bus Slave.

Table 2. Resource I2C_SCR: Bank 0 reg[D7] Status Control Register

Bit	7	6	5	4	3	2	1	0
Value	Bus Error	Lost Arb	Stop Status	ACK out	Address	Transmit	Last Recd Bit (LRB)	Byte Complete

Bus Error: Indicates a Bus Error condition has been detected.

Lost Arbitration: In MultiMaster mode indicates loss of arbitration for this device, (device does not control bus).

Stop Status: An I^{2}C stop condition has been detected.

ACK out: direct the I^{2}C block to Acknowledge (1) or Not Acknowledge (0) a received byte.

Address: Received or transmitted byte is an address.

Last Received Bit (LRB): Value of last received bit (bit 9) in a transmit sequence, status of ACK/NAK from destination device.

Byte Complete: 8 data bits have been received. For Receive Mode, the bus is stalled waiting for an ACK/NAK. For Transmit Mode ACK/NAK has also been received (see LRB) and the bus is stalled for the next action to be taken.

Table 3. Resource I2C_DR: Bank 0 reg[D8] Data Register

Bit	7	6	5	4	3	2	1	0
Value	Data							

Received or Transmitted data. To transmit data, this register must be loaded before a write to the I2C_SCR register. Received data is read from this register. It may contain an address or data.

Table 4. Resource I2C_MSCR: Bank 0 reg[D9] Master Status Control Register

Bit	7	6	5	4	3	2	1	0
Value	Reserved	Reserved	Reserved	Reserved	Bus Busy	Master Mode	Restart Gen	Start Gen

Bus Busy: Master Only, set when any bus Start condition is detected, cleared when a Stop condition is detected.

Master Mode: Indicates the device is currently operating as a bus Master.

Restart Gen: Master only, may be set to generate a repeat start for the I^2C bus.

Start Gen: Master Only, When bus becomes idle, generate an I^2C bus start and transmit an I^2C address using data in the data register (I2C_DR).

Appendix

This section contains additional information that you may find useful when creating an I^2C bootloader.

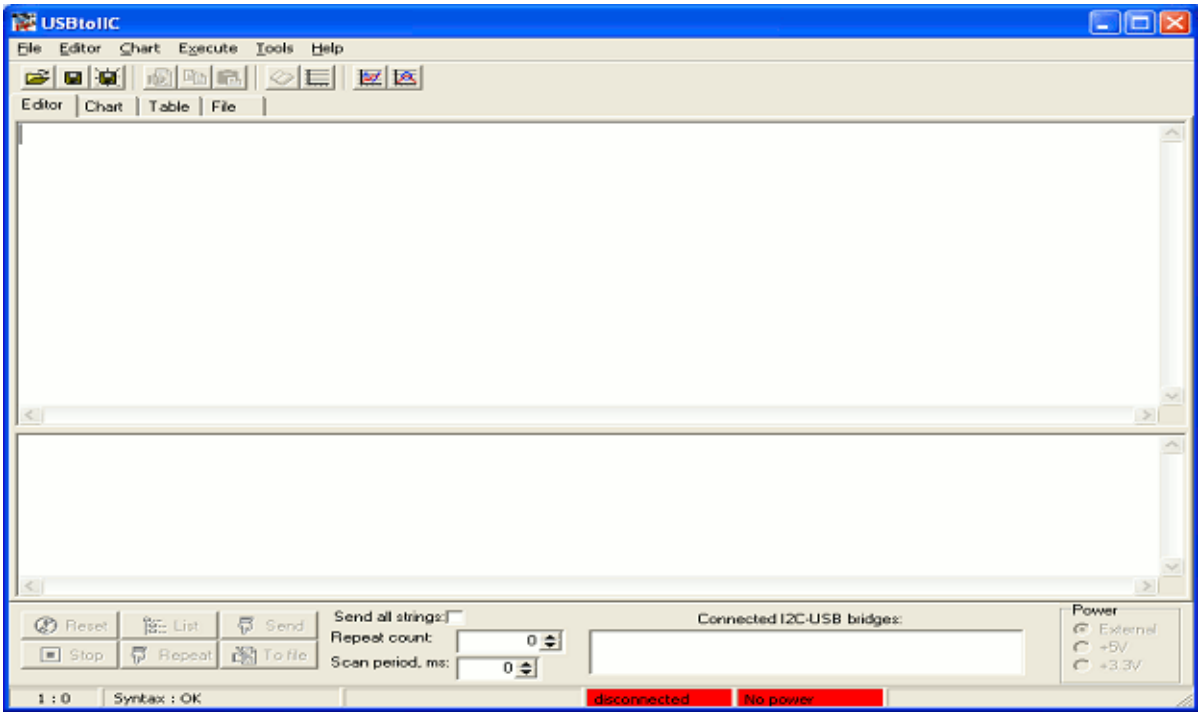
Using the USBtoIIC Bridge GUI Application

The USBtoIIC bridge and the associated GUI is the preferred method to use to download to the bootloader.

More information is available in the application note "I^2C Bootloader Using CY3240 I^2C-USB Bridge," AN45683 discusses the format of the <project_name>.txt file and a procedure for using it to bootload a project. The application note gives information on a tool to convert the .dld format to a .txt format. This is not necessary for the devices described in this datasheet. The <project_name>.txt file is automatically generated.

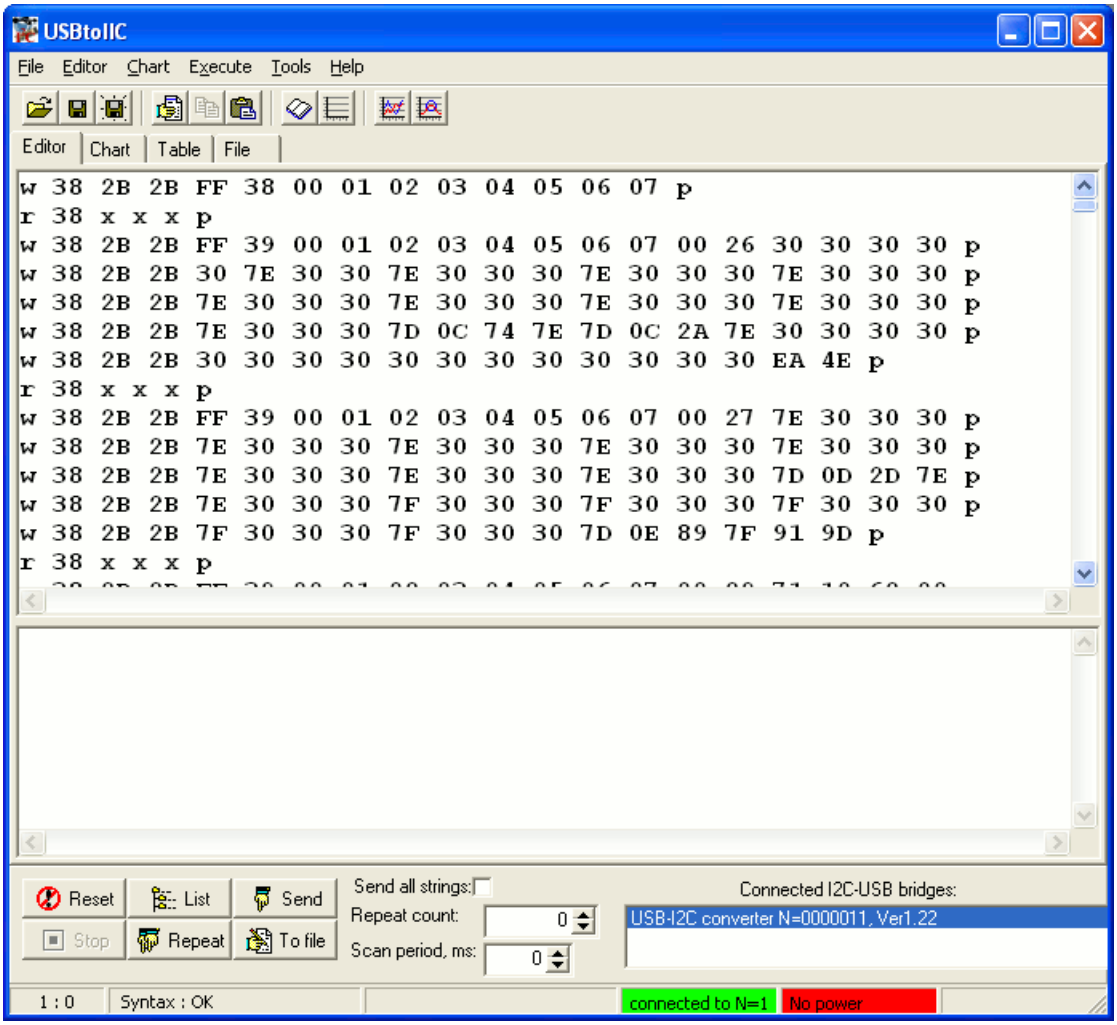
A brief discussion of using the USBtoIIC bridge follows.

Start the application program for the CY3240 USBtoIIC Bridge. The GUI presentation is shown in the following figure:



Import the <projectname>.txt file into the USBtoIIC Bridge GUI

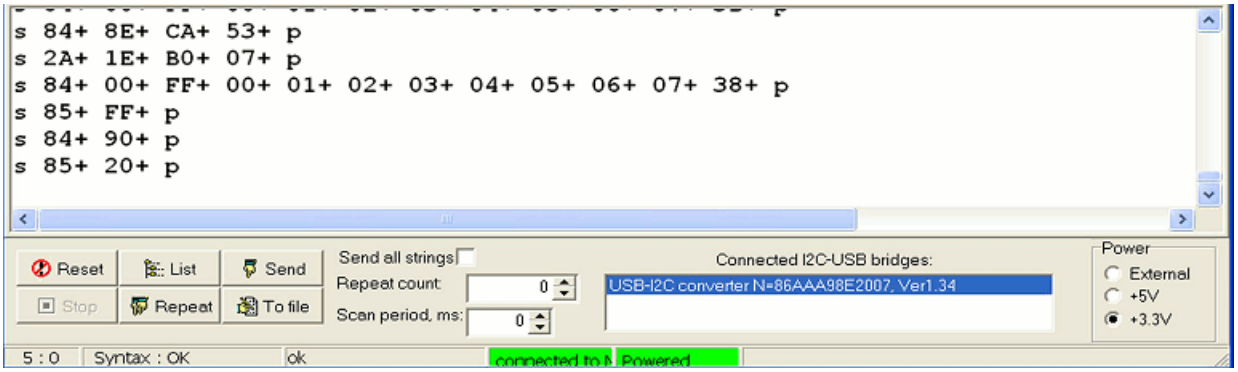
Select **File > Open**, and browse to the output directory of the project that you wish to bootload. Look for a file named <projectname>.txt. It may be necessary to choose the file-type as "all files" in the file browser window. If this file is not present, it may be necessary to regenerate it using the bootloader tools that are described elsewhere in this document. The file may take a few seconds to load after it is selected for open. To ensure that the file has loaded completely, right click in the lower window, if a menu appears, then the GUI is ready.



Connect the CY3240 USBtoIIC bridge to the target system. Use the GUI to set the power to the desired level. This power may be used to supply power to the target depending on the power requirements of the target system. The status bar at the bottom of the GUI must indicate that the bridge is connected and powered. Also, make sure the "Send all strings" box is checked at the bottom of the USBtoIIC GUI to send the entire download file. By unchecking the "Send all strings" box small pieces of the download file may be highlighted and sent for testing purposes.



Click the **Send** button to download the new code. The status of various I²C transactions appears in the status area. Note that "+" denotes a successful transaction.



Bootloader I²C Download Protocol

The application note "*I²C Bootloader Using CY3240 I²C-USB Bridge*", AN45683 discusses the format of the <project_name>.txt file and a procedure for using it to bootload a project. The application note gives information on a tool to convert the .dld format to a .txt format. This is not necessary for the devices described in this datasheet. The <project_name>.txt file is automatically generated.

A discussion of the format of the file <project_name>.dld file follows: Note that the format of the file for devices with a 64-byte and 128-byte flash block size does not change. The size of the block to be programmed is supported internally in the bootloader.

Two sample download records are shown here. These records consist of actual data that would be transmitted between the I²C master and a slave to be bootloaded. The format of the records is shown in the following figure:

Figure 15. Format of Records

First Download Record

70 00 00 FF 38 00 01 02 03 04 05 06 07	Enter bootloader FF, 38
71 20	Master reads bootloader slave to acquire status
70 00 00 FF 39 00 01 02 03 04 05 06 07 00 02 40 40 40 40	No status request or response
70 00 10 30 7E 30 30 7E 30 30 30 7E 30 30 30 7E 30 30 30	
70 00 20 7E 30 30 30 7E 30 30 30 7D 0B 64 7E 7E 30 30 30	
70 00 30 7E 30 30 30 7E 30 30 30 7E 30 30 30 7E 30 30 30	
70 00 40 7E 30 30 30 7E 30 30 30 7E 30 30 30 2E B2	
71 20	Status request and response

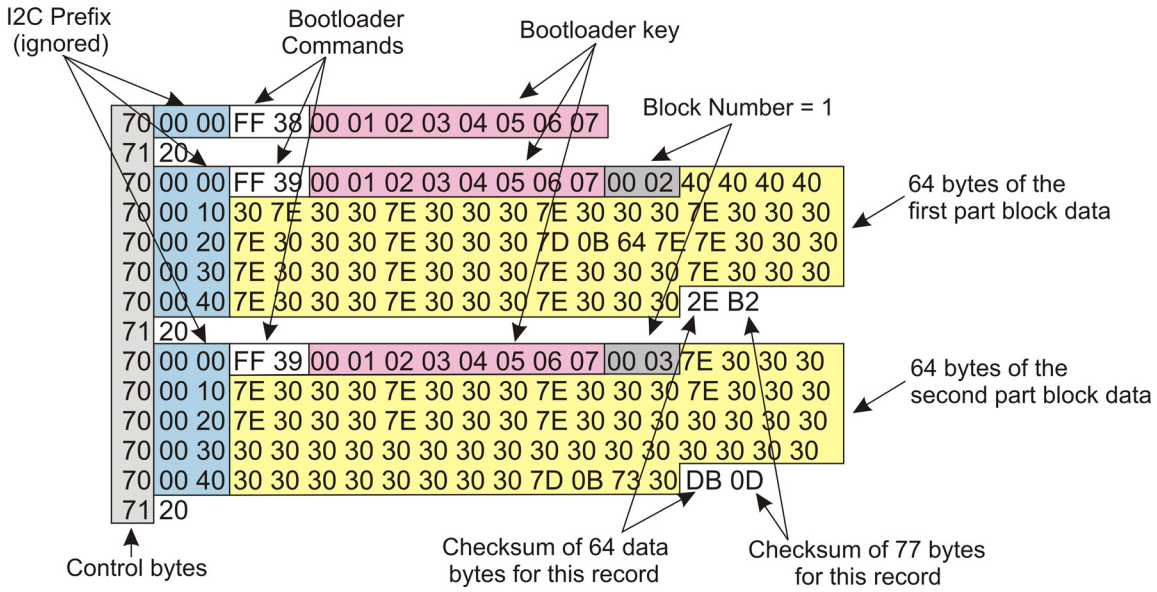
.....

Last Download Record

70 00 00 FF 39 00 01 02 03 04 05 06 07 01 FF 30 30 30 30	No status request or response
70 00 10 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	
70 00 20 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	
70 00 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	
70 00 40 30 30 30 30 30 30 30 30 30 30 30 30 00 54	
71 20	Status request and response
70 00 00 FF 3B 00 01 02 03 04 05 06 07	Exit bootloader FF, 3B
71 20*	Status request and response

The following figure shows the format of the first record:

Figure 16. Format of First Record



Every line begins with a control byte. The two control bytes used by the download protocol are:

- 70 – Slave address 38 write. The address is not considered part of the byte count.
- 71 – Slave address 38 read. The expected response to a slave address read is 0x20, success. Other possible responses are:

Table 5. Slave Address Read Responses

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

For details, see the BootLoader operation flowchart at the end of this document.

Slave address write commands do not require responses, so the next two bytes of each of the slave address write lines is an I^2C prefix that the bootloader ignores. Use the Ignore_N_I2C_Prefix_Bytes parameter to set the number of prefix bytes used in your application.

The first and third lines of the sample download record contain bootloader commands. The following bootloader commands are used:

Table 6. Bootloader Commands

Command	Meaning
FF38	Enter bootloader
FF39	Write block
FF3B	Exit bootloader

All bootloader commands must be sent with the bootloader key. The bootloader ignores commands that are not sent with the proper key. You set the bootloader key with the Bootloader_Key parameter.

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. The third and fourth 64-byte blocks contain the checksum information that will be written into the second 128-byte flash block. The format of the checksum block is discussed in this section. Each of the other write block command transmits a 64-byte hex record to the bootloader in five packets totalling 78 bytes (neglecting addresses and discarded prefixes). For devices with 128 byte blocks, two of these 64byte transfers are accumulated. Block organization is verified to consist of first an even numbered block and then an odd numbered block. When a complete block is accumulated the flash is written.

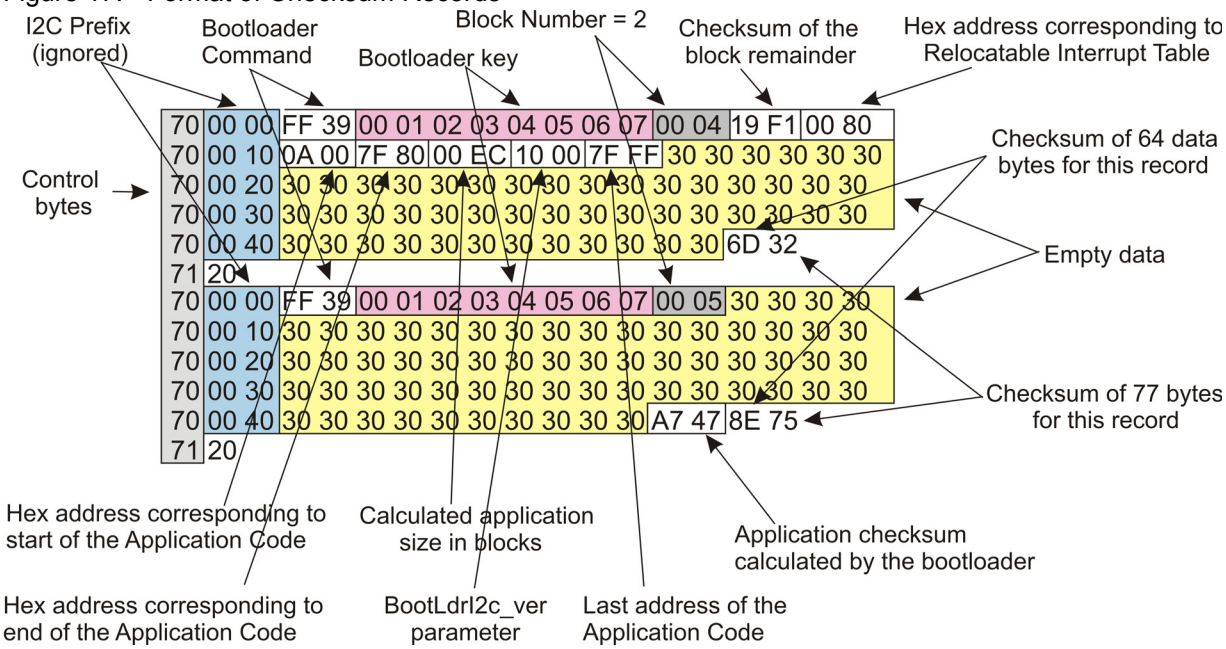
The first line of the write block command contains a control byte, an ignored 2-byte I^2C prefix, the write block command, the bootloader key, the block number being transmitted, and the first four bytes of data. The block number in this example is 0x0002, which corresponds to ROM address 0x0080.

The next three lines contain only the control byte, the I^2C prefix, and 16 bytes of data. The last line of the write block command contains the control byte, the I^2C prefix, the final 12 bytes of data, and two 1-byte checksums. The first checksum, 0x2E in this example, is the checksum of the data bytes for this record. The second checksum, 0xB2 in this example, is the checksum of the 77 byte record, excluding address bytes and prefixes. Address bytes and prefixes are verified internally by the bootloader as they are received.

At the conclusion of the write block command, another status request is sent, and it results in the response shown.

All THE blocks transmitted have the same format. The following figure shows the format of the checksum records:

Figure 17. Format of Checksum Records



The third and fourth records contain the checksum block (note that the block number is always 0x0002). A description of the data in these records is given in this section.

The first line contains a control byte, an I^2C prefix, the bootloader write block command, the bootloader key, and the block number just as the other records did. The next two bytes contain an optional checksum for the remainder of the block, 0x19F1 in this case. The last two bytes of the line contain the hex address of the Relocatable Interrupt Vectors Table (see Figure 2 for more information).

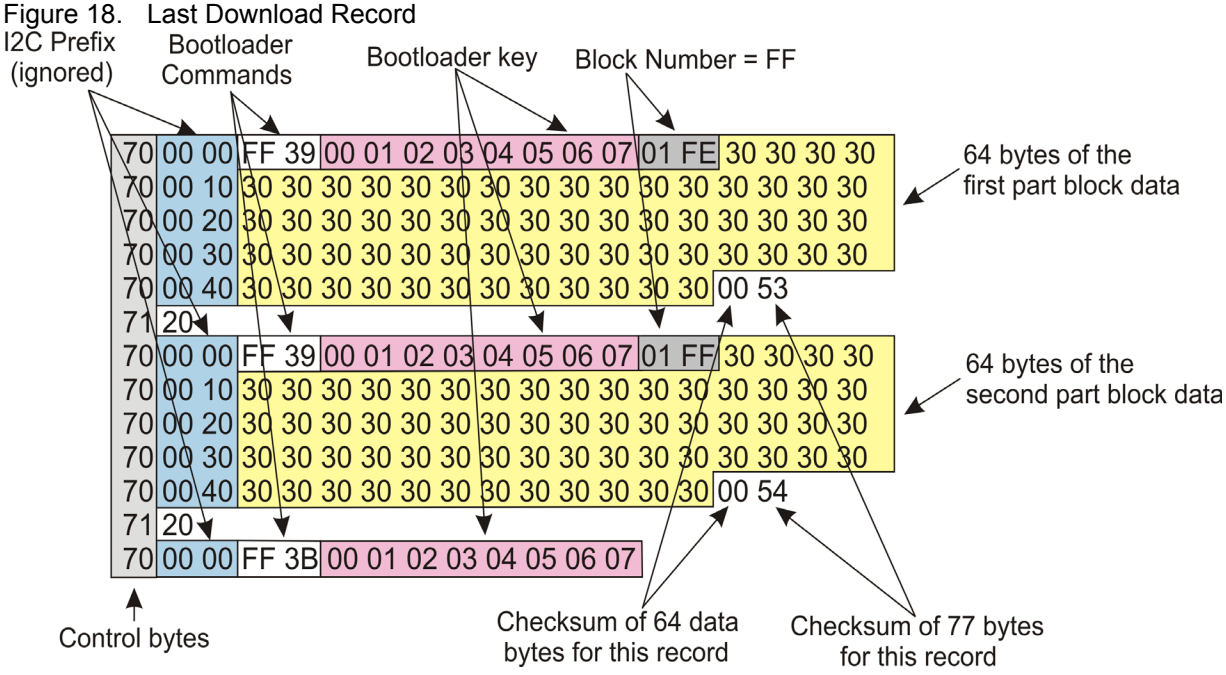
The second line of the record contains a control byte and an I^2C prefix, followed by a two byte value that represents the hex address of the ApplicationCode_Start_Block user module parameter calculated from block 0x14. The next two bytes are the hex address of the end of Application Code calculated from block 0xFF. This is followed by two bytes that are the application size in blocks. The next two byte real data value on this line is the bootloader version number from the BootLdrI2C_ver parameter. The final two byte real data value on this line is the Last address of the Application Code. The remainder of the line is empty data space.

The next two lines contain additional empty data space. The last line of the checksum block contains empty data space until the last four bytes of the line. The last four bytes contain a two byte application checksum calculated by the bootloader, a one byte block checksum from the Intel hex record, and finally the checksum of the 77 byte record, excluding address bytes and prefixes. Address bytes and prefixes are verified internally by the bootloader as they are received.

The next line contains another status request and response.

Figure 18 shows the last record.

Figure 18. Last Download Record



The bootloader exit command consists of control byte 70, a two byte prefix, the bootloader exit command 0xFF3B, and the bootloader key.

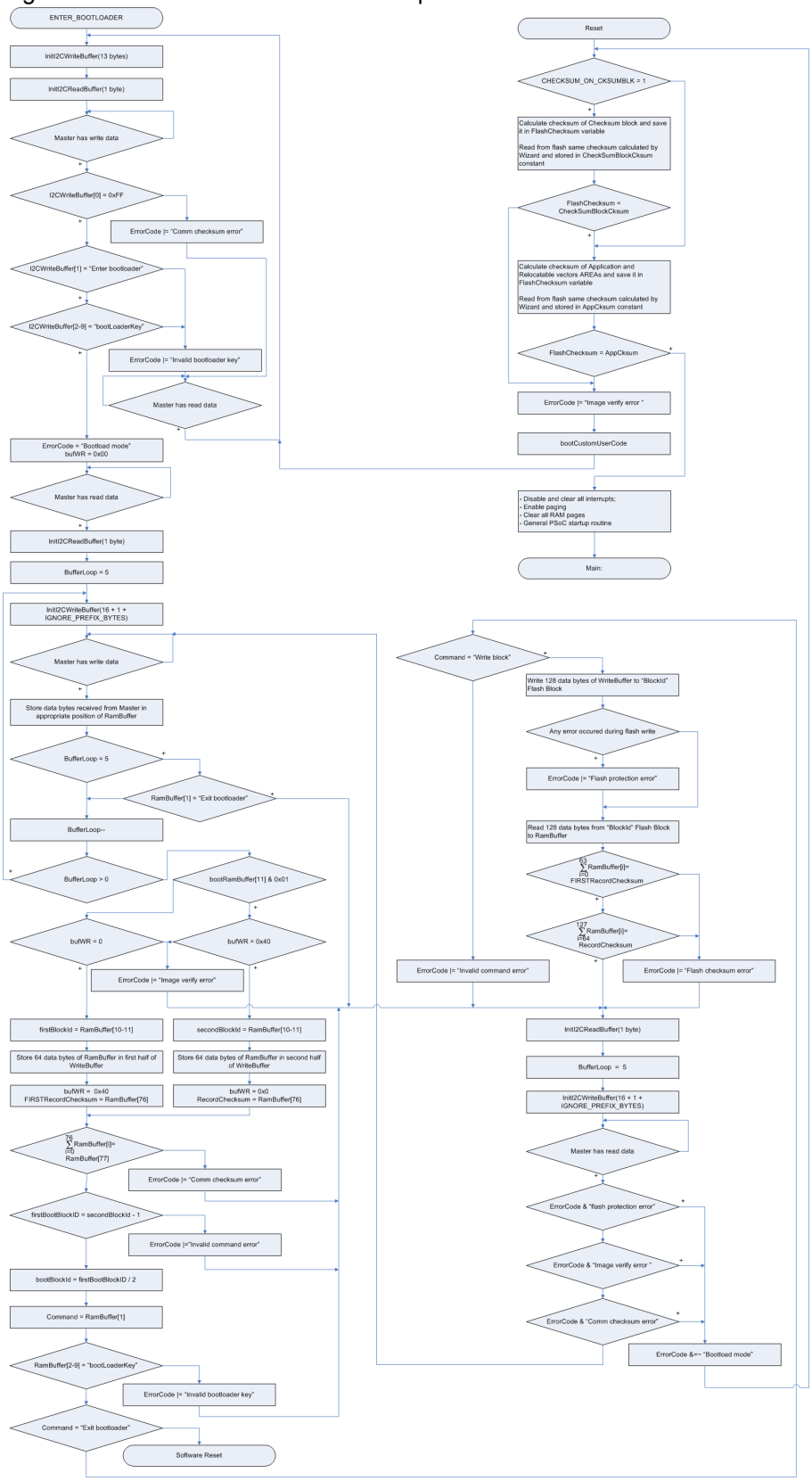
The last line is a final status request and possibly a response. When the exit bootloader command is received, the target system immediately executes an internal reset and begins verifying the checksum for the downloaded application using the parameters given in the checksum block. Depending on the speed of the host system, the bootloader may already have begun its reset process before the master is able to receive a valid status byte. For this reason the status (0x20) may not always be present. Instead, the address 0x71 may simply be NAKed.

BootLdrI2C and E2PROM User Module Coexistence

When placing an E2PROM UM in a Bootloader project, allocate E2PROM blocks in Customer Reserved Blocks area. This area is between the Bootloader Code Area and Application Code Area (see the

Bootloader Memory Organization in Figure 2 on page 4 for information). That way E2PROM blocks are not part of the Application Code Area, and will not be calculated as part of Application checksum.

Figure 19. BootLdrI2C User Module Operations Flowchart



Version History

Version	Originator	Description
1.0	DHA	Added Version History
2.00	DHA	1. Reorganized the memory map. 2. Placed Reloc Interrupt Vectors Table at address 0x0080. 3. Placed Checksum Block at address 0x0100. 4. Placed Bootloader Start at address 0x0180. 5. Confirmed application block size is lesser than 256. 6. Added Bootloader API Jump Table. 7. Updated user module Parameters table.
2.10	DHA	1. Replaced .Literal and .EndLiteral statement with .nocc around SSC call. 2. Removed export `@INSTANCE_NAME`_EnterBootloader statement. 3. Added User code section for I2C address compare customization.
2.20	DHA	1. Fixed the initialization of Application_Checksum_Block and TWO_Block_Relocatable_Interrupt_Table. 2. Added CYONSFN3050 LP devices support. 3. Added support to display bootloader output files in Workspace Explorer. 4. Added description in the "Improper Settings in flashsecurity.txt" section.
2.30	DHA	1. Updated area declarations to support Imagecraft optimization. 2. Updated Javascript to enhance the wizard performance. 3. Fixed issues related to coexistence with the watchdog timer. 4. Added more information in this user module datasheet about the checksum calculation in the *.iic file.
2.40	DHA	1. Added I2C_XCFG register initialization to set clock for I2C block to support CY8C20xx7/7S. 2. Moved I2C clock initialization for CY8C20xx7/7S to BootLoader Wizard.

Version	Originator	Description
2.50	DHA	1. Added output in csv format. 2. Added CYRF89x35 device support. 3. Limited BootLoaderKey parameter to 16 symbols. 4. Corrected RAM and ROM usage in the user module datasheet.
3.00	HPHA	1. Corrected method of clearing posted interrupts. 2. Added CY8C24093 and CY7C69000 support. 3. Fixed memory overlap problem; variables that are used by both Bootloader and Application were locked at the end of RAM page 0.
3.10	HPHA	Added CY8C20055 support.
3.20	HPHA	1. Resolved the issue with bootloader communication at 400 kHz of I2C Clock. 2. Improved the algorithm that sets the "Relocatable code start address" parameter of the ImageCraft compiler.

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

Copyright © 2008-2014 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.