



Application Note

AN2100

Bootloader: PSoC™

Author: Andrew Smetana

Associated Project: Yes

Associated Part Family: CY8C27xxx, CY8C29xxx

PSoC Designer Version: 4.2

Associated Application Notes:

Abstract

This Application Note describes a bootloader using PSoC's self-programming capability. It allows users to reprogram the user Flash memory through a UART interface. A dedicated PC-based Windows application has been developed to simplify PSoC programming via the bootloader.

Introduction

PSoC devices can be programmed after they have been installed in a system. In-System Serial Programming (ISSP) is the standard solution to this feature. The bootloader for PSoC programming was developed as an alternative to this method. It excludes use of the ICE-4000 or ICE-Cube and third-party programming tools. It requires only a serial cable, RS232 level translator, and a terminal program to modify the PSoC firmware.

Bootloader programs are used to upgrade user code in Flash memory without physically replacing the part on the board. To enable this feature, the user must hardcode the bootloader program in the chip via the YProgrammer. In this application, the bootloader code is protected in the high memory addresses area. User space is unprotected and can be upgraded. To increase reliability of the user program in unprotected Flash memory, this area can be protected by checksum, which is verified after each CPU reset. There is an additional feature that suppresses loading the part in non bootloader-based projects, which eliminates the possibility of incorrect program operation. When a bad checksum is calculated or user code is not detected, the program automatically enters bootloader mode for reprogramming.

There are two methods to access the bootloader. The first method is to press a button during reset and in the second method the PSoC interacts with the PC terminal program after power-on-reset.

A terminal program has been developed for the PC to provide a simple user interface for Flash upgrading. You can choose any hexadecimal file and watch the programming process using the progress bar. In this program, the Flash block programming control and timeout functions are implemented to provide reliable program operation. The user is well informed of programming status or any errors.

Hardware Implementation

Communication between the host PC and the PSoC device is established using a serial RS232 interface. The schematic of the bootloader is shown in Figure 1.

A MAX232 (RS232 interface chip from Maxim Integrated Products) is used as a level shifter to translate the TxD and RxD signals to the host PC. A button is needed to enter bootloader mode. The LED is turned on when bootloader mode is entered. During the programming process the LED blinks.

The UART interface is implemented by a serial receiver and transmitter, operating at 115200-baud rate.

The clock to these modules comes from the VC3 source. A Counter16 User Module is used to implement the timeout function. When the controller tries to enter bootloader mode, the counter sends a connection request to the PC after power-on-reset and waits for a response. This is the second method in which to enter bootloader mode.

To distinguish between the CY8C27xxx and CY8C29xxx families, use the following rule: Numbers/addresses referring to CY8C27xxx devices are in plain font: 0x3600/0x3FFF. Numbers/addresses referring to CY8C29xxx devices follow the plain font reference and are bold, italic and in parenthesis: **0x3600/0x3FFF (0x7400/0x7FFF)**.

Memory space of a typical bootloader program is shown in Figure 2. Fully protected blocks are shadowed while the unprotected are not.

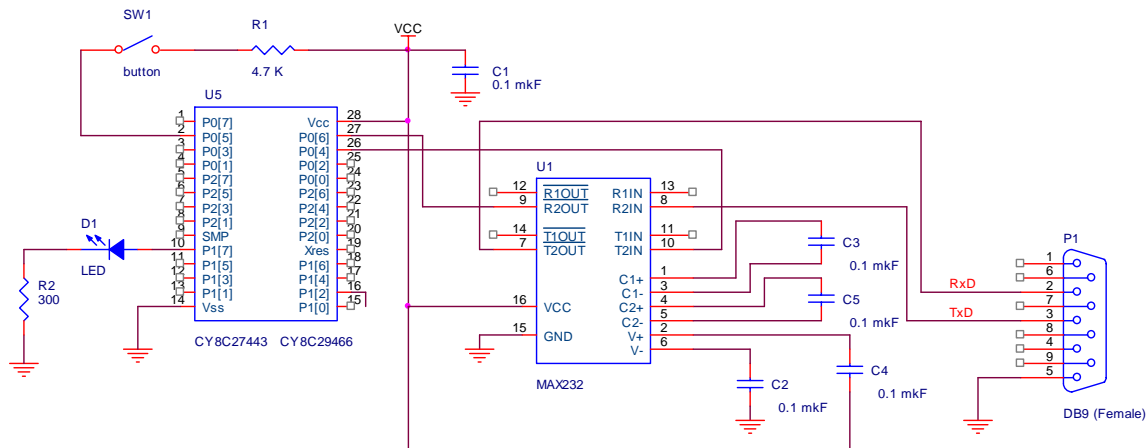


Figure 1. Bootloader Electric Circuit

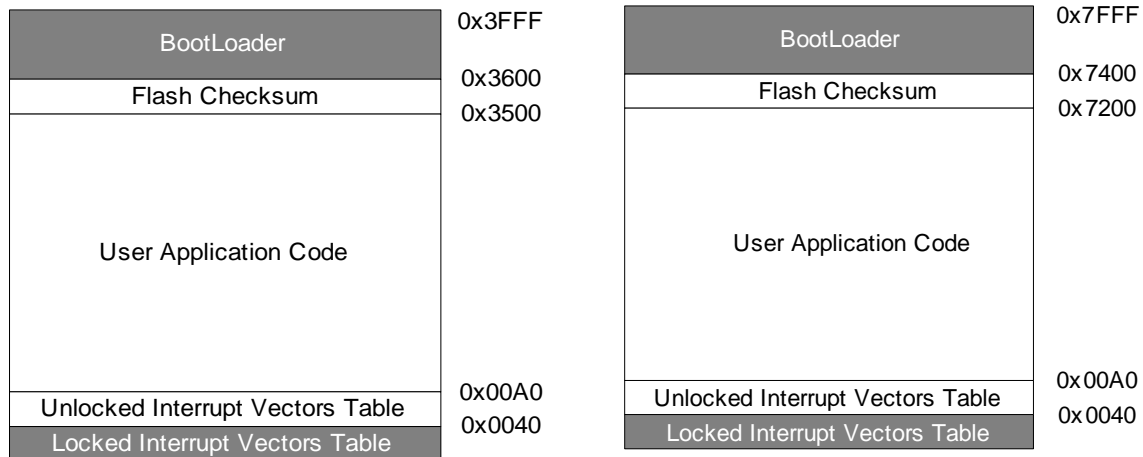


Figure 2. Bootloader Memory Space: CY8C27xxx – Left, CY8C29xxx –Right

The high address space beginning at address 0x3600/0x3FFF (**0x7400/0x7FFF**) is intended to contain bootloader firmware. This memory area is protected from write and erase operations. The first Flash block (64 bytes) is also fully protected. The reset vector jumps directly to boot-verify code, which is part of the bootloader.

This ensures that the bootloader code will be always available regardless of aborted boot loading attempts or incorrect user code.

Because the first 64 bytes of Flash are fully protected, changing addresses in the first 15 interrupt routines is not possible.

In order to change interrupt vector addresses, the locked user interrupts (up to 0x003F) jump to an unlocked table in the *boot.asm*. All changes for *boot.asm* should be done in *boot.tpl*, the template, because after each application generation, the content of the *boot.tpl* file is copied to *boot.asm*.

Address 0x00A0 is the first address after the unlocked interrupt vector table. The byte sequence: 0, 1, 2, 3, 4, 5 (**2, 3, 4, 5, 6, 7**) should be placed at 0x00A0, which is used by the bootloader to verify the loaded project.

If a user program is based on the bootloader, it must have these bytes written in the Flash at this address. After this control byte sequence, all user program code and the `__start` routine follows.

The memory space from address 0x3500 to 0x35FF (**0x7200 to 0x73FF**) exclusively is used to store checksum information for each unprotected block. Blocks that can be protected by checksum have IDs from 1 to 211 (**1 to 455**). In the bootloader project, the last block to be checked is set at address 0x3500 (**0x73FE, 0x73FF**). If it is set to 0, then the checksum feature is turned off. The usage of this memory is shown in Figure 3.

This checksum provides more reliable operation in the case of soft fault and accidental Flash rewrites because all user program code is unprotected from internal writings. Checksum is checked after each reset and if it is bad, bootloader mode is automatically entered.

The protection type of each Flash block is set in the *flashsecurity.txt* file (which can be found in the source tree of your PSoC Designer project). You should set “W” to fully protected blocks and “U” or “R” – for unprotected blocks. It takes longer to do Flash writes (per block) using the ICE than on the actual chip. It is advised that you test the bootloader on the actual chip to estimate real performance. You can disconnect the pod from the ICE and run the program to see upload speed for the real code. Figure 4 shows how *flashsecurity.txt* should be set.

To relocate memory areas for the bootloader project you should make changes in *custom.lkp*, which can be found in the root directory of your project. This file contains the following information:

```
-bBootChecksum:0x3500.0x35ff      CY8C27xxx
-bBootLoaderArea:0x3600.0x3fff
-----
-bBootChecksum:0x7200.0x73ff      CY8C29xxx
-bBootLoaderArea:0x7400.0x7fff
```

These records determine borders of bootloader and checksum segments. To learn more about this file, please refer to the *PSoC Designer: C Language Compiler User Guide*.

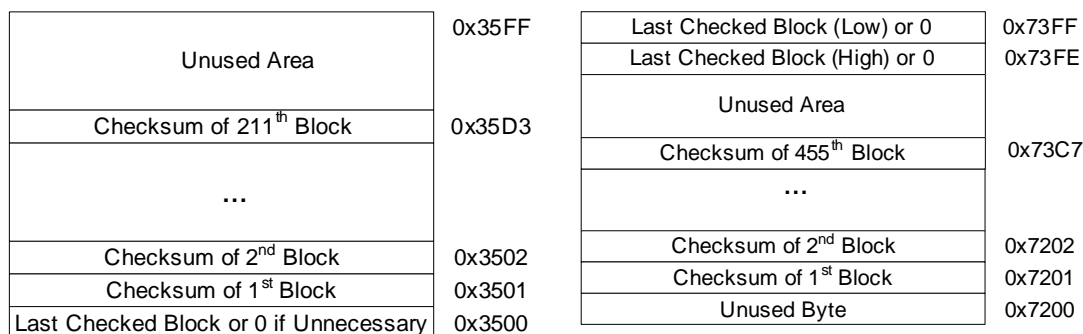


Figure 3. Bootloader Checksum Memory Area: CY8C27xxx – Left, CY8C29xxx – Right

```

; 0 40 80 C0 100 140 180 1C0 200 240 280 2C0 300 340 380 3C0 (+) Base Address
W U U U U U U U U U U U U U U U ; Base Address 0
U U U U U U U U U U U U U U U U ; Base Address 400
U U U U U U U U U U U U U U U U ; Base Address 800
U U U U U U U U U U U U U U U U ; Base Address C00
; End 4K parts
U U U U U U U U U U U U U U U U ; Base Address 1000
U U U U U U U U U U U U U U U U ; Base Address 1400
U U U U U U U U U U U U U U U U ; Base Address 1800
U U U U U U U U U U U U U U U U ; Base Address 1C00
; End 8K parts
U U U U U U U U U U U U U U U U ; Base Address 2000
U U U U U U U U U U U U U U U U ; Base Address 2400
U U U U U U U U U U U U U U U U ; Base Address 2800
U U U U U U U U U U U U U U U U ; Base Address 2C00
U U U U U U U U U U U U U U U U ; Base Address 3000
U U U U U U U U U U U U U U U U ; Base Address 3400
W W W W W W W W W W W W W W W W ; Base Address 3800
W W W W W W W W W W W W W W W W ; Base Address 3C00
; End 16K parts

```

a) CY8C27xxx

```

; 0 40 80 C0 100 140 180 1C0 200 240 280 2C0 300 340 380 3C0 (+) Base Address
W U U U U U U U U U U U U U U U ; Base Address 0
U U U U U U U U U U U U U U U U ; Base Address 400
U U U U U U U U U U U U U U U U ; Base Address 800
U U U U U U U U U U U U U U U U ; Base Address C00
; End 4K parts
U U U U U U U U U U U U U U U U ; Base Address 1000
U U U U U U U U U U U U U U U U ; Base Address 1400
U U U U U U U U U U U U U U U U ; Base Address 1800
U U U U U U U U U U U U U U U U ; Base Address 1C00
; End 8K parts
U U U U U U U U U U U U U U U U ; Base Address 2000
U U U U U U U U U U U U U U U U ; Base Address 2400
U U U U U U U U U U U U U U U U ; Base Address 2800
U U U U U U U U U U U U U U U U ; Base Address 2C00
U U U U U U U U U U U U U U U U ; Base Address 3000
U U U U U U U U U U U U U U U U ; Base Address 3400
U U U U U U U U U U U U U U U U ; Base Address 3800
U U U U U U U U U U U U U U U U ; Base Address 3C00
; End 16K parts
U U U U U U U U U U U U U U U U ; Base Address 4000
U U U U U U U U U U U U U U U U ; Base Address 4400
U U U U U U U U U U U U U U U U ; Base Address 4800
U U U U U U U U U U U U U U U U ; Base Address 4C00
U U U U U U U U U U U U U U U U ; Base Address 5000
U U U U U U U U U U U U U U U U ; Base Address 5400
U U U U U U U U U U U U U U U U ; Base Address 5800
U U U U U U U U U U U U U U U U ; Base Address 5C00
U U U U U U U U U U U U U U U U ; Base Address 6000
U U U U U U U U U U U U U U U U ; Base Address 6400
U U U U U U U U U U U U U U U U ; Base Address 6800
U U U U U U U U U U U U U U U U ; Base Address 6C00
U U U U U U U U U U U U U U U U ; Base Address 7000
W W W W W W W W W W W W W W W W ; Base Address 7400
W W W W W W W W W W W W W W W W ; Base Address 7800
W W W W W W W W W W W W W W W W ; Base Address 7C00
; End 32K parts

```

b) CY8C29xxx

Figure 4. flashsecurity.txt Settings

Firmware Implementation

Bootloader firmware implementation occurs in the following files: *boot.asm*, *bootloader.c*, *bootloaderconfig.asm*, and *flashapi.asm*. The description of each file is below.

The **boot.asm** is a project start-up file. It reflects the locked and unlocked interrupt vector tables, boot control sequence 0, 1, 2, 3, 4, 5 (**2, 3, 4, 5, 6, 7**) placed at 0x00A0 address, `__Start` routine – user application initialization procedure, and the `__Boot_Start` routine to where the program jumps when the first instruction is executed after reset. This routine initiates the device for bootloader mode, sets the CPU clock equal to 12 MHz, sets top of the stack, loads user module configuration, and then calls `BootLoader()` function. `__Boot_Start` routine is allocated in the protected bootloader area 0x3600 ..0x3FFF (**0x7400 ..0x7FFF**).

`__Start` routine carries out initial CPU operations and loads user modules for customer-developed application. At the end of this procedure it calls `_main()` project function.

The **bootloaderconfig.asm** contains user module configurations and APIs. It includes serial receiver and transmitter user modules and a timeout counter module. All module API names start with “Boot_” to minimize confusion with user variables, functions, and labels. The configuration function name is `Boot_LoadConfigInit` and it is called from the `__Boot_Start` routine at the initial stage of bootloader operation.

The **flashapi.asm** implements functions to handle Flash memory operations. Among these functions are:

`bflashWriteBlock` – executes Flash block write action;

`FlashReadBlock` – reads one block of Flash;

`FlashCheckSum` – calculates Flash block checksum.

Boot_Is_Program_Good verifies if a user-loaded program is created using the bootloader project. It simply tests 6 bytes starting from address 0x00A0. If the bytes are not equal to 0, 1, 2, 3, 4, 5 (**2, 3, 4, 5, 6, 7**), then the function returns the error result.

For reliability, it is imperative to place code related to Flash modification in the bootloader segment.

bootloader.c is a main file in the bootloader. It contains all functions required for boot-verify operations. It makes the connection with the PC host and functions, which perform data transmission with the PC during the Flash programming process. These routines are explained below:

`void BootLoader()` is called after the bootloader was initialized. First, it tries to set up communication with the PC. If the terminal program on the PC responds, `Boot_PerformWrite()` is called.

Alternately, `Boot_Is_Program_Good` is called to verify that the project has loaded in the user Flash space. If the user code is not based on the bootloader project, then the program automatically enters bootloader mode. The next phase is verification of Flash blocks' checksum.

If the previous two steps successfully have been completed, the boot-verify action is performed. It checks if the bootloader button has been pressed. If pressed, the program enters the bootloader mode. Otherwise, it calls the `__Start` routine to start user program execution.

Once the program is in bootloader mode, the LED is switched on and the program waits for communication from the PC. After, `Boot_PerformWrite()` is called to complete Flash programming operations. If this stage is successfully completed, the program executes a software reset by using the supervisory code.

`void Boot_PerformWrite()` obtains Flash blocks from the PC in Intel HEX format, writes them in Flash, reads them back, and sends them to the PC for comparison with source blocks. The frame of the data block has the following structure:

*Start Symbol – 'S' (1 byte);
Flash Block Data (68 bytes);
Finish Symbol 'F' (1 byte).*

The Flash block data block has the following records: length of data to be written (1 byte), starting address (2 bytes), type of data (1 byte), and immediate data block (64 bytes).

All fields of data represent information in ASCII encoded hexadecimal. This means that every 8 bits of information are encoded in two ASCII characters.

Once all blocks are written and if the checksum option is set, the function calculates the checksum for each block protected by the checksum and writes this information in the checksum area 212 – 215 (**456 - 463**) Flash blocks.

`char Boot_ASCIIToBYTE(char low, char high)` translates ASCII encoded byte representation into binary format.

The following functions are high level UART APIs:

`BYTE Boot_UART_cGetChar(void)` reads a byte from RxD, blocking program execution if the buffer is empty.

`void Boot_UART_PutChar(char TxDData)` sends a character to the TxD buffer, blocking program execution if the buffer is not empty.

`void Boot_UART_CputString()` sends ASCII string out of TxD port.

Macros definitions:

`#define LAST_BLOCK_TO_CHECK 211`

This macro defines the last block of Flash that will be protected by checksum. Block distribution is shown in Figure 5.

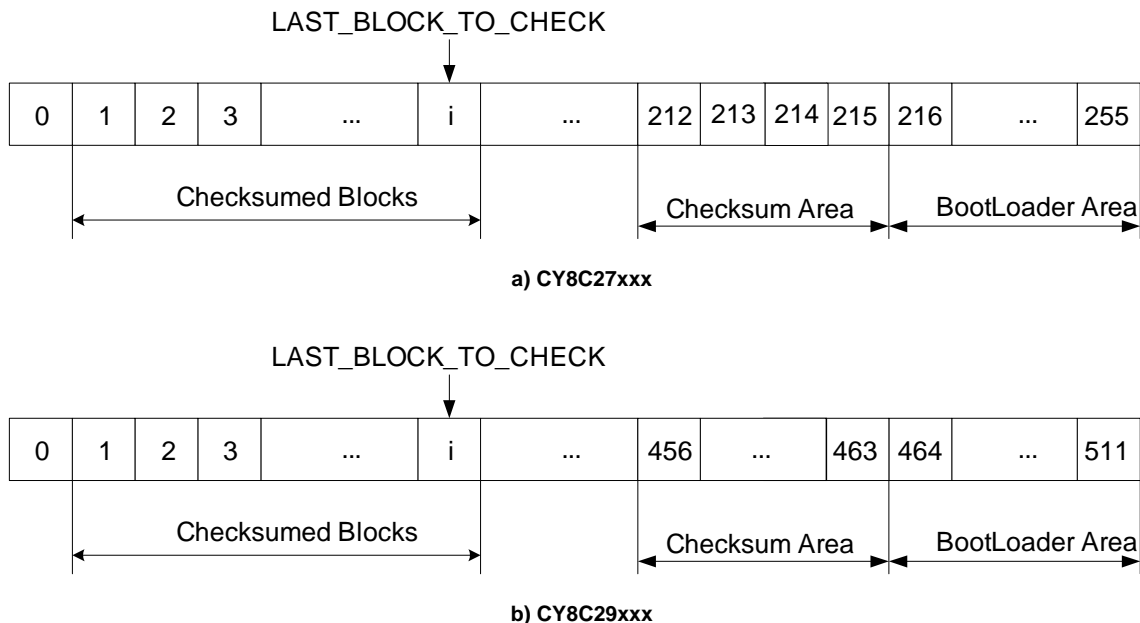


Figure 5. Checksum Blocks

Blocks from 1 to `LAST_BLOCK_TO_CHECK` are protected by checksum. Blocks from `LAST_BLOCK_TO_CHECK+1` to 211 (**455**) can be used as reprogrammable (EEPROM emulation) data storage to save calibration parameters. You can set this macro to 0 to disable the checksum feature.

`#define SUPPORT_CONNECT_BY_PSOC 1`

If this macro is not 0, then after power-on-reset, the bootloader tries to connect with the PC to automatically enter bootloader mode.

The following macros are defined for easy bootloader pin redefinition:

`GETBUTTON()` – get value on switch button pin;
`BOOT_LOADER_MODE_LED_ON()` – turn on LED;
`BOOT_LOADER_MODE_LED_OFF()` – turn off LED.

Flowcharts

Following are flowcharts that explain `bootloader()` and `Boot_PerformWrite()` functions.

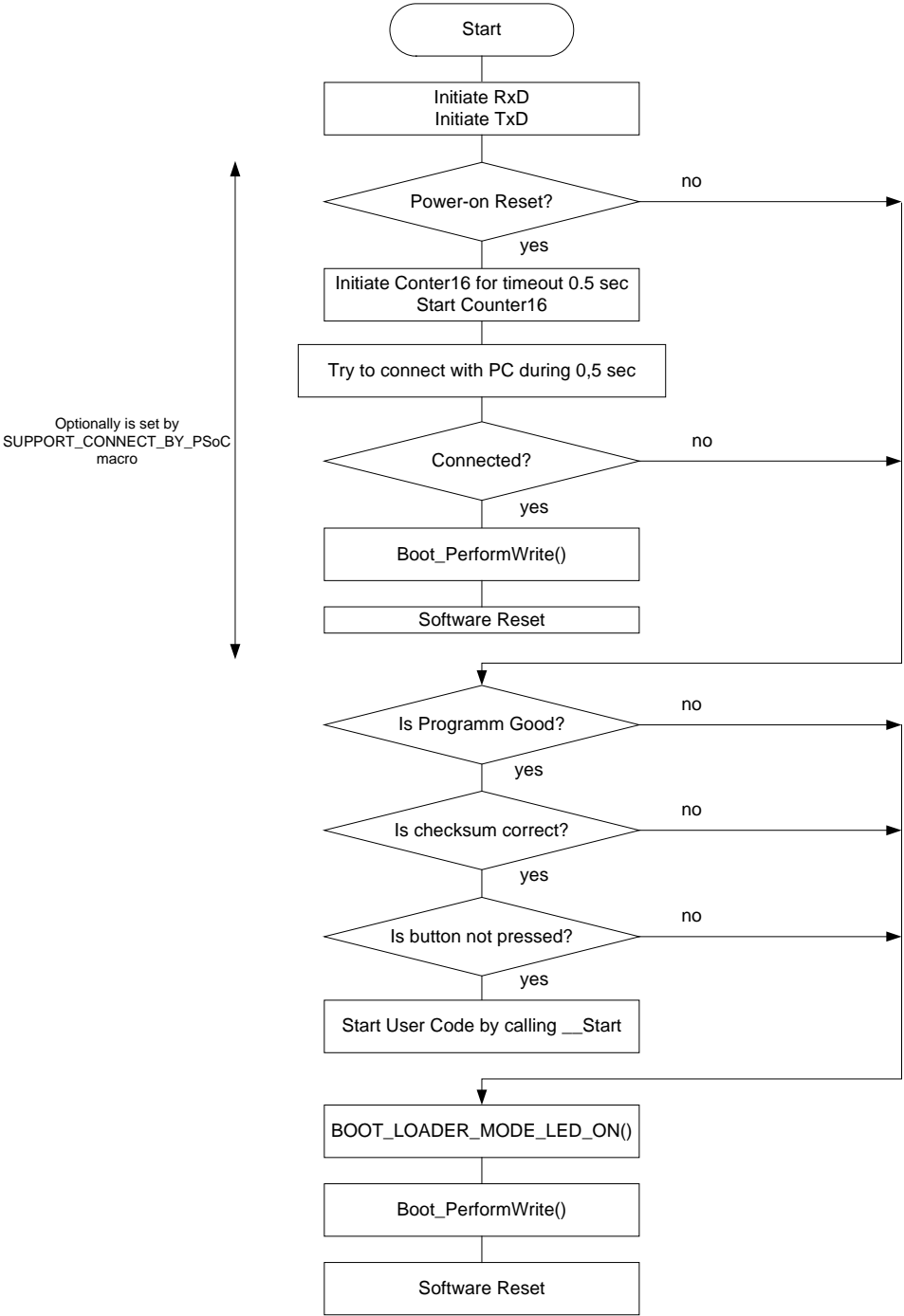


Figure 6. BootLoader() Function Flowchart

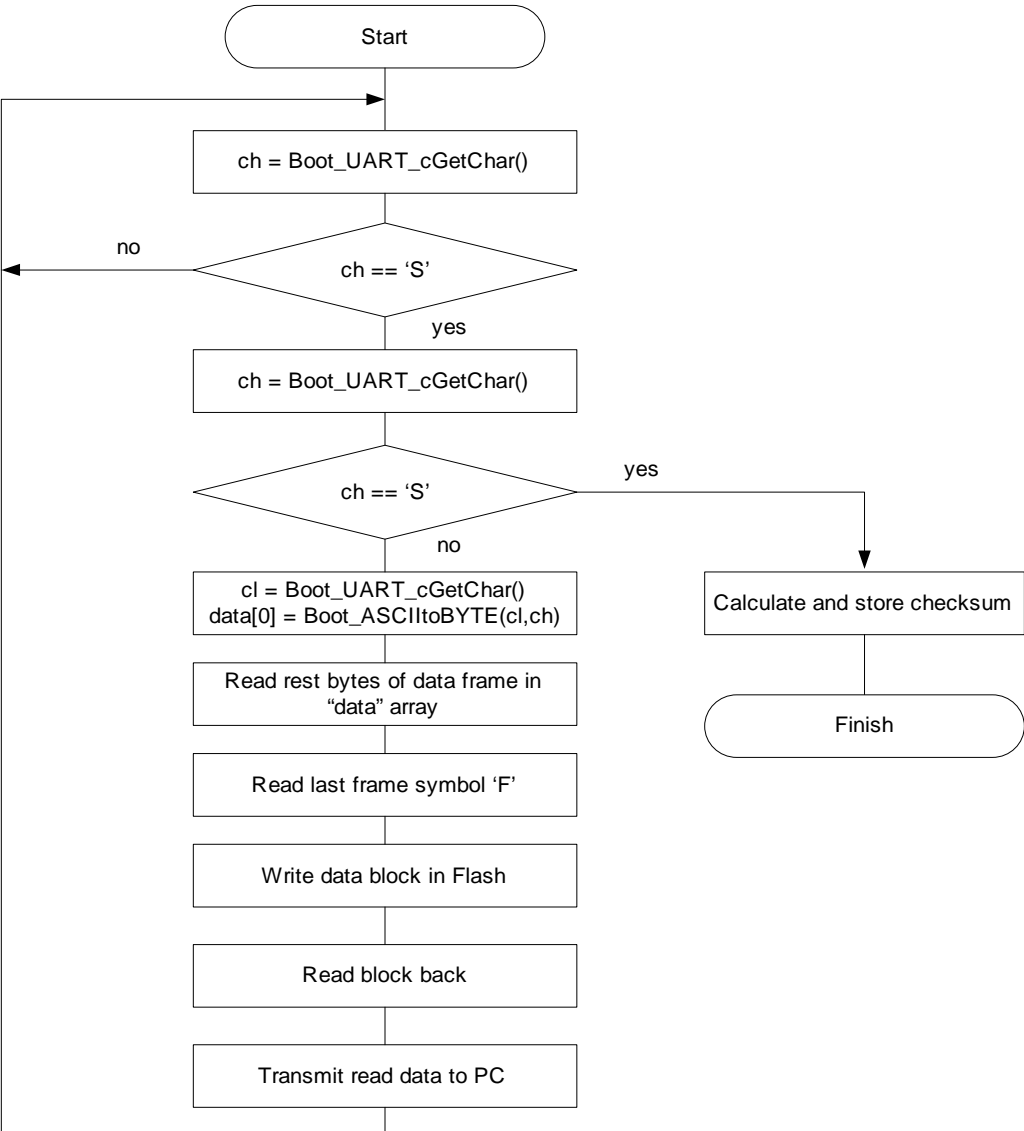


Figure 7. Boot_PerformWrite() Function Flowchart

PC Terminal Program

The terminal program was developed to facilitate the user reprogramming process. This program was released using C++ Builder tools. The program's functions are explained below and its main menu is shown in Figure 8.

CONNECT – connects with the PSoC if the device is already in bootloader mode.

Select HEX File – selects the HEX file of the project, which has to be loaded in PSoC. This means that only the name of the file is read, not its contents.

Program Device – starts programming.

About – short information about the program.

EXIT – close program.

If you check “Wait for connection with PSoC,” the program will be waiting until the PSoC device initiate communication after power-on-reset.

The working process of the program is well documented by narrations in the program window. You can see connection status and the selected file as well as watch the programming process using the progress bar. All invalid user actions are prevented and accompanied by warning messages.

This program verifies correct Flash write operations by receiving from the device written Flash blocks and comparing them with each source block. If there are any errors, the program tries to repeat block programming up to three times. In cases of failure, the program informs the user about programming fault and disconnects.

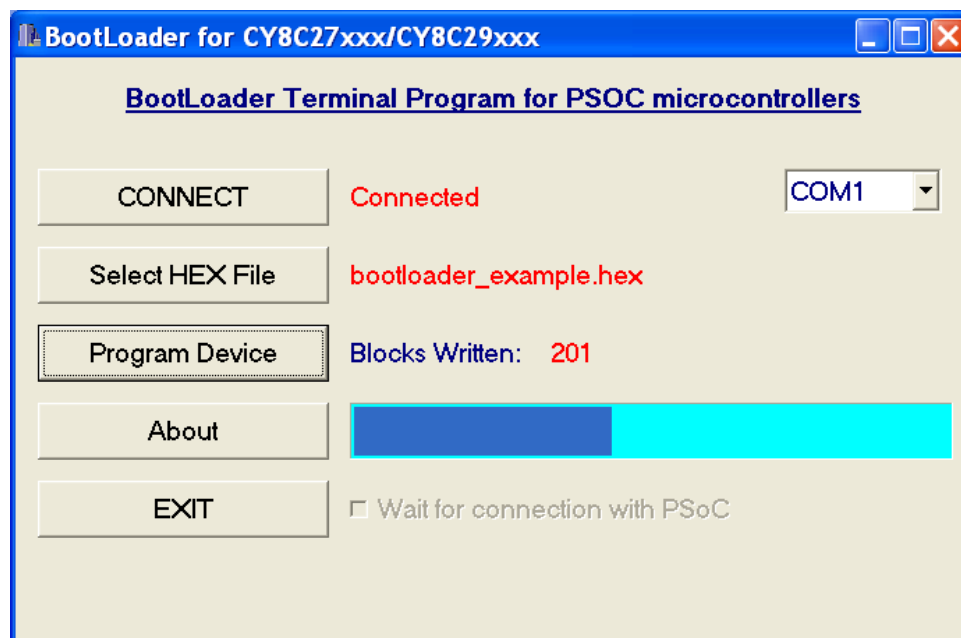


Figure 8. Example PC Terminal Program Window

Bootloader Usage

You can find the following directories under `\BootLoader_Delivarables`:

- *Bootloader* – project intended to create customized bootloader code.
- *Bootloader_Config* – project with configuration of bootloader where configuration changes can be done.
- *Bootloader_Example* – project that demonstrates how to use the bootloader.

1. Creating a bootloader-based project.

Sequence of operations:

- 1.1. Start PSoC Designer.
- 1.2. Select *Clone...* from *File >> New Project*.
- 1.3. In the New Project dialog box, enter new project name and its location. Click Next.
- 1.4. In the next window specify the existing bootloader project path, which is *BootLoader* directory under *BootLoader_Delivarables*. Click Finish.
- 1.5. Follow the procedure explained in 3.1 to 3.11 to configure the hardware portion of the bootloader project.
- 1.6. Place the user modules and develop your program as usual.

2. How to add bootloader to an existing project.

Let's say we have a project named *Existing_Project* and the *BootLoader* project. To add the behavior of the *BootLoader* into *Existing_Project* follow this sequence:

- 2.1. Select the following files from the *BootLoader* project: *boot.tpl*, *bootloader.c*, *flashapi.asm*, *bootloaderconfig.asm*, *custom.lkp*, *flashsecurity.txt*.
- 2.2. Copy these files to the *Existing_Project* root directory and replace existing files.
- 2.3. Select the following files from the *BootLoader/lib* directory: *bootloader.h*, *bootloader.inc*.
- 2.4. Copy these files to the *Existing_Project/lib* directory.
- 2.5. Open *Existing_Project* in PSoC Designer.
- 2.6. Add copied files to your project: *bootloader.c*, *flashapi.asm*, *bootloaderconfig.asm*, *bootloader.h*, *bootloader.inc*.

- 2.7. Click Generate Application icon. Now you can use bootloader features in your application.
- 2.8. Follow procedure 3.1 to 3.11 to configure the hardware portion of the bootloader project.

3. Bootloader pins and configuration redefinition.

In some applications it may be difficult or impossible to use hardwired pins (Rx, Tx, LED and press-button pins) established in the bootloader project. For this reason, the ability to redefine bootloader pins has been provided. In the *BootLoader_Delivarables* directory you can find project *BootLoader_Config*, which is to be used to create a new bootloader configuration. The sequence of operations is shown below:

- 3.1. Open project *BootLoader_Config* and modify its configuration and redefine pins, as you desire. Route the Rx and Tx pins and set the drive mode of the pin where the button is connected to StdCPU PullDown. Set the drive mode of the pin where the LED is connected to StdCPU Strong. Generate Application.
- 3.2. Open *psocconfigtbl.asm* in Application Editor.
- 3.3. Locate the following label: "LoadConfigTBL_BootLoaderConfig_Ordered:"
- 3.4. Select text starting from this position to the end of the file and copy it to the clipboard ([Ctrl]+C).
- 3.5. Close this project and open your bootloader project, where the pins have to be modified.
- 3.6. Open *bootloaderconfig.asm* and locate the following label: "LoadConfigTBL_BootLoaderConfig_Ordered:"
- 3.7. Select text starting from here to the end of "LoadConfigTBL_BootLoaderConfig_Bank1" register definition. Delete it.
- 3.8. Paste text from clipboard here ([Ctrl]+V).
- 3.9. Open *bootloader.c* in your program and modify the following macros according to the changes done in your configuration:

```
GETBUTTON()
BOOT_LOADER_MODE_LED_ON()
BOOT_LOADER_MODE_LED_OFF()
```

- 3.10. Compile and program the code to the PSoC.
- 3.11. Now your project is set to use the newly redefined pins in bootboader mode.

Special Considerations

1. Flash Checksum

If you have set macro `LAST_BLOCK_TO_CHECK` not equal to 0, the blocks from 1 to `LAST_BLOCK_TO_CHECK` will be protected by checksum. Afterwards, the reset `bootloader()` routine calculates the checksum for each given block and compares it with the corresponding checksum stored in Flash. When you use this feature and program the PSoC the first time through YProgrammer, your application will not be started after reset. It will enter bootloader mode. This occurs because the checksum for each block has not yet been calculated. The checksum feature works correctly only after programming the part through the terminal program. So you should run the terminal program, open your project .hex file, and store it in Flash. If you do not use the checksum feature, set `LAST_BLOCK_TO_CHECK` to 0, and your application will work immediately after using the YProgrammer to write program in the chip.

2. **Entering bootloader Mode via Button**
 - a. Start terminal program.
 - b. Click the button in the bootloader application.
 - c. Turn on the supply.
 - d. You are now in bootloader mode as it waits for connection with the PC.
 - e. Click "CONNECT" in the terminal program.
 - f. Once the PC is connected to the PSoC, the bootloader code waits for programming to begin.
 - g. Select the .hex file and click Program.
 - h. After programming, a software reset will be initiated.
3. **Entering bootloader after power-on-reset**
 - a. Start terminal program.
 - b. Check "Wait for connection with PSoC."
 - c. Switch off power to PSoC and then switch the power on.
 - d. PSoC is connected to the PC and the user code can be programmed.
 - e. Select the .hex file and click Program.
 - f. After programming, a software reset occurs and your application starts.
4. **Updating Project**

Whenever you start using a newer version of PSoC Designer, during Project Update, the existing `boot.tpl` is moved to the backup directory and a new `boot.tpl` is created. So remember, if you do a project update, execute the following:

 - a. Open the old `boot.tpl` from the backup directory.
 - b. Copy the contents from "export __Boot_Start" to the end of the file.
 - c. Replace the content of the new `boot.tpl` with the copied text below the export directives.
 - d. Save the `boot.tpl`.
 - e. Generate application.

About the Author

Name: Andrew Smetana
Title: Electronic Engineer
Education: Andrew earned Master of Science diploma in 2004 from National University "Lviv Polytechnic"(Ukraine). His interests involve various aspects of embedded systems development.
Contact: smetana@ukrwest.net

Cypress MicroSystems, Inc.
2700 162nd Street SW, Building D
Lynnwood, WA 98037
Phone: 800.669.0557
Fax: 425.787.4641

<http://www.cypress.com/> / <http://www.cypress.com/support/mysupport.cfm>

Copyright © 2004-2005 Cypress MicroSystems, Inc. All rights reserved.

PSoC™, Programmable System-on-Chip™, and PSoC Designer™ are trademarks of Cypress MicroSystems, Inc.

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

The information contained herein is subject to change without notice. Made in the U.S.A.