AN2100

Bootloader: PSoC® 1

Author: Andrew Smetana
Associated Project: Yes
Associated Part Family: CY8C21x34, CY8C27xxx, CY8C29xxx
Software Version: PSoC Designer™ 5.4 CP1 or later
Related Application Notes: For a complete list of the application notes, click here.

AN2100 describes a bootloader that uses the PSoC® 1 self-programming capability to allow users to reprogram the user flash memory through a UART interface. A dedicated Windows application is developed to simplify PSoC 1 programming through the bootloader.

Contents

What Is a Bootloader? 1
- Using a Bootloader 2
- Bootloader Using PSoC 1 2
- Hardware Implementation 2

Firmware Implementation 6
- boot.asm 6
- bootloaderconfig.asm 6
- flashapi.asm 6
- Macro Definitions 7

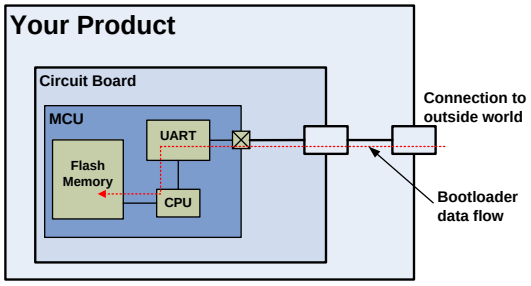
Flow Charts 8
PC Terminal Program 11
Bootloader Use 11
Project Implementation 11
- Create a Bootloader-Based Project 11
- Add a Bootloader to an Existing Project 12
- Redefine Bootloader Pins and Configuration 12
- Special Considerations 12
- Enter Bootloader Mode Via Button 12
- Enter Bootloader Mode After Power-On Reset 13

Updating the Project 13
Summary 13
Related Application Notes 13
Worldwide Sales and Design Support 15

What Is a Bootloader?

Bootloaders are a common part of MCU system design. A bootloader makes it possible for a product's firmware to be updated in the field. In a typical product, the firmware is embedded in an MCU's flash memory. The MCU is mounted on a PCB and embedded in a product, as Figure 1 shows.

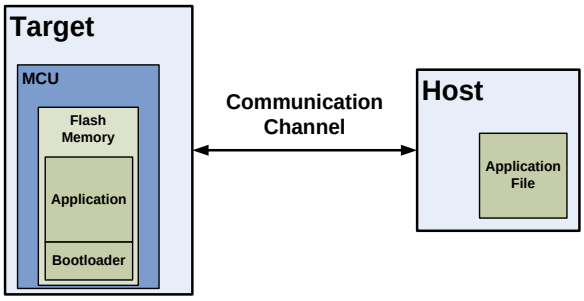
Figure 1. Bootloader Data Flow Block Diagram



At the factory, the initial programming of firmware into a product is typically done through the MCU's Joint Test Action Group (JTAG) or serial wire debug (SWD) interface. However, these interfaces are usually not available in the field, and it can be difficult and expensive to open up the product and directly access the PCB. A better method is to use an existing connection between the product and the outside world. That connection may be a common protocol such as I^2C, USB, or UART, or it may be a proprietary protocol.

Figure 1 shows that the product's embedded firmware must be able to use the communication port for two purposes: normal operation and updating flash. The portion of the embedded firmware that knows how to update the flash is called a "bootloader," as Figure 2 shows.

Figure 2. Bootloader System



Typically, the system that provides the data to update the flash is called the “host,” and the system being updated is called the “target.” The host can be an external PC or another MCU on the same PCB as the target.

The act of transferring data from the host to the target flash is called “bootloading,” a “bootload operation,” or “bootload” for short. The data that is placed in the flash is called the “application” or “bootloadable.”

Another common term in bootloading is “in-system programming (ISP).” Cypress uses a proprietary protocol with a similar name called “in-system serial programming (ISSP)” and an operation called “host-sourced serial programming (HSSP).” For more information, see AN44168.

Using a Bootloader

A communication port is typically shared between the bootloader and the application code. The first step in using a bootloader is to manipulate the product so that the bootloader, not the application, is executing.

Once the bootloader is running, the host can send a “start bootload” command over the communication channel. If the bootloader sends an “OK” response, bootloading can begin.

During bootloading, the host reads the file for the new application, parses it into flash write commands, and sends those commands to the bootloader. After the entire file is sent, the bootloader can pass control to the new application.

Bootloader Using PSoC 1

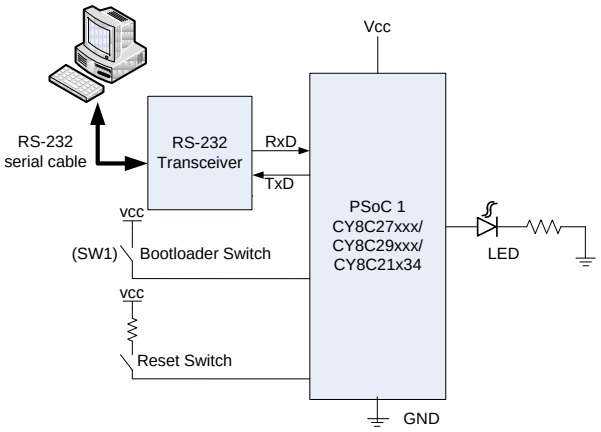
PSoC 1 devices are programmed after they are installed in a system. Although ISSP is the standard method used to program PSoC 1, a bootloader is an alternative. This method does not require the use of the Cypress ICE-Cube programmer or third-party programming tools. It requires only a serial cable, an RS-232 level translator, and a terminal program to modify the PSoC 1 firmware.

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

A system-level connection diagram for the PSoC 1 bootloader is shown in Figure 3. There are two ways to access the bootloader:

- When you press and hold the bootloader switch (SW1) at power-on reset, the PSoC 1 device goes into bootloader mode. Then you can start the PC terminal program and connect it with the PSoC 1 device for bootloading.
- When you select **Wait for connection with PSoC** in the PC terminal program and then reset the device, PSoC 1 interacts with the PC terminal program. Then the host communicates with the PSoC 1 device and the bootloader can be accessed.

Figure 3. System-Level Diagram for PSoC 1 Bootloader



A terminal program is developed for the PC to provide a simple user interface for the flash upgrade. You can choose any hexadecimal file and observe 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. This keeps you informed of the programming status or if any errors occur.

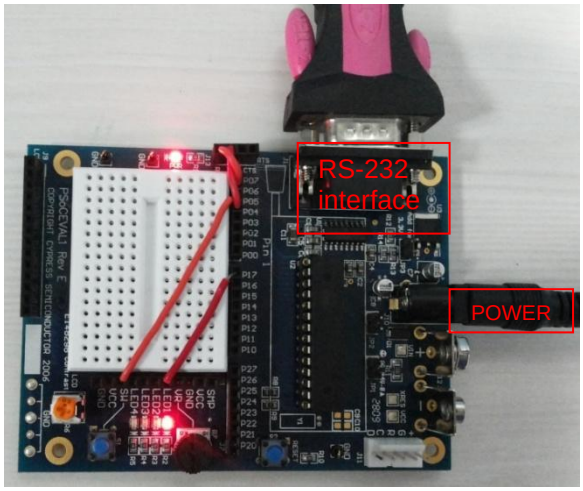
Hardware Implementation

The communication between the host PC and the PSoC 1 device is established using an RS-232 interface. The hardware connection for the bootloader on the CY3210-PSoCEVAL1 EVK is shown in Figure 4. Connect P0[6] to RxD, P0[4] to TxD, P0[5] to SW1, P1[7] to LED1, and an RS-232 serial cable to J1.

For the CY8C21x34 device, you need to use the CY3280-21x34 UCC board. To test the project, connect P0[6] and P0[4] on the UCC board to the RX and TX pins of an external RS-232 transceiver. Connect external switch SW1 to port P0[5] and the LED to P1[7], as shown in Figure 5.

Note You need to connect a current-limiting resistor in series with the LED.

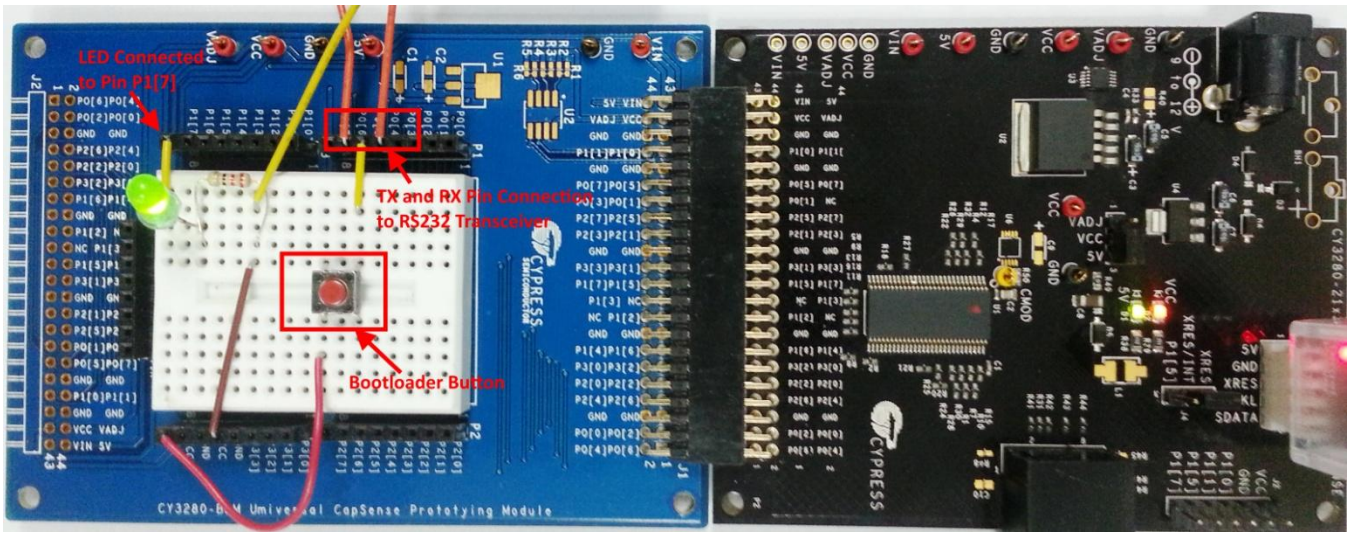
Figure 4. Hardware Connection on CY3210-PSoCEVAL1



A MAX232 (RS-232 interface chip from Maxim Integrated Products) is used as a level shifter to translate the TxD and RxD signals for the host PC. A button is needed to enter the bootloader mode. The LED is turned on when the bootloader mode is entered. During the programming process, the LED blinks.

A serial receiver and transmitter implements the UART interface, operating at 115200-baud rate.

Figure 5. Hardware Connection on CY3280 UCC Kit for CY8C21x34



The clock for these modules comes from the VC3 source. A Counter16 user module implements the timeout function. When the controller tries to enter the 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 used to enter the bootloader mode.

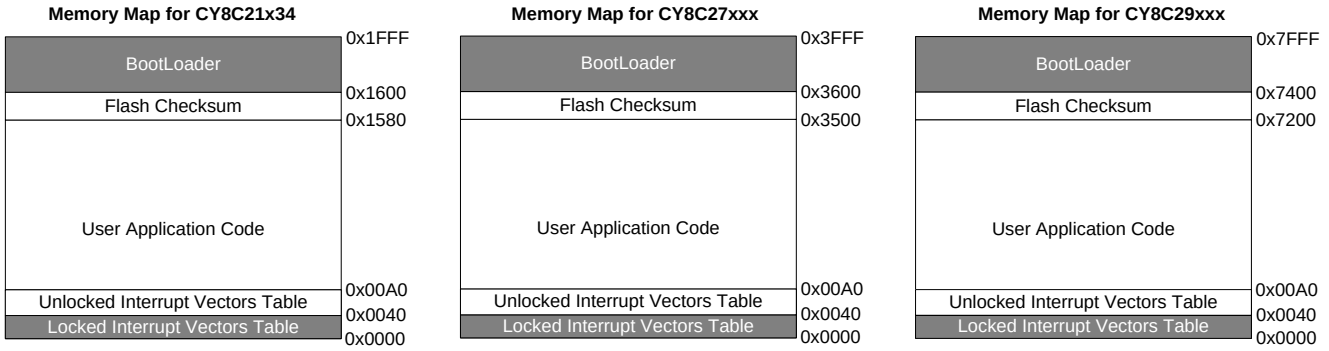
The memory space of a typical bootloader program is shown in Figure 6, in which fully protected blocks are shaded and unprotected blocks are not.

The high address space beginning at the following address for each device is intended to contain the bootloader firmware:

- **CY8C21x34:** 0x1600-0x1FFF
- **CY8C27xxx:** 0x3600-0x3FFF
- **CY8C29xxx:** 0x7400-0x7FFF

These memory areas are protected from write and erase operations.

Figure 6. Bootloader Memory Space



In the PSoC 1 device, 64 bytes in the flash memory constitute a block. The first flash block, 0x0000 to 0x003F, is fully protected. The reset vector jumps directly to the boot-verify code that is part of the bootloader. This ensures that the bootloader code is always available regardless of aborted bootloading attempts or incorrect user code.

Because the first 64 bytes, 0x0000 to 0x003F, of flash are fully protected, changing addresses in the first 15 interrupt routines is not possible. To change interrupt vector addresses, the locked user interrupts, up to 0x003F**,** jump to an unlocked table in *boot.asm*. All changes to *boot.asm* are done in the template *boot.tpl*, because after each application generation, the content of the *boot.tpl* file is copied to *boot.asm*.

The address 0x00A0 is the first address after the unlocked interrupt vector table. To verify the loaded project, the bootloader uses the following byte sequence for each device:

- **CY8C21x34:** 1, 2, 3, 4, 5, 6
- **CY8C27xxx:** 0, 1, 2, 3, 4, 5
- **CY8C29xxx:** 2, 3, 4, 5, 6, 7

If a user program is based on the bootloader, it must have these bytes written in the flash at the address 0x00A0. After this control byte sequence, all user program code and the __Start routine follow.

The memory space from the following address for each device exclusively stores the checksum information for each unprotected block:

- **CY8C21x34**: 0x1580-0x15FF
- **CY8C27xxx:** 0x3500-0x35FF
- **CY8C29xxx:** 0x7200-0x73FF

The checksum protects the blocks with IDs from 1 to 85 for CY8C21x34, 1 to 211 for CY8C27xxx, and 1 to 455 for CY8C29xxx. The block with the address 0x0040 to 0x00A0 has ID 1, the next 64 bytes (block 2) has ID 2, and so on. In the bootloader project, the last block to be checked is set at the following address for each device:

- **CY8C21x34:** 0x1580
- **CY8C27xxx:** 0x3500
- **CY8C29xxx:** 0x7200

If it is set to 0, then the checksum feature is turned off. Figure 7 illustrates how the bootloader checksum memory is used. The checksum feature provides a more reliable operation in the event of a soft fault and accidental flash rewrites because all user program code is unprotected from internal writes. The checksum is checked after each reset. If it is bad, the bootloader mode is automatically entered.

The protection type of each flash block is set in the *flashsecurity.txt* file (in the source tree of your PSoC Designer™ project). You should set “W” for fully protected blocks and “U” or “R” for unprotected blocks. It takes longer to do the flash writes (per block) using the in-circuit emulator (ICE) than on the chip. Test the bootloader on the chip to estimate the real performance. You can disconnect the pod from the ICE and run the program to see the upload speed for the real code. Figure 8 through Figure 10 show how the *flashsecurity.txt* file should be set for each device.

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

-bBootCheckSum:0x1580.0x15ff *-bBootLoaderArea:0x1600.0x1fff*	CY8C21x34
-bBootCheckSum:0x3500.0x35ff *-bBootLoaderArea:0x3600.0x3fff*	CY8C27xxx
-bBootCheckSum:0x7200.0x73ff *-bBootLoaderArea:0x7400.0x7fff*	CY8C29xxx

These records determine the borders of the bootloader and checksum segments. To learn more about this file, see the PSoC Designer ImageCraft C Compiler Guide.

Figure 7. Bootloader Checksum Memory Area

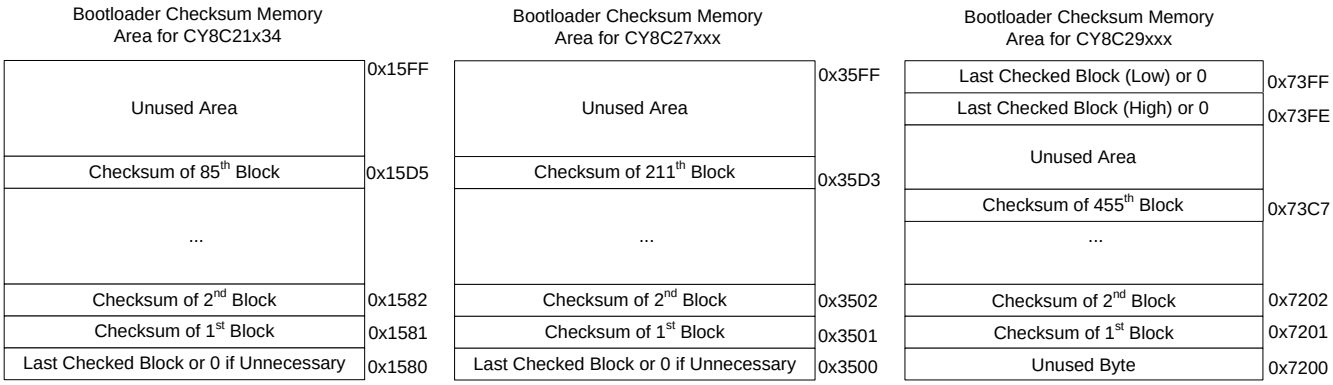


Figure 8. Flash Security Settings for CY8C21x34

```
; 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
  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   W   W   W   W   W   W   W   W ;    Base Address 1400
  W  W  W  W  W   W   W   W   W   W   W   W   W   W   W   W ;    Base Address 1800
  W  W  W  W  W   W   W   W   W   W   W   W   W   W   W   W ;    Base Address 1C00
; End 8K parts
```

Figure 9. Flash Security Settings for 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   W   W   W   W   W   W   W   W ;    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
```

Figure 10. Flash Security Settings for CY8C29xxx

```
; 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
```

Firmware Implementation

Bootloader firmware implementation occurs in the following files:

- *boot.asm*
- *bootloaderconfig.asm*
- *flashapi.asm*
- *bootloader.c*

boot.asm

This is the project startup file. It reflects the locked and unlocked interrupt vector tables, boot control sequence (0, 1, 2, 3, 4, 5 for CY8C27xxx) placed at address 0x00A0, the `__Start` routine—the user application initialization procedure, and the `__Boot_Start` routine to which the program jumps when the first instruction is executed after reset. This routine initiates the device for the bootloader mode, sets the CPU clock equal to 12 MHz, sets the top of the stack, loads the user module configuration, and then calls the `BootLoader()` function. The `__Boot_Start` routine is allocated to the protected bootloader area (0x1600 to 0x1FFF for CY8C21x34, 0x3600 to 0x3FFF for CY8C27xxx, and 0x7400 to 0x7FFF for CY8C29xxx).

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

bootloaderconfig.asm

This file contains the user module configurations and APIs. It includes the serial receiver and transmitter user modules and a timeout counter module. All module API names start with “Boot_” to minimize confusion with the 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 the bootloader operation.

flashapi.asm

This file implements the functions to handle the flash memory operations. Among these functions are the following:

- `bflashWriteBlock`: Executes the flash block write action
- `FlashReadBlock`: Reads one block of flash
- `FlashCheckSum`: Calculates the flash block checksum

`Boot_Is_Program_Good` finds out whether 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 the following value for the specific device, then the function returns an error result.

- **CY8C21x34:** 1, 2, 3, 4, 5, 6
- **CY8C27xxx:** 0, 1, 2, 3, 4, 5
- **CY8C29xxx:** 2, 3, 4, 5, 6, 7

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

bootloader.c

bootloader.c is the main file in the bootloader that contains all the functions required for boot-verify operations. It makes the connection with the PC host and functions that perform data transmission with the PC during the flash programming process.

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

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

If the previous two steps are completed successfully, the boot-verify action is performed. If verified, the program enters the bootloader mode. Otherwise, it calls the `__Start` routine to start user program execution.

Once the program is in the bootloader mode, it waits for a communication from the PC. Afterward, `Boot_PerformWrite()` is called to complete the flash programming operations. If this stage is completed successfully, the program executes a software reset using the supervisory code.

`void Boot_PerformWrite()` obtains the flash blocks from the PC in Intel HEX format, writes them in the flash, reads them back, and sends them to the PC for comparison with the 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 contains the following records:

- Length of data to be written (1 byte)
- Starting address (2 bytes)
- Type of data (1 byte)
- Immediate data block (64 bytes)

All fields of data represent information in ASCII-encoded hexadecimal. This means that every 8 bits of information is encoded into 2 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, as shown in Figure 7.

`char Boot_ASCIItoBYTE(char low, char high)` translates the ASCII-encoded byte representation into a binary format.

The following functions are high-level UART APIs:

- `BYTE Boot_UART_cGetChar(void)` reads a byte from RxD and blocks program execution if the buffer is empty.
- `void Boot_UART_PutChar(char TxDData)` sends a character to the TxD buffer and blocks program execution if the buffer is not empty.
- `void Boot_UART_CputString()` sends the ASCII string out of the TxD port.

Macro Definitions

```
#define LAST_BLOCK_TO_CHECK  211
```

The checksum protects the flash block from 1 to the value specified by the macro LAST_BLOCK_TO_CHECK. The checksum block distribution is shown in Figure 11 to Figure 13. 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 the 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

Figure 11. Checksum Blocks for CY8C21x34

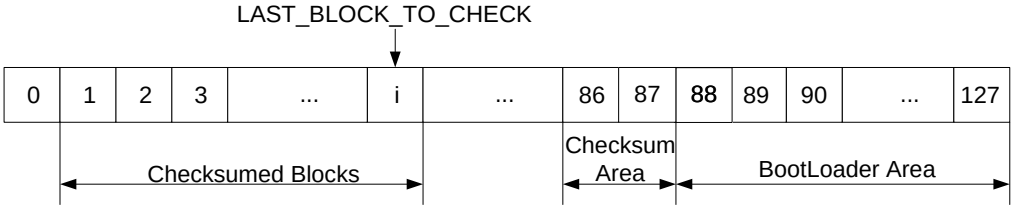


Figure 12. Checksum Blocks for CY8C27xxx

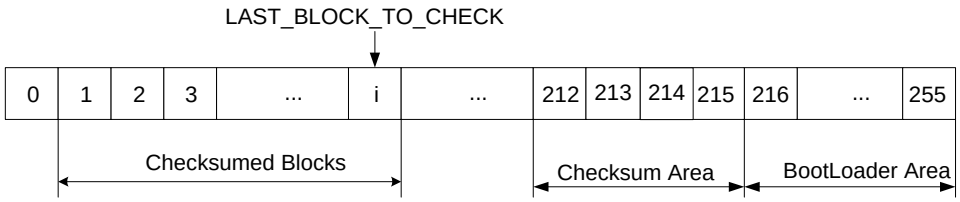
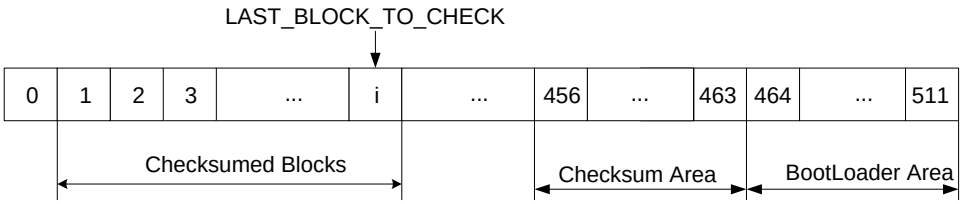


Figure 13. Checksum Blocks for CY8C29xxx



Flow Charts

Figure 14 depicts the flow chart for the `bootloader()function`, and Figure 15 depicts the flow chart for the `Boot_PerformWrite()` function.

Figure 14. BootLoader() Function Flow Chart

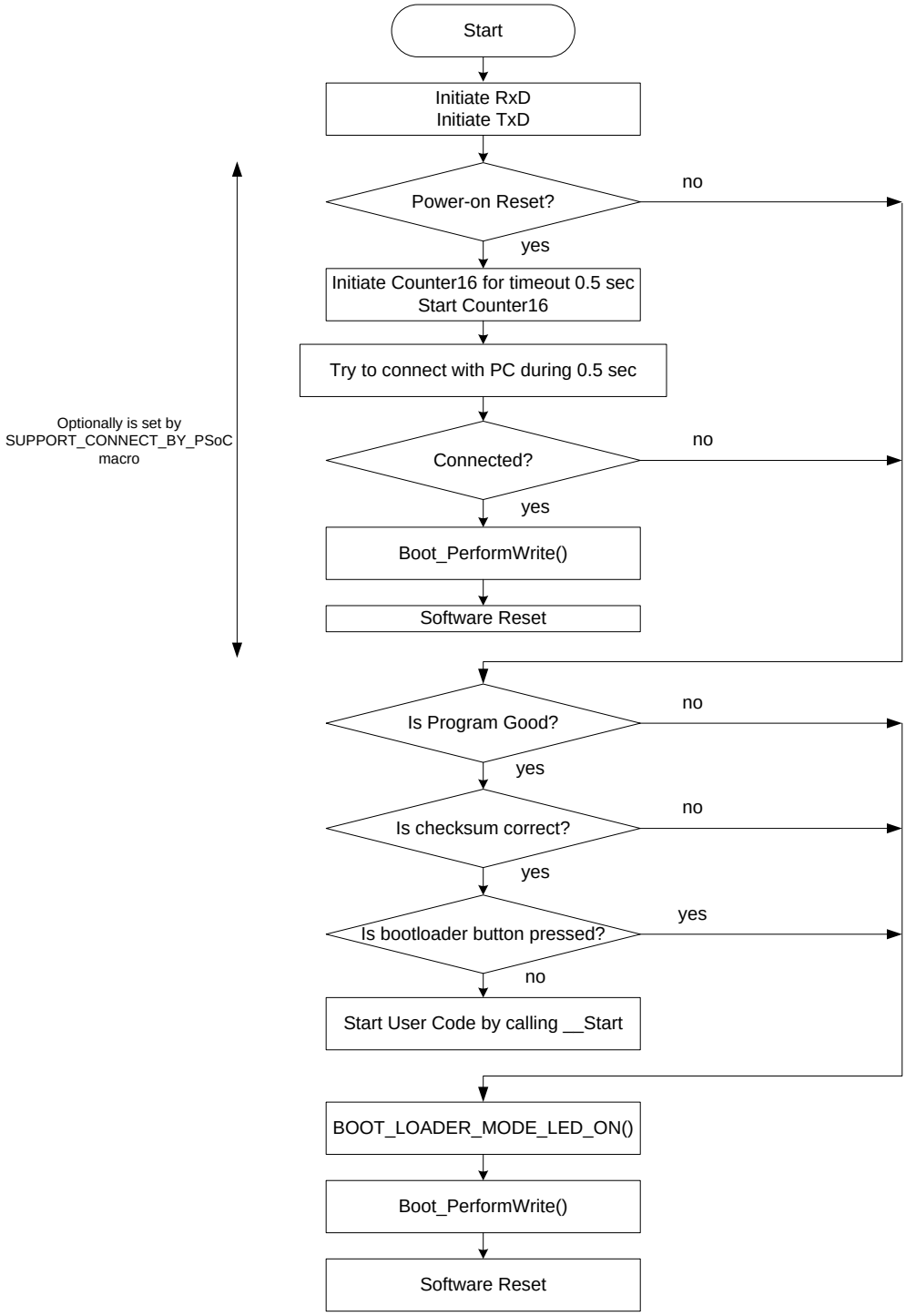
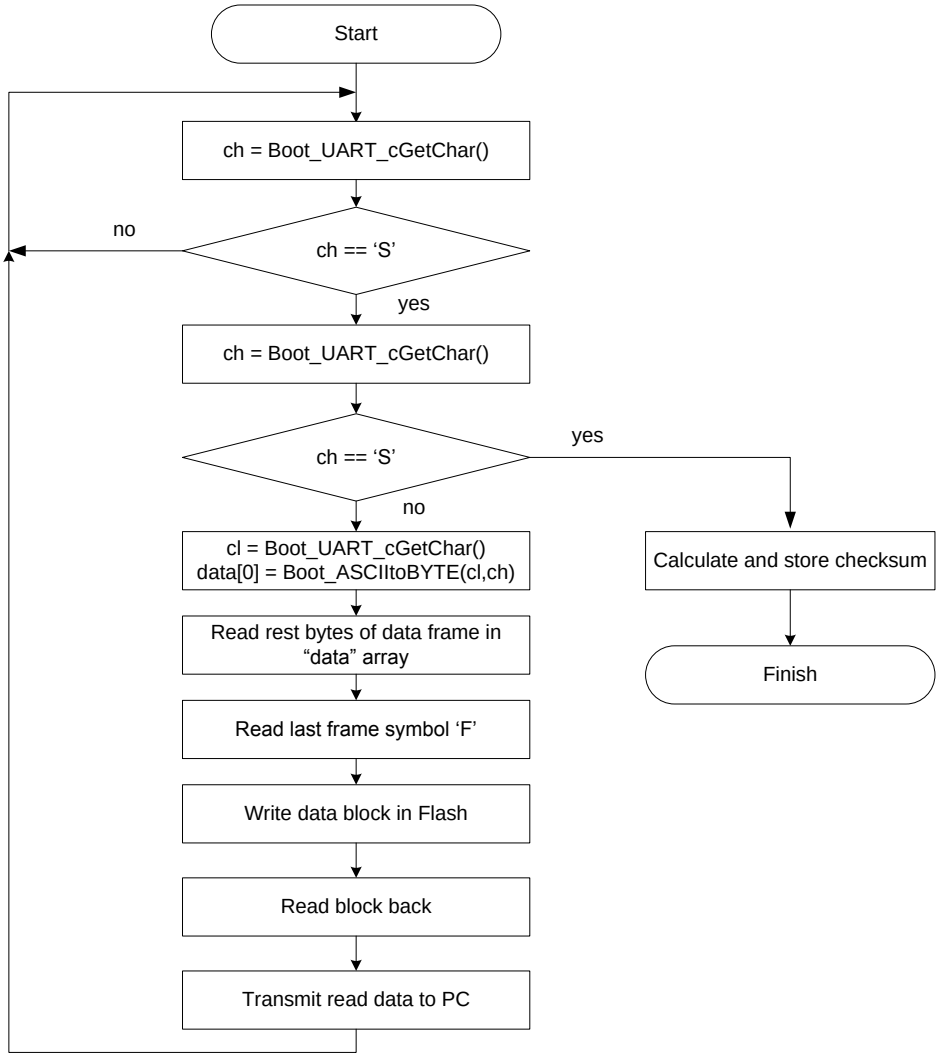


Figure 15. Boot_PerformWrite() Function Flow Chart



PC Terminal Program

The PC terminal program is developed to facilitate the user reprogramming process. This program is released using C++ Builder tools. The program's main menu is shown in Figure 16. The following list explains the program's functions.

- COM port drop-down list: Select the COM port corresponding to the UART communication port.

  Note Make sure that the COM port number is in the range COM1 to COM10.

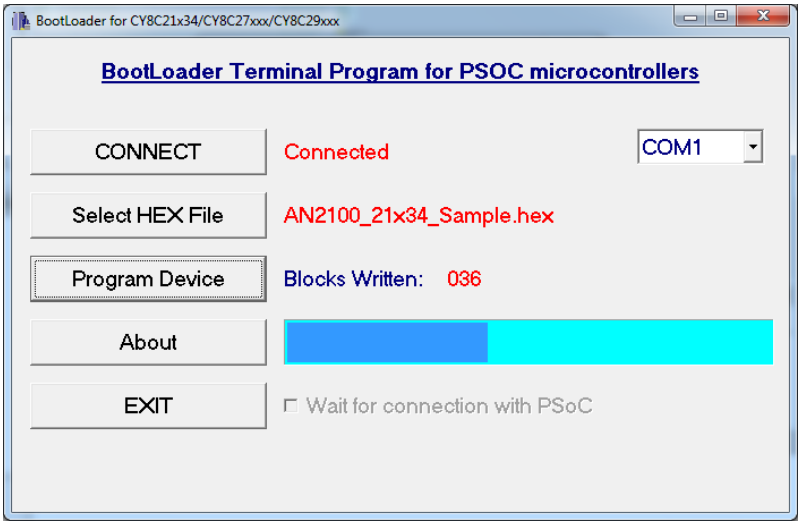
- **CONNECT:** Connects with the PSoC 1 device if it is already in bootloader mode.
- **Select HEX File:** Selects the hex file of the project, which is loaded in PSoC 1. This means that only the name of the file is read, not its contents.
- **Program Device:** Starts programming.
- **About:** Presents brief information about the program.
- **EXIT:** Closes the program.

If you select **Wait for connection with PSoC**, the program waits until the PSoC 1 device initiates communication after power-on reset.

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

The PC terminal program verifies the correct flash write operations by receiving data from the device's 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 the event of failure, the program informs you about the programming fault and disconnects.

Figure 16. Example PC Terminal Program Window



Bootloader Use

- *AN2100_2x_Loader:* Bootloader source code.
- *AN2100_2x_Sample:* Sample of bootloader-based application.
- *AN2100_2x_Config:* Project for bootloader pins and configuration redefinition. See Redefine Bootloader Pins and Configuration.

Project Implementation

Create a Bootloader-Based Project

1. Start PSoC Designer.
2. In the **New Project** dialog box, enter the new project name and its location. Click **OK**.
3. Place the user modules and develop your program as usual.

Add a Bootloader to an Existing Project

1. Select the following files from the BootLoader project folder location:
   - ❑ *boot.tpl*
   - ❑ *bootloader.c*
   - ❑ *flashapi.asm*
   - ❑ *bootloaderconfig.asm*
   - ❑ *BLconf.h*
   - ❑ *custom.lkp*
   - ❑ *HTLinkOpts.lkp*
   - ❑ *flashsecurity.txt*
2. Copy these files to the Existing_Project directory and replace the existing files.
3. Select the following files from the BootLoader/lib directory: *bootloader.h, bootloader.inc.*
4. Copy these files to the Existing_Project/lib directory.
5. Open Existing_Project in PSoC Designer.
6. Add the copied files to your project:
   - ❑ *bootloader.c*
   - ❑ *flashapi.asm*
   - ❑ *bootloaderconfig.asm*
   - ❑ *BLconf.h*
   - ❑ *bootloader.h*
   - ❑ *bootloader.inc*
7. Click **Generate Application**. Now you can use the bootloader features in your application.

Redefine Bootloader Pins and Configuration

In some applications, it is difficult or impossible to use the hardwired pins (RxD, TxD, LED, and press-button pins) established in the bootloader project. For this reason, the ability to redefine the bootloader pins is provided. In the BootLoader_Deliverables directory, you can find project BootLoader_Config, which is used to create a new bootloader configuration.

1. Open project *BootLoader_Config,* modify its configuration, and redefine the pins as required.
   a. Route the RxD and TxD pins, and set the drive mode of the pin where the button is connected to StdCPU PullDown.
   b. Set the drive mode of the pin where the LED is connected to StdCPU Strong.
2. Generate the application.
3. Copy the chip configuration from file *psocconfigtbl.asm* (AN2100_2x_Config project) to file *bootloaderconfig.asm* (AN2100_2x_Loader project):
   - ❑ Copy the content of Config_Ordered.
   - ❑ Copy the content of Config_Bank0.
   - ❑ Copy the content of Config_Bank1.
4. Open the *BLconf.h* file 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()
5. Compile and program the code to the PSoC 1 device.

Now your project is set to use the newly redefined pins in bootloader mode.

Special Considerations

Flash Checksum

If you set macro LAST_BLOCK_TO_CHECK not equal to 0, the blocks from 1 to LAST_BLOCK_TO_CHECK are protected by the 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 1 device the first time through MiniProg, your application will not start after reset. Instead, it enters the bootloader mode because the checksum for each block is not yet 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, you must set the macro LAST_BLOCK_TO_CHECK equal to 0.

Your application works immediately after using PSoC Programmer to write program in the chip.

Enter Bootloader Mode Via Button

1. Start the terminal program.
2. Click the button in the bootloader application.
3. Turn on the supply.
4. The device is now in bootloader mode, as it waits for connection with the PC.
5. Click **CONNECT** in the terminal program.
6. Once the PC is connected to the PSoC 1 device, the bootloader code waits for programming to begin.
7. Select the *.hex* file and click **Program**.
8. After programming, a software reset will be initiated.

Enter Bootloader Mode After Power-On Reset

1. Start the PC terminal program.
2. Select the **Wait for connection with PSoC** option.
3. Switch off power to PSoC 1 and then switch it back on.
4. PSoC 1 is connected to the PC, and the user code is programmed.
5. Select the *.hex* file and click **Program**.
6. After programming, a software reset occurs, and your application starts.

Updating the 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. If you do a project update, perform the following steps:

1. Open the old *boot.tpl* from the backup directory.
2. Copy the contents from “export __Boot_Start” to the end of the file.
3. Replace the content of the new *boot.tpl* with the copied text to the export directives.
4. Save the *boot.tpl*.
5. Generate the application.

Summary

This application note provided a basic overview of bootloaders—their use and important design considerations. It also described how the PSoC Designer development environment addresses these considerations for PSoC 1 devices.

The document also discussed how to use PSoC Designer to quickly and easily add a bootloader to your design. For more detailed information about bootloaders, see Related Application Notes.

Related Application Notes

- AN60317– PSoC 3 and PSoC 5LP I^2C Bootloader
- AN86526 – PSoC 4 I^2C Bootloader
- AN73503 – USB HID Bootloader for PSoC 3 and PSoC 5LP
- AN68272 – PSoC 3, PSoC 4 and PSoC 5LP UART Bootloader
- AN84401 – PSoC 3 and PSoC 5LP SPI Bootloader
- AN44168 – PSoC 1 Device Programming Using an External Microcontroller (HSSP)
- AN73054 – PSoC 3/PSoC 5LP Programming Using an External Microcontroller (HSSP)
- AN75320 – Getting Started with PSoC 1
- AN54181 – Getting Started with PSoC 3
- AN79953 – Getting Started with PSoC 4
- AN77759 – Getting Started with PSoC 5LP

About the Author

Name:	Andrew Smetana
Title:	Electronic Engineer
Education:	Andrew earned a Master of Science diploma in 2004 from Lviv Polytechnic National University (Ukraine). His interests involve several aspects of embedded systems development.
Contact:	andi@cypress.com

Document History

Document Title: AN2100 – Bootloader: PSoC® 1

Document Number: 001-32904

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1494923	YIS	09/21/2007	New application note
*A	2936426	ANDI	05/24/2010	Updated cross reference
*B	3201039	BIOL_UKR	03/20/2011	Minor updates
*C	3291243	ANBI_UKR	06/23/2011	Updated software version.
*D	3340375	ANBI_UKR	08/09/2011	Updated software version and minor text edits. Updated project to support Hi-Tech compiler.
*E	4414552	ASRI	06/20/2014	Updated the introduction and included “What is a Bootloader?” section. Included system level and hardware connection diagrams for PSoC 1 bootloader Updated the projects to PSoC Designer 5.4 Included section Summary and Related Application Notes
*F	4700239	DCHE	04/08/2015	Added bootloader projects for CY8C21x34. Updated the terminal program to program the CY8C21x34 device. Updated the projects to PSoC Designer 5.4 CP1.
*G	5688098	AESATMP8	04/19/2017	Updated logo and Copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

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

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2007-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.