AN73854

PSoC® 3, PSoC 4, and PSoC 5LP Introduction to Bootloaders

Author: Mark Ainsworth

Associated Part Family: All PSoC 3, PSoC 4, and PSoC 5LP

Related Application Notes: For a complete list, click here.

To get the latest version of this application note, please visit http://www.cypress.com/AN73854.

AN73854 gives a brief introduction to bootloader theory and technology. It then shows how bootloaders are quickly and easily implemented in PSoC® 3, PSoC 4, and PSoC 5LP MCUs, using PSoC Creator™. Topics include bootloader system description, features, and customization.

Contents

1 Introduction

This application note gives an overview of bootloader fundamentals and design principles, and then shows how they are implemented for PSoC 3, PSoC 4, and PSoC 5LP in PSoC Creator projects.

If you are new to bootloaders in general, see the basic concepts and design principles explained in the sections What is a Bootloader? and General Bootloader Design Considerations.

If you are familiar with bootloaders and want to see how they are implemented for PSoC 3, PSoC 4, and PSoC 5LP devices using PSoC Creator, see the section PSoC Bootloader – How It Works.

To get an overview of adding a bootloader to your PSoC Creator project, see the section Add a Bootloader to Your PSoC Creator Project.

For a list of bootloader application notes related to I^2C, UART, SPI, and USB, refer to the section Related Application Notes. Each of the bootloader application notes listed in this section has associated code examples.

You can also access bootloader-related example projects from PSoC Creator using the menu option, **File** > **Example Project**. Search for "bootloader" in the pop-up window to filter the projects related to bootloader.

Click here for a complete list of PSoC 3, PSoC 4, and PSoC 5LP code examples.

Note: Beginning with PSoC Creator 2.1, the bootloader system has been reorganized to provide more configuration options. In previous releases, the bootloader system was part of the cy_boot Component (a required Component that is automatically and invisibly instantiated in all designs). Now, bootloader functions are available as separate Components. To migrate older versions of bootloader projects to PSoC Creator 2.1 SP1 or later, see Chapter 11 in the *System Reference Guide* (**Help** > **Documentation** > **System Reference**).

2 PSoC Resources

Cypress provides a wealth of data at www.cypress.com to help you to select the right PSoC device for your design and quickly and effectively integrate the device into your design. For a comprehensive list of resources, see KBA86521, How to Design with PSoC 3, PSoC 4, and PSoC 5LP. The following is an abbreviated list for PSoC x:

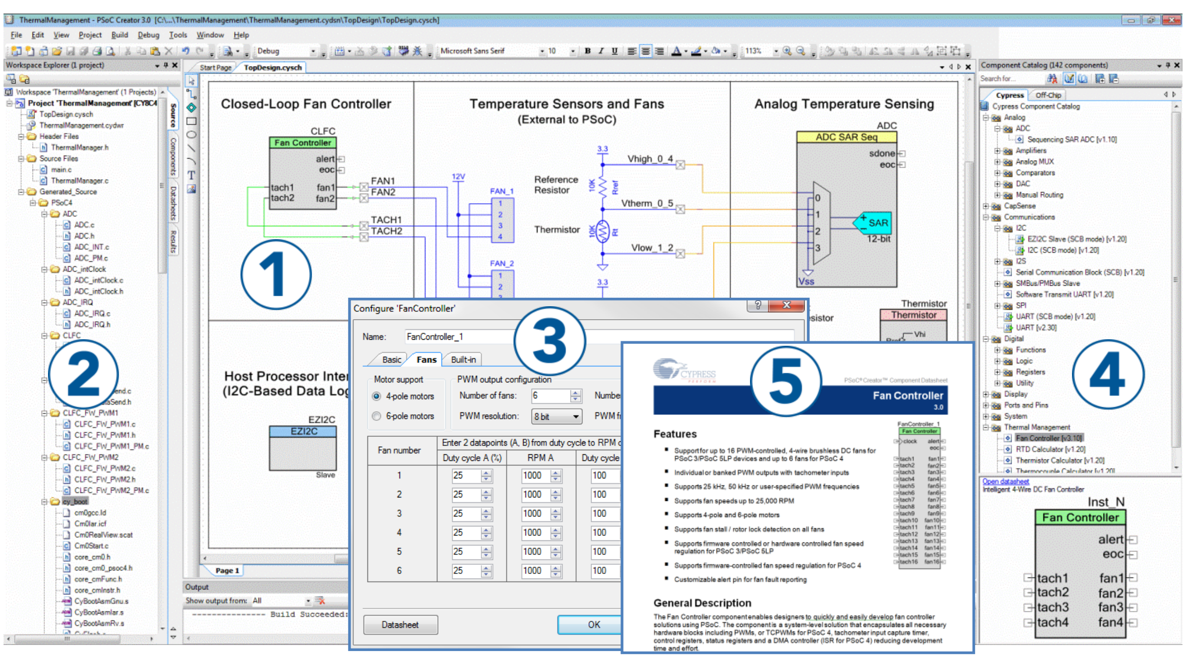
- **Overview:** PSoC Portfolio, PSoC Roadmap
- **Product Selectors:** PSoC 1, PSoC 3, PSoC 4, or PSoC 5LP. In addition, PSoC Creator includes a device selection tool.
- **Datasheets** describe and provide electrical specifications for the PSoC 3, PSoC 4, and PSoC 5LP device families.
- **Application Notes and Code Examples** cover a broad range of topics, from basic to advanced level. Many of the application notes include code examples.
- **Technical Reference Manuals (TRM)** provide detailed descriptions of the architecture and registers in each of the PSoC 3, PSoC 4, and PSoC 5LP device families.
- **Development Kits:**
  - CY8CKIT-001 is a common development platform for all PSoC family devices.
  - CY8CKIT-030 and CY8CKIT-050 are designed for analog performance. They enable you to evaluate, develop, and prototype high-precision analog, low-power, and low-voltage applications powered by PSoC 3 and PSoC 5LP, respectively.
  - CY8CKIT-046, CY8CKIT-044, CY8CKIT-042 and CY8CKIT-040, PSoC 4 Pioneer kits, are easy-to-use and inexpensive development platforms. These kits include connectors for Arduino™ compatible shields and Digilent® Pmod™ daughter cards.
  - The MiniProg3 device provides an interface for flash programming and debug.

3 PSoC Creator

PSoC Creator is a free Windows-based Integrated Design Environment (IDE). It enables concurrent hardware and firmware design of systems based on PSoC 3, PSoC 4, and PSoC 5LP. See Figure 1 – with PSoC Creator, you can:

1. Drag and drop Components to build your hardware system design in the main design workspace
2. Codesign your application firmware with the PSoC hardware
3. Configure Components using configuration tools
4. Explore the library of 100+ Components
5. Review Component datasheets

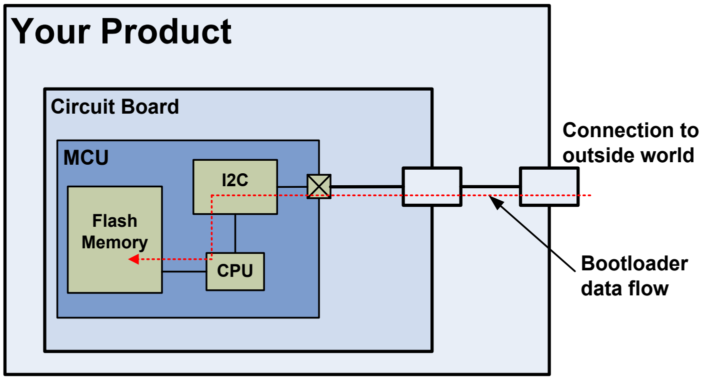
Figure 1. PSoC Creator Features



4 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, firmware is embedded in an MCU's flash memory. The MCU is mounted on a PCB and embedded in a product, as Figure 2 shows.

Figure 2. Bootloader Data Flow Block Diagram

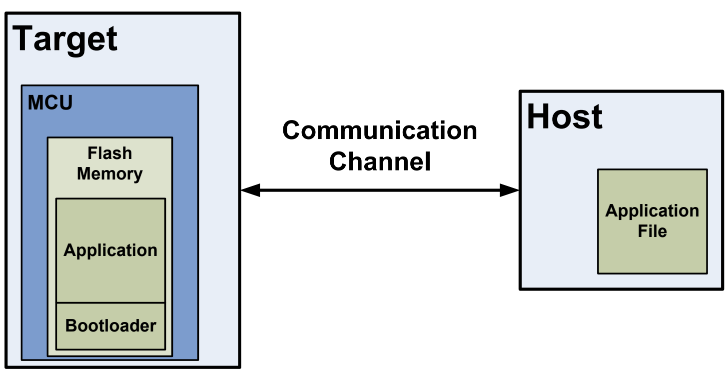


At the factory, initial programming of firmware into a product is typically done through the MCU's Joint Test Action Group (JTAG) or Serial Wire Debugger (SWD) interface. However, these interfaces are usually not available in the field – it can be difficult and expensive to open up the product and directly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

4.1 Terms and Definitions

Figure 2 shows that the product's embedded firmware must be able to use the communication port for two different purposes – normal operation and updating flash. That portion of the embedded firmware that knows how to update the flash is called a **bootloader**, as Figure 3 shows.

Figure 3. 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**, or a **bootload operation**, or just **bootload** for short. The data that is placed in flash is called the **application** or **bootloadable**.

Another common term for bootloading is **in-system programming (ISP)**. Cypress has a product with a similar name called In-System Serial Programmer (ISSP) and an operation called Host-Sourced Serial Programming (HSSP). For more information, see HSSP on page 6.

4.2 Using a Bootloader

A communication port is typically shared between the bootloader and the application. The first step to use a bootloader is to manipulate the product so that the bootloader, and 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.

4.3 Bootloader Function Flow

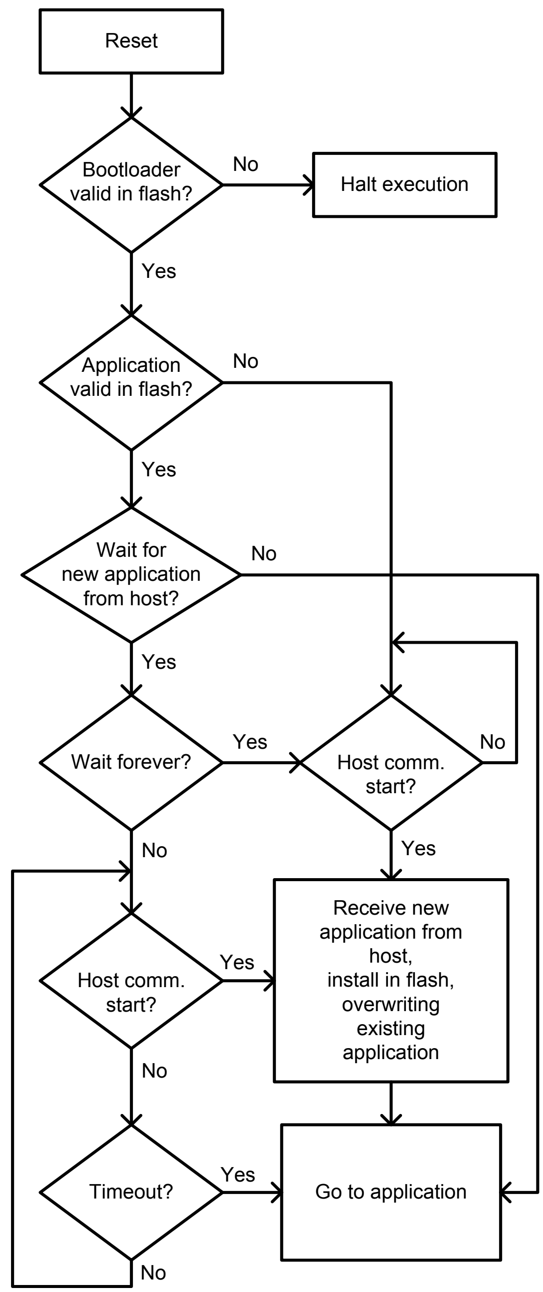
A bootloader typically executes first at reset (see Memory Use and Modular Configuration on page 6). It can then perform the following actions:

- Check the application’s validity before letting it run
- Manage the timing to start host communication
- Do the bootload / flash update operation
- And finally, pass control to the application

Figure 4 is a flow diagram that shows how this works.

Note: PSoC Creator supports a dual-application option, where the “Go to application” function in Figure 4 operates in a more complex fashion. For more information, see Application Launch Process.

Figure 4. Bootloader Function Flow



5 General Bootloader Design Considerations

There are many considerations to keep in mind when designing a bootloader system. In this section, we will look at some of them.

5.1 Bootloader Alternatives

As mentioned previously, a bootloader makes it possible for a product's firmware to be updated in the field.

There are other ways to solve this problem. For example, a flash update function can be coded within the application itself. However, the application must then be able to overwrite part or all of itself, which adds complexity. It is a better design practice to break the bootloader out as a separate module or program.

5.1.1 HSSP

Another alternative to having a bootloader is to use Host-Sourced Serial Programming (HSSP). In this method, the MCU's JTAG or SWD pins are directly manipulated by an external host to program a new application into flash. The Cypress CY8CKIT-002 MiniProg3 and other programmers use this method.

Although HSSP is commonly used during development and for factory-based programming, it is not usually used in the field. The most frequent use of HSSP in the field is on PCBs with multiple MCUs, where one MCU may directly program another MCU.

For details on accessing the PSoC JTAG/SWD pins, see the following:

- AN61290, PSoC 3 and PSoC 5LP Hardware Design Considerations
- AN88619, PSoC 4 Hardware Design Considerations
- PSoC 3 Programming Specifications
- PSoC 4 Programming Specifications
- PSoC 5LP Programming Specifications

For more information on HSSP, see the following:

- AN73054, PSoC 3 and PSoC 5LP Programming Using an External Microcontroller (HSSP)
- AN84858, PSoC 4 Programming Using an External Microcontroller (HSSP)

5.2 Memory Use and Modular Configuration

As noted previously, the bootloader code should be separate from application code – frequently they are designed as completely separate modules. Given that both modules must reside in flash, where should the bootloader code reside? Some MCUs contain a hard-coded bootload read-only memory (ROM) that is separate from flash, as Figure 5 shows. Other MCUs use a part of flash for the bootloader, as Figure 6 shows.

Figure 5. Bootloader in Separate ROM Memory

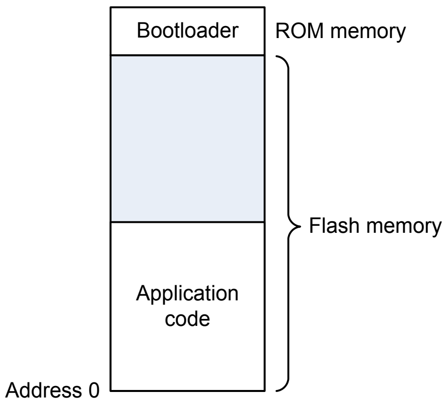
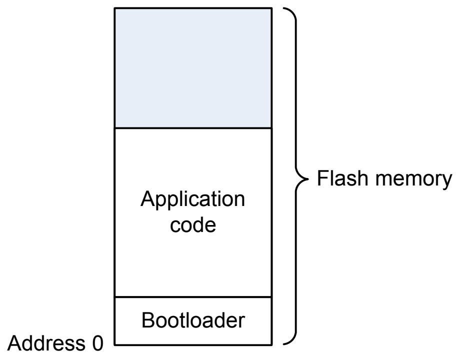


Figure 6. Bootloader in Flash Memory



The ROM method lets the application use all of flash, while the flash method allows a flexible bootloader design. The flash method is generally preferred. The bootloader is usually placed in flash starting at address 0. Because most CPUs start executing code at address 0, the bootloader runs first at device reset.

One potential problem with the flash method is that the bootloader uses memory that could otherwise be used by the application. This may impact application design, or cost if an MCU with more memory must be used. Thus, for the flash method, bootloaders should be designed to be as small as possible. To reduce size, keep the requirements and design simple; see Bootloader Memory Usage on page 10. Keeping the bootloader small may become difficult if additional features must be included; see Customization on page 8.

5.3 Bootloader - Host Timing

An important consideration in bootloader design is the timing to begin communication with the host. As Figure 4 shows, after determining that the application is valid, the bootloader can wait for a certain amount of time for the host to start a new bootload operation.

The wait time typically ranges from 50 to 500 msec. If the wait time is too short, the host may not be able to reliably start the communication. If it is too long, your product's overall startup time may be too long.

Another solution to the timing problem is to let the application call the bootloader. Then, the application can respond to some external event, such as a button press or a message from the host, and start the bootload operation.

5.4 Communication Port

In most cases, the specifications of the communication port shown in Figure 3 are set with regard to the overall product requirements. In addition to those requirements, for robust support of a bootloader system, the port should be able to do the following:

- Packet-based data transfers. The port should not parse the packets; the bootloader and host should do that.
- Packet error detection. The port should be able to detect and report packets with invalid data. The rest of the system should be able to handle invalid packet reports.
- Command-response protocol. Usually, the host sends a command packet to the bootloader and then waits for a success / fail / status response packet.
- Medium-speed transfers. Because it can take several milliseconds to write a single row of flash, having a high-speed port may not significantly improve overall bootload time.
- Transfers that can take place while flash is being written. This enables a row of data to be downloaded while the previous row is being written to flash.

Of all the commonly available protocols in embedded systems, USB supports these features best, although USB code can use a large amount of flash. UART, I^2C, and SPI are simpler but may require extra code for packet management. Note that I^2C is controlled solely by the master side (usually the host), which makes a command-response protocol more difficult to implement.

5.5 Recovering from Failures

A bootloader should be able to detect, report, and gracefully handle errors that occur during the bootload operation, such as power failure, loss of communication, and flash write error.

This is frequently done by storing in flash some check bits (checksum or CRC) for the application. When the bootload operation is started, the bits are cleared. If the application is downloaded and installed successfully, the bits are updated. If, for example, a power failure occurs during bootloading, at reset the bootloader detects invalid check bits and does not pass control to a partially loaded application. Instead, it waits for the host to start another bootload operation.

5.6 Future-Proofing

Another design consideration is that, after installation, a bootloader should never need to be updated in the field. It is possible to make a bootloader that can overwrite or update itself in the field, but it is complex and best avoided. The key to making a bootloader robust and future-proof is to keep the requirements and design simple. To avoid defects, use coding best practices and thorough code reviews.

Because the bootloader and application are separate modules, you can use different compilers or even different development systems to build them. Because tools such as compilers may change between versions, make sure that the mechanism to transfer control between the two modules remains constant. Also, as you upgrade your development tools over time, make sure that an old bootloader can still load new applications.

5.6.1 Application Management

Because the bootloader and the application are separate modules, and the application can change, you must consider how best to transfer control from the bootloader to the application. Some of the methods to do this are as follows:

- Jump to a fixed location where the application will always start. This method is simple but may be less flexible for future changes to the bootloader or application.
- Maintain the application start address in a common area of flash. The bootloader then uses that location as a pointer to the application start address.
- Link the application to a bootloader in a common development system, so that the bootloader has a symbolic address to jump to.

The second method has the best combination of simplicity and flexibility, and is usually the preferred solution.

5.6.2 Flash Protection

A bootloader should be able to check its own image in flash memory to see if it is valid. If it is not valid, it must stop executing. Unfortunately, this makes the product unusable.

The best way to keep the bootloader valid in flash is to use the hardware to make sure that the bootloader is never overwritten by firmware. One way to do this is to use flash write protection circuits that prevent accidental overwrites of bootloader flash. See Flash Protection on page 8 for PSoC implementation details.

5.7 Customization

Bootloaders should be designed such that they are easy to modify for different product applications. For example, a bootloader system should be able to easily use different communication ports, even multiple communication ports.

Also, a bootloader system may need to operate in a high-reliability product, which has three main aspects:

- There may be a need to preserve important pin states during the transition from the bootloader to the application. This can be a problem if the transition is done through a device reset. See Appendix A for details.
- Important tasks may need to be done at the same time as bootloading. Extra code may need to be added to the bootloader to enable a multitasking system. See Customization on page 11.
- Multiple (typically, two) application images are stored in flash. If one becomes corrupted in flash, the bootloader can pass control to the other image, reducing your product’s mean time between failure (MTBF). PSoC Creator supports dual-image bootloaders.

6 PSoC Bootloader – How It Works

In the previous sections, we looked at general bootloader functions and design considerations. Now let us see how these principles are put into practice in PSoC 3, PSoC 4, and PSoC 5LP, using the PSoC Creator integrated design environment (IDE).

PSoC devices have memory and configurable peripheral hardware that make it possible to create highly capable and flexible bootloader systems. Development is done using PSoC Creator, a free IDE provided by Cypress that is used to build PSoC-based solutions. For information on PSoC devices, see:

- AN54181, Getting Started with PSoC 3
- AN79953, Getting Started with PSoC 4
- AN77759, Getting Started with PSoC 5LP

For information on PSoC Creator, see the PSoC Creator home page.

Note: The PSoC 3, PSoC 4, and PSoC 5LP implementation of a bootloader system is different from that for PSoC 1. For more information on PSoC 1 bootloaders, see AN2100, Bootloader: PSoC 1.

As with all Cypress PSoC products and support IDEs, PSoC Creator attempts to reduce your design time by automating implementation of basic system functions. The bootloader is no exception – it can literally take just minutes to add a simple I^2C bootloader to your project. For information on how to do this, see Add a Bootloader to Your PSoC Creator Project.

6.1 PSoC Creator Bootloader Projects

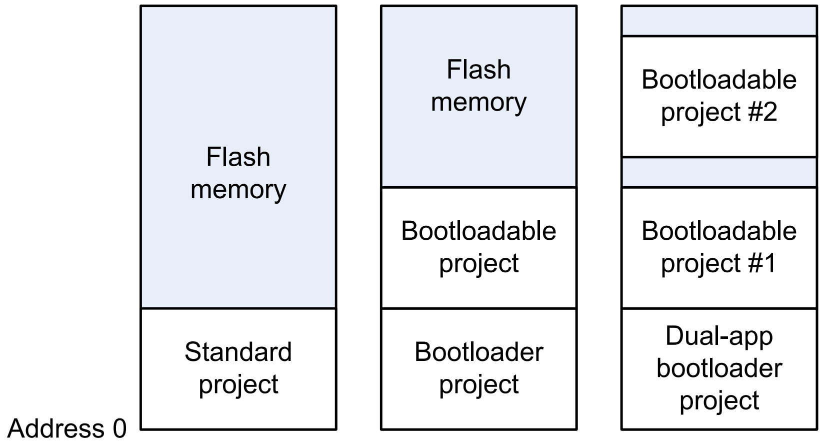
PSoC Creator uses the term “project” to define a complete, self-contained application. In addition to the CPU code, a project has data bytes that are used to configure the PSoC device’s analog and digital peripherals for your application.

It is important to remember that with PSoC Creator, bootloaders and applications are implemented in completely separate projects. Available project types are: **Standard** (or “normal,” no bootloader), **Bootloader**, and **Bootloadable**. A fourth project type, **Dual-App Bootloader**, supports dual application images for high-reliability applications as described in Customization. You can easily change a project type, for example, from standard to bootloadable.

You must associate a bootloadable project with a bootloader project. A bootloader project can be associated with multiple bootloadable projects.

Because PSoC has no bootload ROM, the bootloader is placed in flash, as Figure 7 shows. A bootloader project is placed in flash starting at address zero, and is executed first at device reset. It then implements the program flow shown in Figure 4.

Figure 7. PSoC Creator Projects and Flash Memory Usage



6.2 Bootloader Options

PSoC Creator provides a Bootloader Component, which has configuration options to set the run-time behavior of the bootloader. Some of the options are:

- Wait for Command: Yes or no to wait for a command from the host before passing control to the bootloadable.
- Wait for Command Time: Timeout from 1 to 2550 msec, or wait forever. Valid only if Wait for Command is Yes.
- Communication Component: The communication Component that the bootloader uses. The PSoC Creator bootloader supports many types of communication ports, including a custom option.
- Checksum Type: Type of check bits to use with packets to and from the host: checksum or CRC.

For more information, see Figure 11 on page 14 as well as the Bootloader Component datasheet.

6.3 Communication Component

A PSoC Creator bootloader project must include at least one bootloader-compatible communication Component. Currently, the I^2C Slave, UART, SPI, and USBFS Components are supported for the bootloader as standard .

If you want to use a nonstandard communication channel for bootloading, you can easily create a custom Component. You must write an API for the Component that supports just five functions: Start, Stop, Reset, Read, and Write. For more information on how to create a bootloader custom communication Component, see the PSoC Creator Component Author Guide.

6.4 Recovering from Failures

The PSoC Creator Bootloader Component uses the top (64-, 128-, or 256-byte) row of flash to store data on the application (or both applications for the dual-app option). This data includes checksums and other validity bits for each application. When a bootload starts, these bits are cleared. They are recalculated and updated when the bootload successfully completes.

If power fails or communication is lost during the bootload operation, the checksum of the bootloadable project will be incorrect at the next device reset. The bootloader then waits for another command from the host to start another bootload operation.

6.5 Backward Compatibility

PSoC Creator is designed such that bootloadable projects built with new versions can be linked to and are compatible with bootloaders built with older versions.

6.6 Bootloader Memory Usage

As noted previously, a PSoC Creator bootloader uses memory that could otherwise be used by an application. This may impact application design, or cost, and thus bootloader memory usage is an important specification. The PSoC Creator Bootloader Component memory usage varies significantly, depending on the following:

- Communication Component used
- Bootloader Component configuration options selected (see Figure 11 on page 14)
- Target device – PSoC 3, PSoC 4, and PSoC 5LP have 8051, Cortex-M0, and Cortex-M3 CPUs, respectively
- Compiler and its optimization settings

For details, see the specifications listed in the Bootloader Component datasheet. For a specific bootloader project, after building the project, check the *.map* file generated by the compiler to determine the exact memory usage.

6.7 Flash Protection

All PSoC 3, PSoC 4, and PSoC 5LP devices include a flexible flash-protection system that controls access to the flash memory. This feature is designed to secure the proprietary code, but it can also be used to protect against inadvertent writes to the bootloader portion of the flash memory.

The flash memory is organized in rows of 64 to 256 bytes, depending on the device family. You can assign one of the four protection levels (two levels for PSoC 4) to each row; see Table 1. Flash protection levels can only be changed by performing a complete flash erase. For more information on PSoC flash and security features, see a device data sheet or Technical Reference Manual (TRM).

Table 1. Flash Protection Levels

	PSoC 3 and PSoC 5LP		PSoC 4	
Protection Setting	**Allowed**	**Not Allowed**	**Allowed**	**Not allowed**
Unprotected	External read and write, Internal read and write	–	External read and write, Internal read and write	–
Factory Upgrade	External write, Internal read and write	External read	NA	NA
Field Upgrade	Internal read and write	External read and write	NA	NA
Full Protection	Internal read	External read and write, Internal write	Internal read	External write, Internal write (see Note below)

Note: To protect the PSoC 4 device from external read operations, you must change the device protection settings to "Protected" in PSoC Creator *.cydwr* system settings and use the PSoC Programmer software to program the device. You must also enable "Chip Lock" from **Options** > **Programmer Options** before programming the device for these settings to take effect.

To protect the bootloader portion of flash, set the corresponding rows to "full protection". PSoC Creator lets you easily select the protection setting for each row. For more information, see the PSoC Creator help or one of the advanced bootloader application notes listed in Related Application Notes.

6.8 Customization

A bootloader is a PSoC Creator project and, similar to any other PSoC Creator project, enables PSoC to be configured for any application. This, in turn, makes it easy to customize a bootloader, especially for high-reliability applications:

- Other tasks during bootloading: Components can be added to the bootloader project schematic; in many cases, these Components can perform complex tasks without the use of the CPU.

  If you do need to use the CPU to perform another task while bootloading, the easiest way to do so is to structure the task as a state machine, embedded in a periodic interrupt handler. This way, the bootloader and the secondary task can operate as independent processes.

- Preserve pin states: Pin Components can be placed on the schematic and their states set for both device reset and bootloader startup. For more information on controlling pin states, see AN61290, PSoC 3 and PSoC 5LP Hardware Design Considerations, or AN88619, PSoC 4 Hardware Design Considerations. See also Appendix A.

7 Add a Bootloader to Your PSoC Creator Project

Now that we have seen how bootloaders are implemented in PSoC Creator, let us look at some practical steps for doing so. For more details, see one of the advanced bootloader application notes listed in Related Application Notes.

7.1 Building a Bootloader

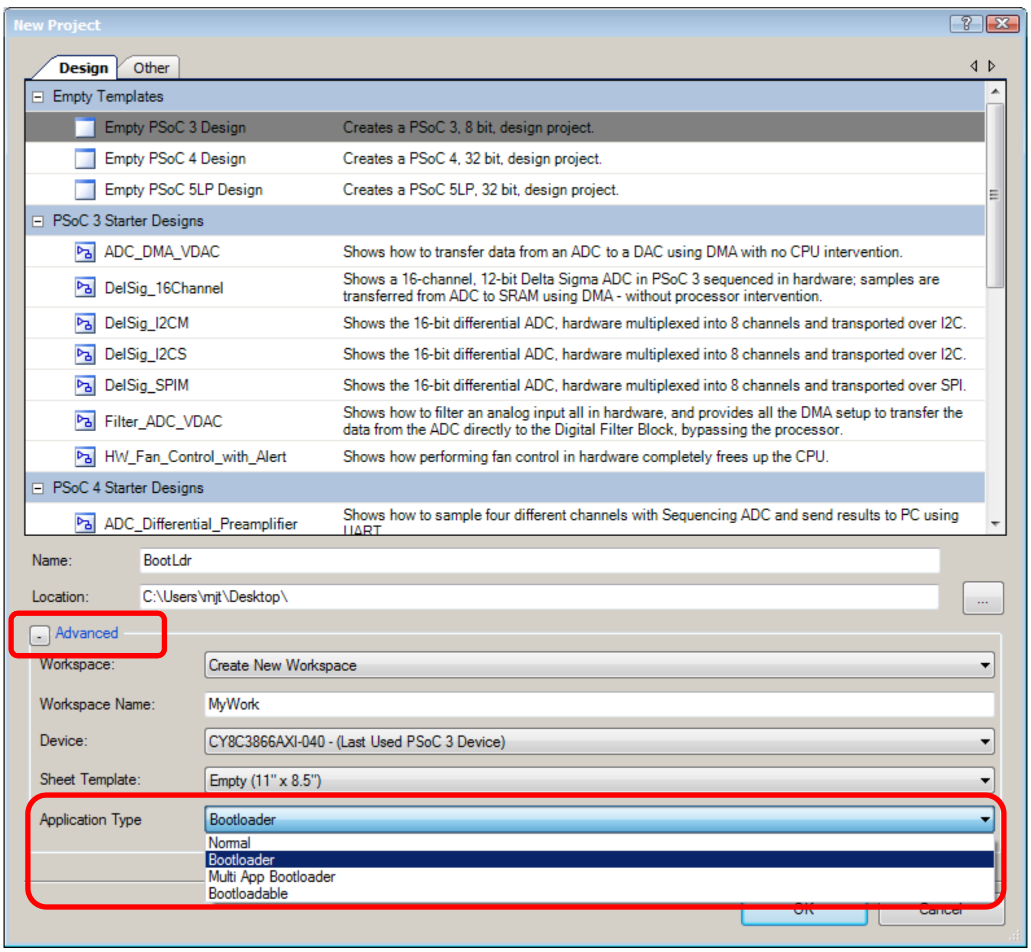
With other MCUs, you usually add a bootloader to an application. However, with PSoC Creator, the best practice is to design in the opposite direction – first, create a bootloader project and then create one or more bootloadable projects.

To create a bootloader project, simply create a project of type **Bootloader** or **Multi-App Bootloader**, as Figure 8 shows. Note that in the example, the project name "BootLdr" is different from the workspace name "MyWork" – a PSoC Creator workspace can contain multiple projects of different types.

Note: Beginning with **PSoC Creator 3.2**, the "Application Type" option is removed from the New Project dialog box. PSoC Creator automatically recognizes the application type from the TopDesign schematic. This update does not affect existing bootloader projects.

Note: Beginning with PSoC Creator 3.3, the New Project dialog box has changed, and you are not required to select the Application Type option.

Figure 8. Creating a Bootloader Project in PSoC Creator 3.1 or earlier



After a project is created, drag onto the project schematic a Bootloader Component and the communication Component to be used for bootloading. As Figure 9 shows, the Bootloader and Bootloadable Components are available under the System tab in the Component Catalog window. Figure 10 shows a bootloader project schematic with a Bootloader Component and an I²C communication Component for bootloading.

Figure 9. Component Catalog Window

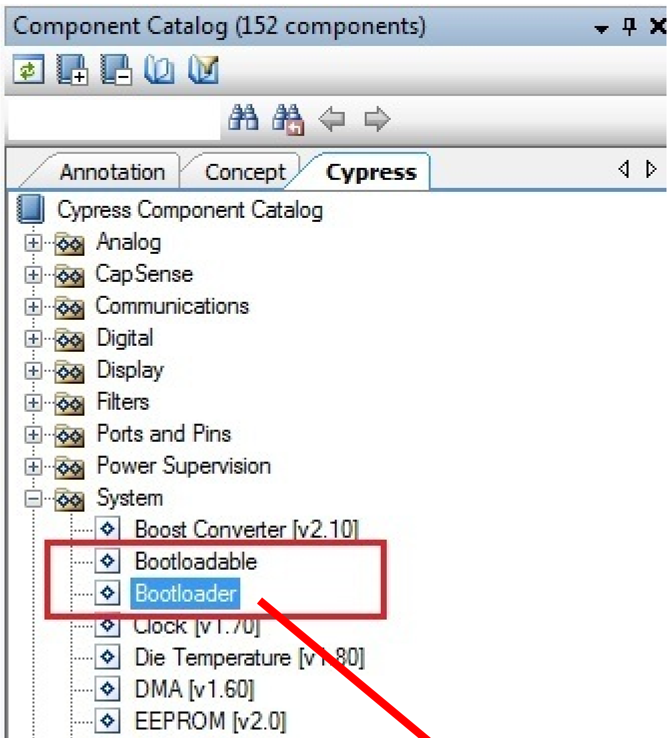
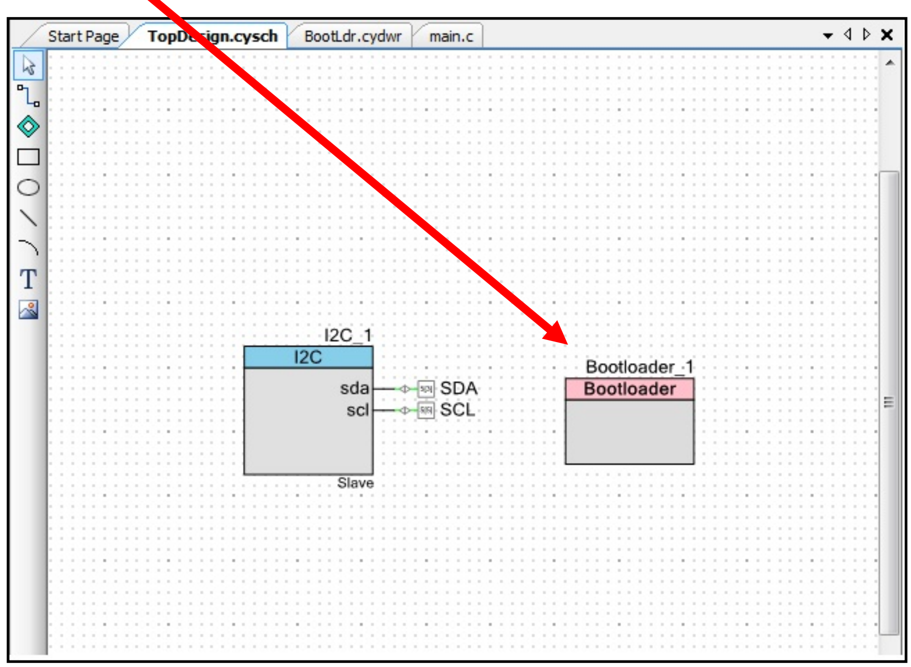
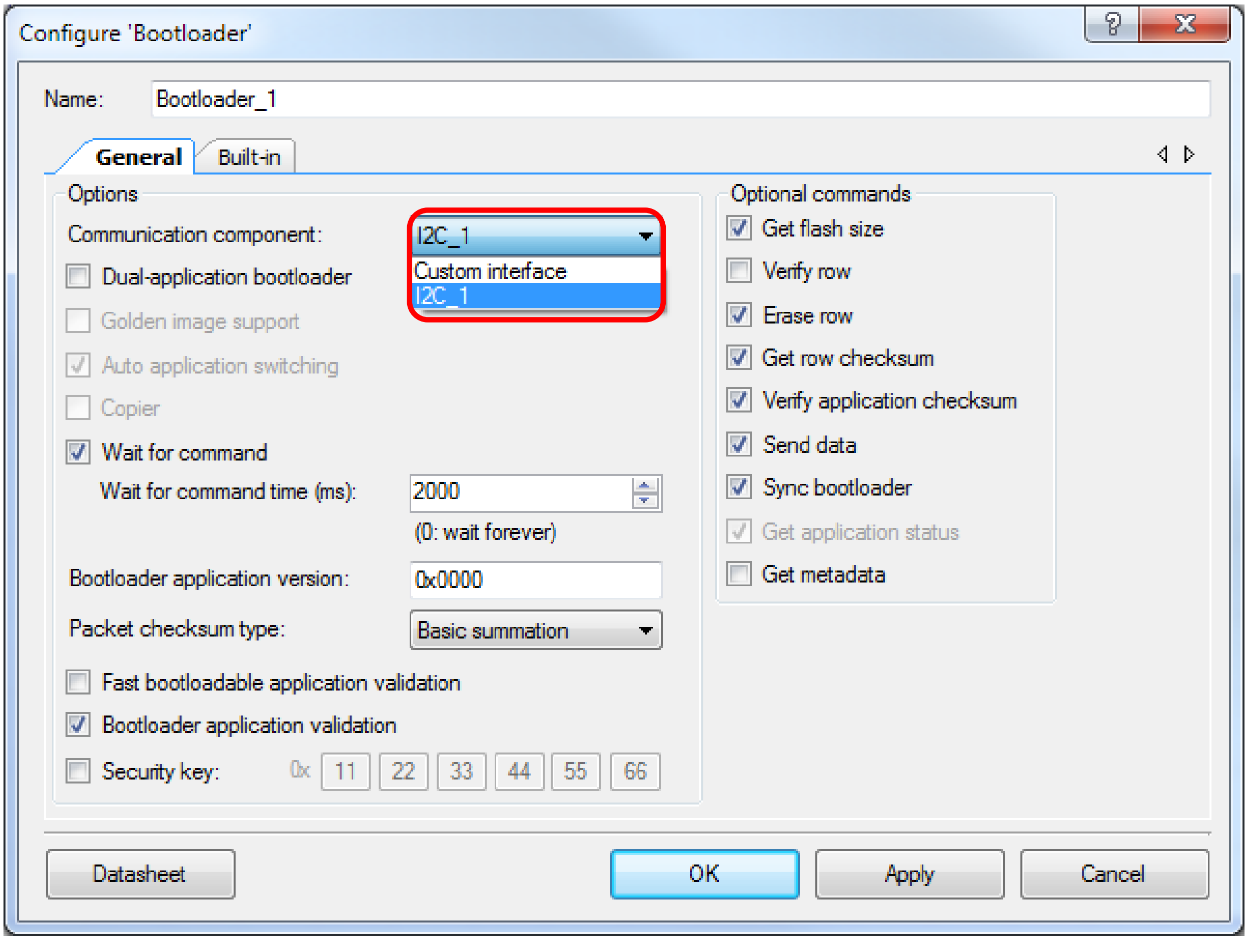


Figure 10. Bootloader Project Schematic



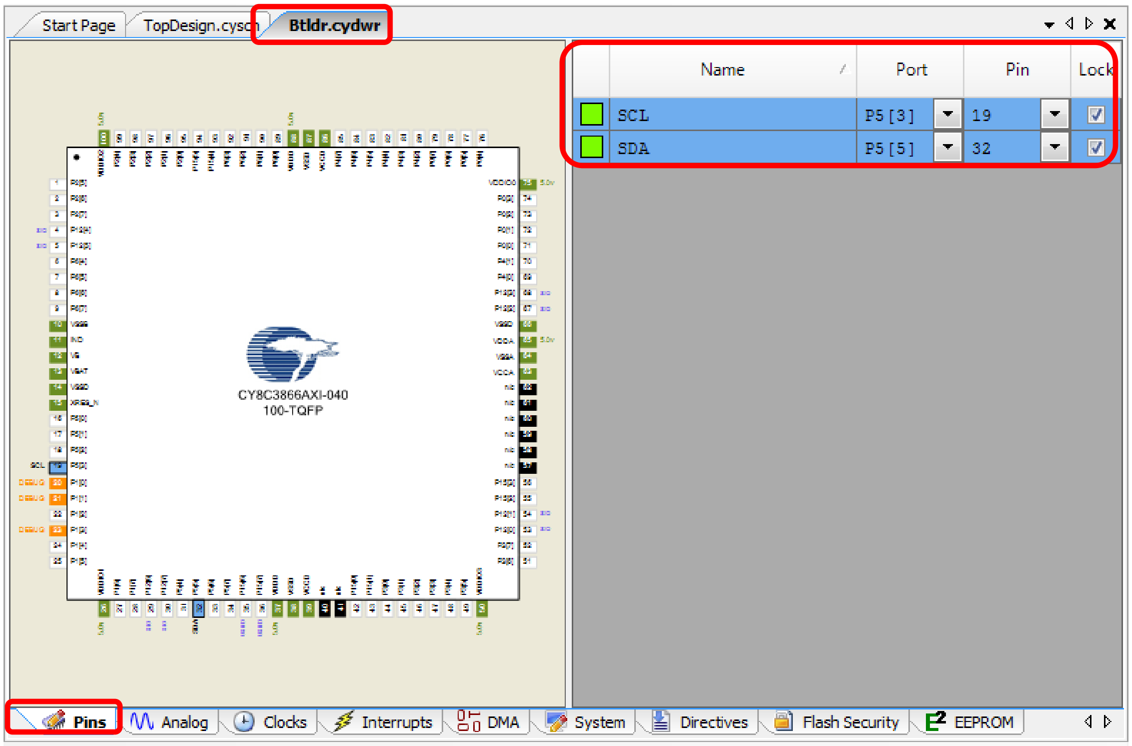
Then, configure the Bootloader Component, as Figure 11 shows. Note the menu to select the communication Component in Figure 11 – this is the Component that is used to communicate with the host. A bootloader project must have this Component defined and selected. You can select from all of the bootloader-compatible communication Components that you have on your schematic or you can select **Custom_Interface** and define your own.

Figure 11. Bootloader Component Configuration



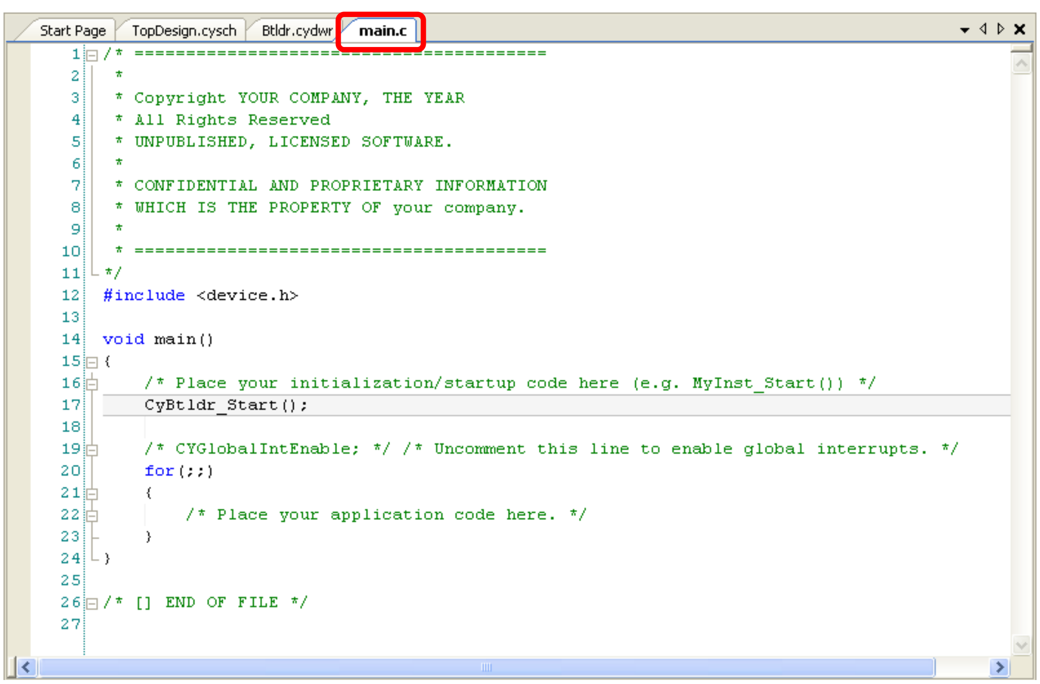
Finally, in the design-wide resources (DWR) window, finish the project by connecting schematic pins to physical pins, as Figure 12 shows.

Figure 12. Bootloader Project Pin Assignments



Build the project. Everything else is done for you, and the result is a basic bootloader project. The auto-generated *main.c* file has just one line of code, to call the bootloader start function, as Figure 13 shows.

Figure 13. Bootloader Default *main.c*

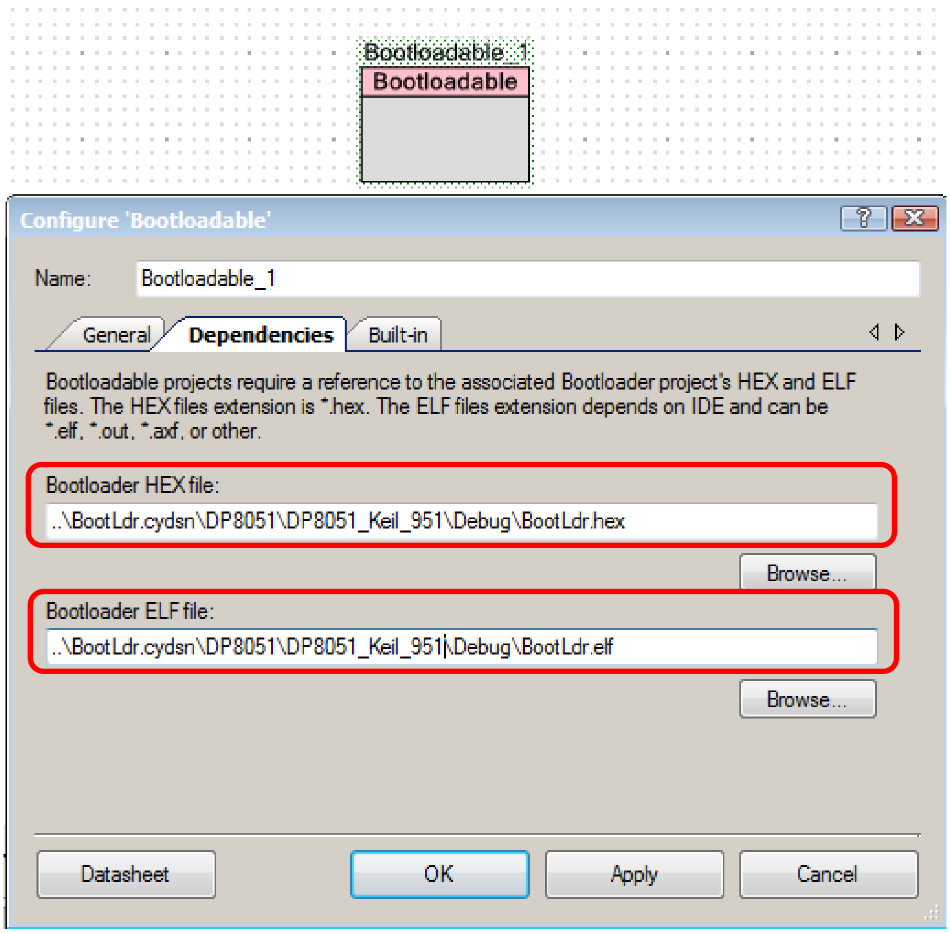


7.2 Adding Bootloadable Applications

After the bootloader is created, you can define as many bootloadable applications, that is, projects, as you want, using the **Bootloadable** option shown in Figure 8 on page 12. You can also change an existing **Normal** project to type **Bootloadable**; see page 17 for details.

A bootloadable project must have on its schematic a Bootloadable Component (see Figure 9 on page 13). The project must also be associated with a bootloader project, as Figure 14 shows. To do so, select the location of the bootloader's *.hex* and *.elf* file in the Bootloadable Component configuration dialog; see Project Files on page 19 for details.

Figure 14. Bootloadable / Bootloader Projects Link



A PSoC Creator workspace can have multiple projects. In many cases, a bootloader project exists in the same workspace as its associated bootloadables. However, bootloaders and bootloadables can exist in separate workspaces and separate locations on your PC. Before getting started with PSoC, it is a good idea to work out a workspaces / projects plan for your overall system development needs.

Note: Flash protection settings for a bootloadable project are ignored; the associated bootloader project's flash protection settings take precedence.

Note: For PSoC Creator versions before 3.0, if the bootloader is updated, you must also rebuild all bootloadable projects that depend on that bootloader project. Use the "Clean and Build" option.

7.3 Debugging Bootloadable Projects

In the PSoC Creator bootloader system, the bootloader project executes first and then the bootloadable project. The jump from the bootloader to the bootloadable project is done through a software-controlled device reset; see Appendix A for details. This resets the debugger interface, which means that the bootloadable project cannot be run in debugger mode.

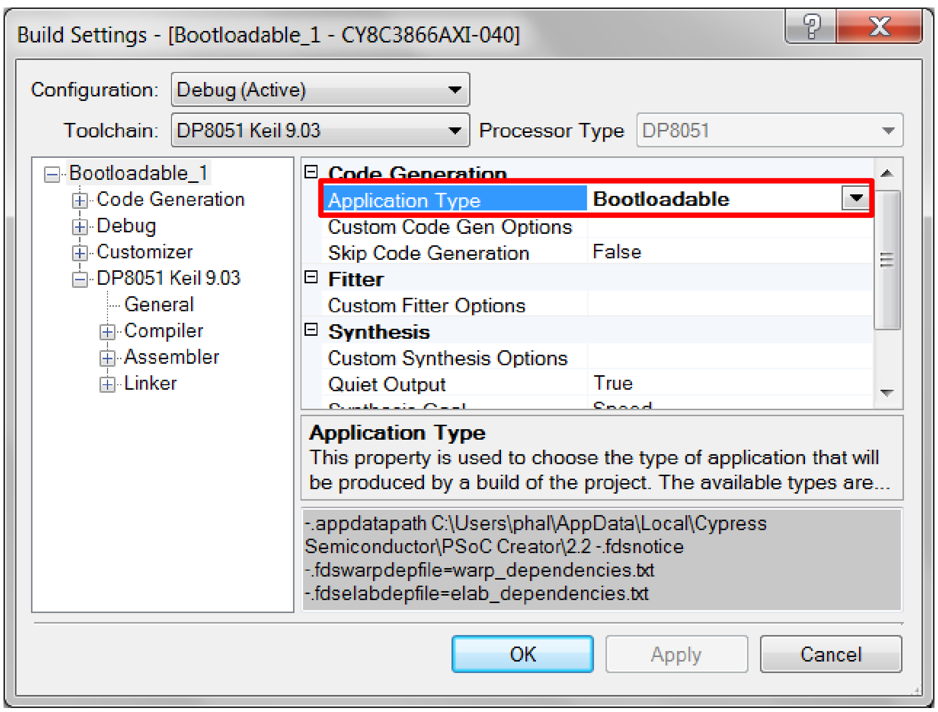
To debug a bootloadable project, convert it to Application Type Normal (Figure 15), debug it, and then convert it back to Bootloadable after debugging is done.

Another option is to program the Bootloadable project *.hex* file onto the device and then use the **Attach to running target** option for debugging while the bootloadable project is running. In this case, you can debug the bootloadable project only from the point where debugger is attached to the device.

7.3.1 Converting a Normal Application Project to a Bootloadable Project

If you have already created a standard (Normal) project and want to convert it to a bootloadable project, you can change the **Application Type** of the project to Bootloadable, as Figure 15 shows.

Figure 15. Changing Application Type to Bootloadable



After changing the application type, you must add a Bootloadable Component to the project schematic, and add a bootloader project's *.hex* file as a dependency, as Figure 14 shows.

Note: Beginning with PSoC Creator 3.2, the "Application Type" option is removed from the Build Settings dialog box. PSoC Creator automatically recognizes the application type from the components in the TopDesign schematic. If a bootloadable Component is present in the TopDesign schematic, the PSoC Creator considers the project as a bootloadable project.

7.4 Customizing Your Bootloader

As mentioned in Customization on page 8, you can customize your bootloader by dragging additional Components onto your schematic and adding code to *main.c*. As a simple example, you can add a PWM, a Clock, and a Pin Component to blink an LED as a "bootloading" indicator, as Figure 16 and Figure 17 show. You can easily configure the Components to make the LED blink at any desired frequency and duty cycle.

Note that the PSoC configuration for the bootloader project exists only until the bootloader transfers control to the bootloadable. The PSoC device is then reconfigured for the bootloadable project. If you want the same functionality in both projects, you can place the same Components and code in both projects.

Figure 16. Bootloader Project Customization

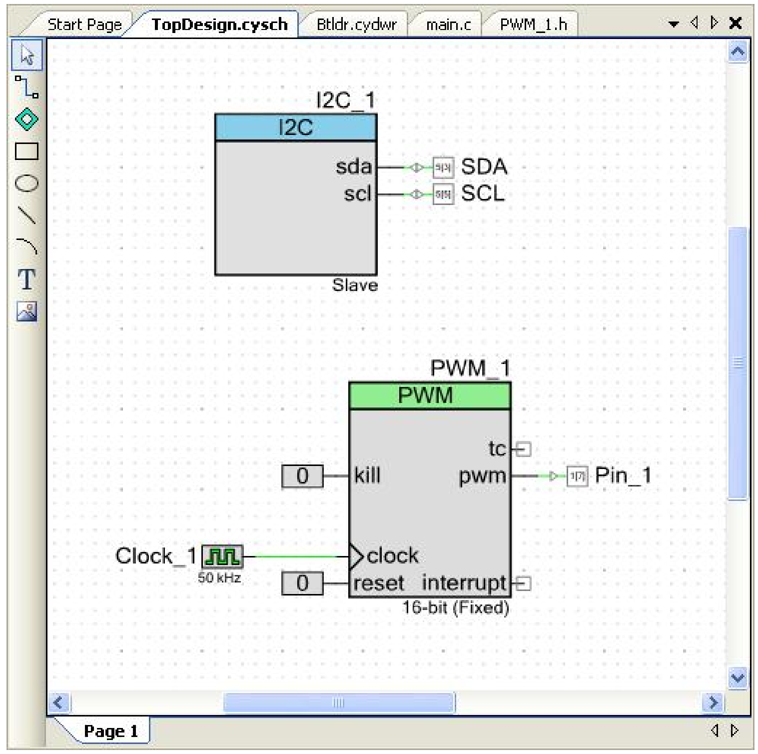
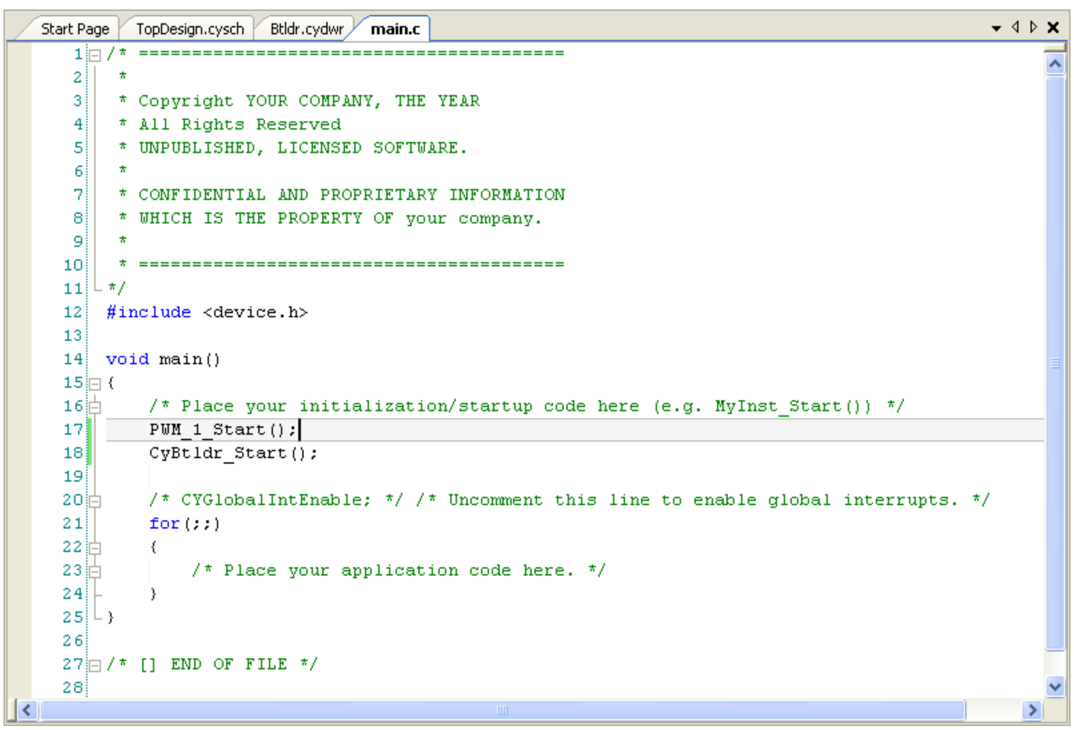


Figure 17. Bootloader Customization in *main.c*




```c
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include <device.h>

void main()
{
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    PWM_1_Start();
    CyBtldr_Start();

    /* CYGlobalIntEnable; */ /* Uncomment this line to enable global interrupts. */
    for(;;)
    {
        /* Place your application code here. */
    }
}

/* [] END OF FILE */
```

7.5 Calling the Bootloader

As mentioned in Bootloader - Host Timing, you can avoid bootloader initial timing issues by having the application call the bootloader. Then, the application can respond to some external event, such as a button press or a message from the host, and start a bootload operation.

The Bootloader Component has an application program interface (API) with a public function Bootloader_Start(). Call this function to start a bootload operation from bootloadable project code.

Bootloader_Start() does a software reset of the device, and the bootloader takes over the CPU. Resources and peripherals are reconfigured for the bootloader; the bootloadable configuration is disabled. Bootloadable project code, including interrupt handlers, is no longer executed. When the bootload operation is complete, the CPU is reset again. See Appendix A for details.

8 Loading Your Projects into PSoC

Similar to standard projects, a bootloader project can be installed in a target PSoC device only through JTAG or SWD, using PSoC Creator or PSoC Programmer. After a bootloader is installed and active, bootloadable projects can be installed by a bootload operation instead of through JTAG or SWD.

PSoC Creator includes a PC program called Bootloader Host, which does the PC side of a bootload operation. It communicates with the bootloader in a target PSoC, either directly through a USB port or through an I^2C port, by use of a programmer such as the CY8CKIT-002 MiniProg3 kit. To use Bootloader Host, you need to know about the output files generated for bootloader and bootloadable projects.

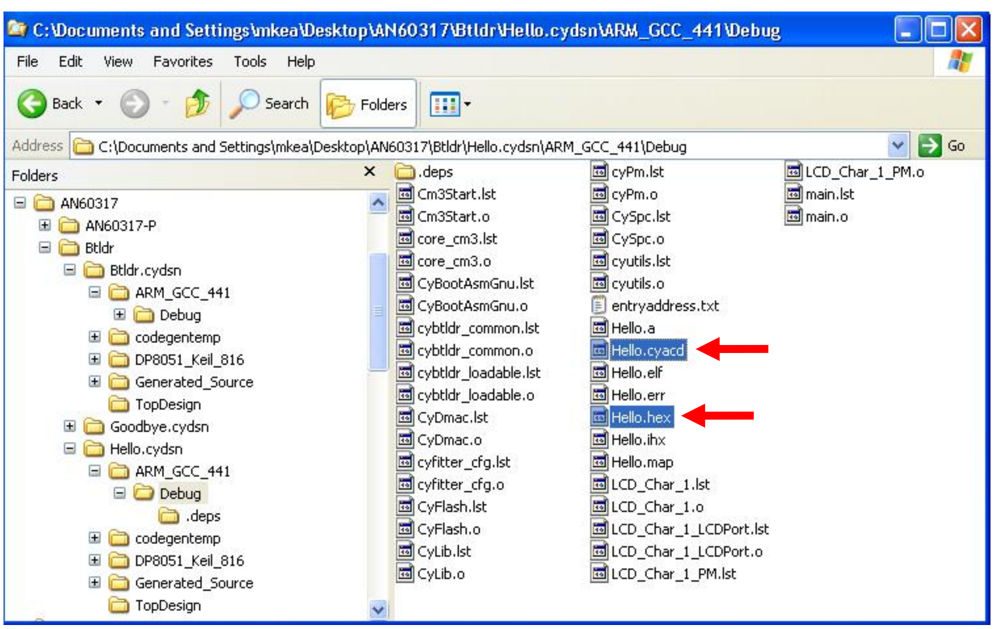
8.1 Project Files

Once built, all PSoC Creator projects produce *.hex* files as outputs. These files can be used with any HSSP programmer device, including MiniProg3. Hex files contain bytes for both CPU code and project configuration.

For normal and bootloader projects, the *.hex* file contains code and data bytes for just that project. Bootloadable *.hex* files are different in that they contain code and data bytes for the bootloadable *and* the associated bootloader project – both projects are programmed in at the same time.

Bootloadable projects are also different from normal projects in that they produce a second file, of type *.cyacd*, as an output, as Figure 18 shows. The *.cyacd* file contains code and data for just the bootloadable project, without the associated bootloader. It is intended to be used by a bootloader host program, and downloaded to a target PSoC that has the associated bootloader project already installed.

Figure 18. Bootloadable Project Files



8.2 Use Cases

After the projects are built and the output files created, you typically use one of the following scenarios (see also Figure 4 on page 5):

1. Create and build a bootloader project. Program its *.hex* file into the target PSoC using an HSSP programmer such as MiniProg3.
2. Reset the target PSoC to start the bootloader. Because the bootloader is the only project in flash, it waits forever for bootload commands from the host.
3. Create a bootloadable project, associate it with the bootloader project, and build it. Download its *.cyacd* file to the target using a host program and the previously installed bootloader.

For subsequent bootload operations, note that because a valid bootloadable exists in flash, the bootloader waits for the host for only a short period of time before passing control to the bootloader.

In a factory production scenario, you can do the following instead:

1. Create and build a bootloader project.
2. Create a bootloadable project, associate it with the bootloader project, and build it. Program its *.hex* file (which contains both bootloader and bootloadable) into the target PSoC device using an HSSP programmer such as MiniProg3.
3. Reset the target PSoC device to start the bootloader. The bootloader sees a valid bootloadable in flash and, after a possible timeout wait for bootload commands from the host, passes control to the bootloadable.

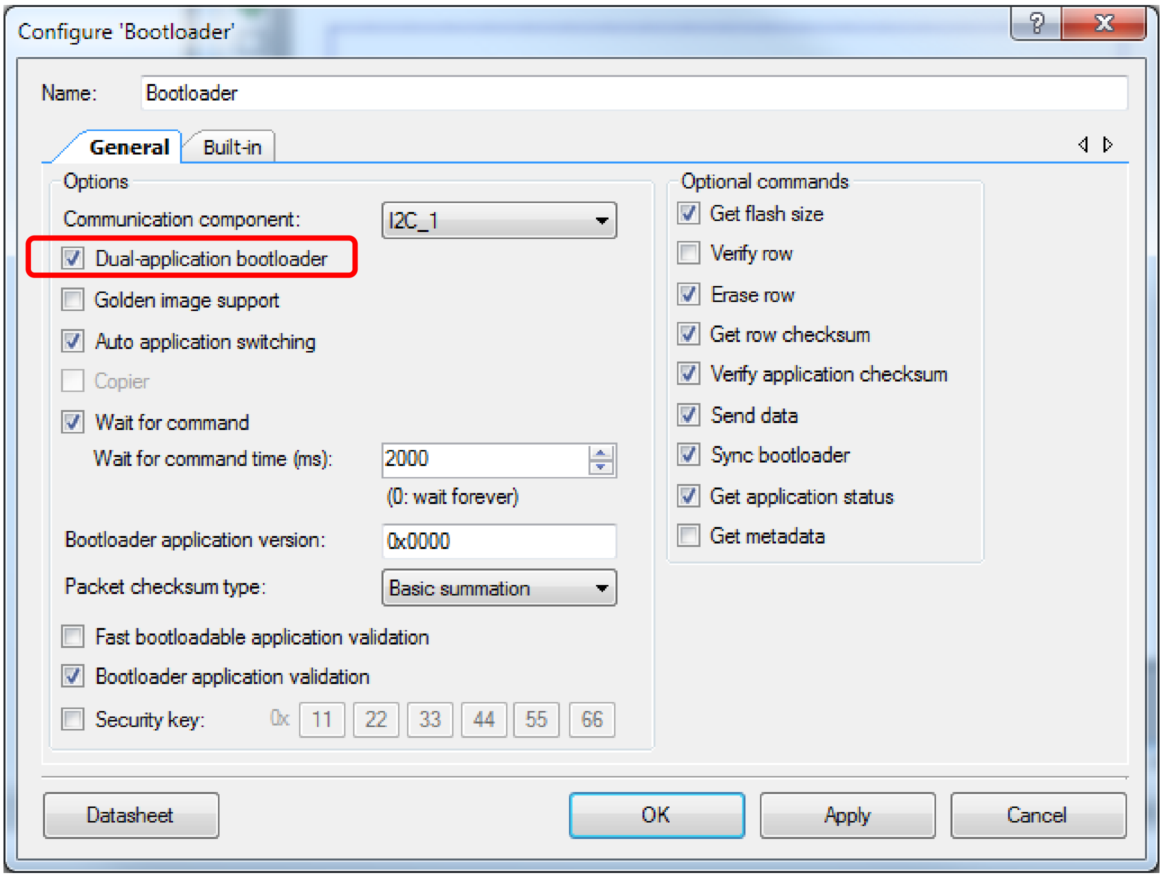
If in future you update the bootloadable, you can download its *.cyacd* file to the target using a host program. This overwrites the previous version of the bootloadable.

9 Dual-Application Bootloader Considerations

The PSoC Creator Bootloader and Bootloadable Components support dual application images for high-reliability applications, as described in Customization. The PSoC Creator build process for a dual-application bootloader is similar to that for single applications, but there are a few differences:

1. Select **Multiple-application bootloader** (see Figure 8) to create a dual-application bootloader project. This step is not required for PSoC Creator 3.2 and later.
2. In the Bootloader Component configuration dialog, check the box **Dual-application bootloader**, as Figure 19 shows. In PSoC Creator 3.2 and earlier, this option is named **Multi-Application bootloader**.

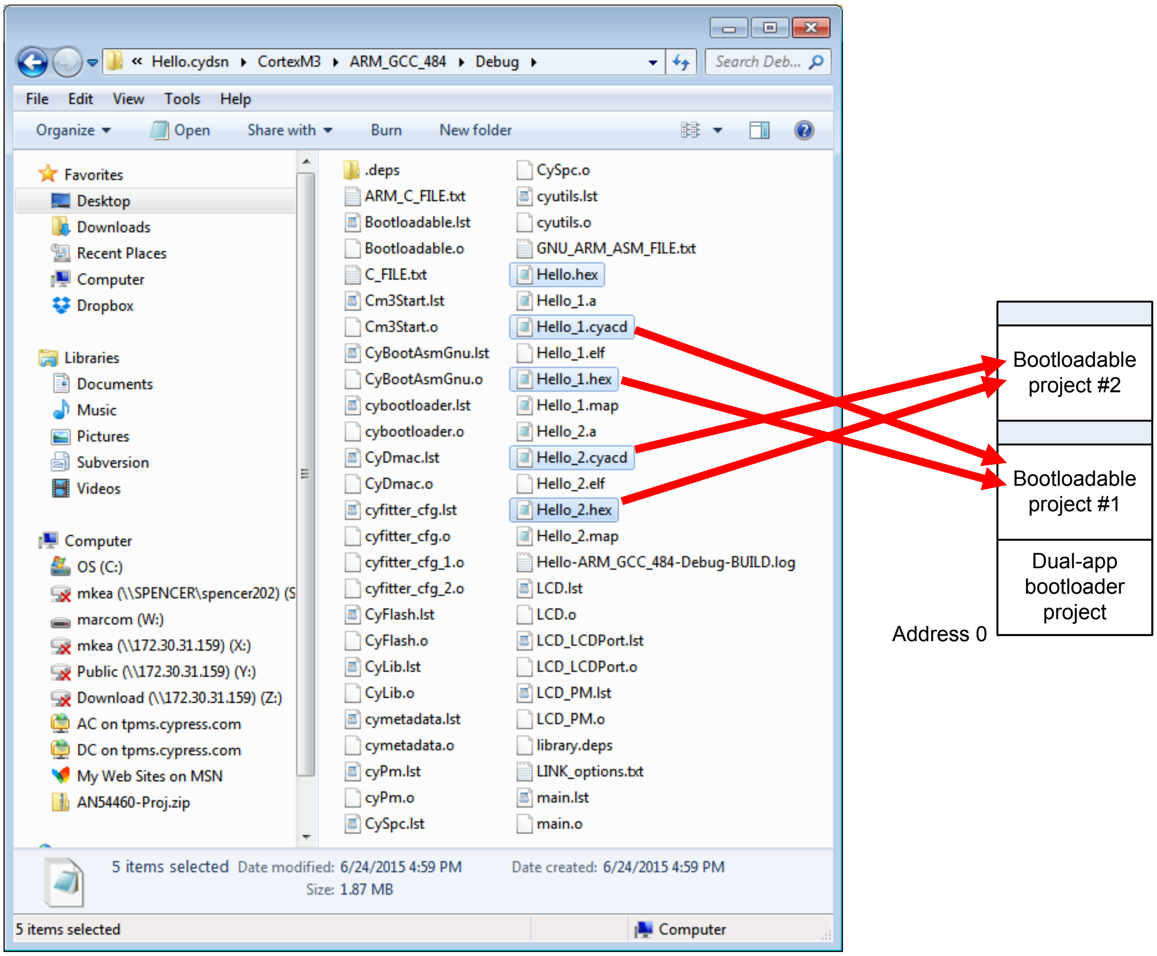
Figure 19. Bootloader Component Configuration



3. Project Files: A dual-application bootloadable project has five output files, instead of the two files shown in Figure 18. These files allow placement of the bootloadable project as either application 1 or application 2, as Figure 20 shows. For a high-reliability application, you can place two copies of the same bootloadable project into flash.

   You can also create two different bootloadable projects. You can then install one of them as the first application and the other as the second application.

Figure 20. Dual-Application Bootloadable Project Files



As noted previously, *.hex* files are installed through the JTAG/SWD port, typically when a product is manufactured. The *.hex* files contain the bootloader project as well as one or both of the bootloadable projects.

The *.cyacd* files are installed in the field, for example with a bootloader host program.

9.1 Application Launch Process

One of the decisions that a dual-application bootloader must make is which (if any) application to “launch”, or transfer control to. Each application has two characteristics that drive this decision:

- Active: As noted previously, the PSoC Creator Bootloader Component uses the top rows of flash to store data on the applications (also known as “metadata”). This data includes an “active” bit. Only one of the applications has its active bit set, and that is the application that is preferred for launching.
- Valid: Before launching, the bootloader tests each application against the check bytes in the metadata to determine which, if any, of the applications are valid. The test may fail, for example, due to corrupted flash memory or having no application installed in that flash. In this case, the application is “not valid”.

Table 2 shows the decision matrix that the bootloader uses to decide which application to launch. Note that some of the cases, such as both applications being active, are illegal and should not happen under normal conditions.

Table 2. Application Launch Decision Matrix

	Application #1		Application #2		
Case	**Active**	**Valid**	**Active**	**Valid**	**Bootloader Action**
0	0	0	0	0	Stay in bootloader, wait forever for host
1	0	0	0	1	same as case #0
2	0	0	1	0	same as case #0
3	0	0	1	1	Go to Application #2
4	0	1	0	0	same as case #0
5	0	1	0	1	same as case #0
6	0	1	1	0	same as case #0
7	0	1	1	1	Go to Application #2
8	1	0	0	0	same as case #0
9	1	0	0	1	same as case #0
10	1	0	1	0	same as case #0
11	1	0	1	1	Go to Application #2
12	1	1	0	0	Go to Application #1
13	1	1	0	1	Go to Application #1
14	1	1	1	0	Go to Application #1
15	1	1	1	1	Go to Application #1

10 Summary

This application note has provided a basic overview of bootloaders – how they are used and important design considerations. It has also shown how the PSoC Creator design environment addresses these considerations for PSoC 3, PSoC 4, and PSoC 5LP devices.

You have also seen a basic overview of how to use PSoC Creator to quickly and easily add a bootloader to your design. For application notes that cover these topics in more detail, see Related Application Notes.

11 Related Application Notes

Bootloader Application Notes

All the bootloader application notes listed below have associated code examples on the Cypress webpage.

- AN60317, PSoC 3 and PSoC 5LP I^2C Bootloader
- AN86526, PSoC 4 I2C 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

Other Related Application Notes

- AN73054, PSoC 3 and PSoC 5LP Programming Using an External Microcontroller (HSSP)
- AN84858, PSoC 4 Programming Using an External Microcontroller (HSSP)
- AN61290, PSoC 3 and PSoC 5LP Hardware Design Considerations
- AN54181, Getting Started with PSoC 3
- AN79953, Getting Started with PSoC 4
- AN77759, Getting Started with PSoC 5LP
- AN2100, Bootloader: PSoC 1

About the Author

Name: Mark Ainsworth

Title: Applications Engineer Principal

Background: Mark has a BS in Computer Engineering from Syracuse University and an MSEE from University of Washington, as well as many years experience designing and building embedded systems.

A Appendix A – Bootloader and Device Reset

As noted elsewhere in this application note, transferring control from the bootloader to the bootloadable, or vice versa, is always done through a device reset. This may be a consideration if your system must continue to perform mission-critical functions while changing from one program to the other. This section details why reset must be used, as well as its implications for device performance in your application.

A.1 Why is Device Reset Needed?

To understand why device reset is needed, it is important to note that the bootloader and bootloadable projects in your system are each completely self-contained PSoC Creator projects. Each project has its own device configuration settings. Thus, when you change from one project to the other, you can completely redefine the hardware functions of the PSoC device.

To implement complex custom functions, device configuration can involve the setting of thousands of PSoC registers. This is especially true for PSoC's digital and analog routing features. When you configure the registers and routing, you must make sure that, in addition to setting the bits for the new configuration, you reset the bits for the old configuration. Otherwise, the new configuration may not work, and may even damage the device.

So when changing between bootloader and bootloadable projects, a device software reset (SRES) is done. This causes all PSoC registers to be reset to their default states. Configuration for the new project can then begin. Note that by assuming that all PSoC registers are initialized to their device reset default states, configuration time and flash memory usage are both reduced.

A.2 Effect on Device I/O Pins

As described in application notes AN61290, PSoC 3 and PSoC 5LP Hardware Design Considerations, and AN60616, PSoC 3 and PSoC 5LP Startup Procedure, during the reset and startup process the PSoC I/O pins are in three distinct drive modes, as Table 3 shows.

Table 3. PSoC I/O Pin Drive Modes During Device Reset

Startup Event	I/O Pin Drive Mode	Duration (Typical)		Comment
		Slow IMO (12 MHz)	Fast IMO (48 MHz)	
Device reset (SRES) active Device reset removed	HI-Z Analog	40 µs		While reset is active, the I/Os are held in the HI-Z Analog mode.
Nonvolatile latches (NVLs) copied to I/O ports Code starts executing	NVL setting: HI-Z Analog, Pull-up, or Pull-down	~12 ms	~4 ms	Duration depends on code execution speed and configuration complexity.
I/O ports and pins are configured	PSoC Creator project configuration	N/A		8 possible drive modes. See the device datasheet for details.
Code reaches main()	Code may change I/O pin function	N/A		

For details on NVL usage in PSoC, see a device datasheet. In your PSoC Creator project, the NVL settings are established in two places:

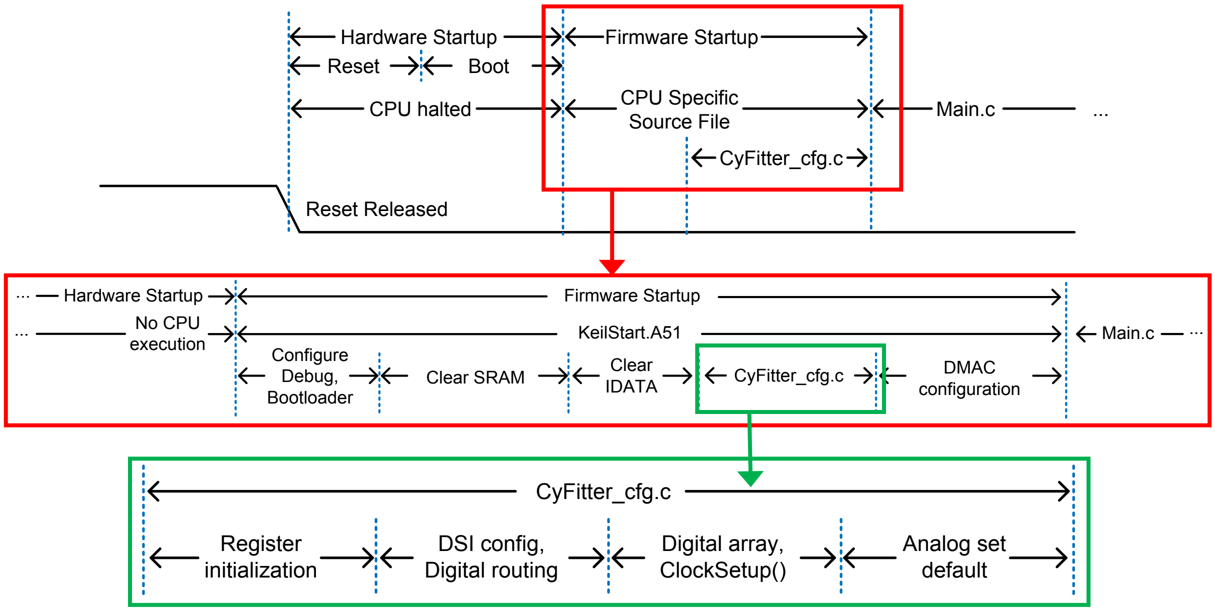
- I/O ports: the **Reset** tab in the individual Pin Component configurations
- All other NVLs: The **System** tab in the design-wide resources (DWR) window

The NVLs are updated when the device is programmed with your project. Note that a bootloadable project cannot set NVLs; its DWR settings must match those in the associated bootloader project.

Final I/O drive modes are set by individual Pin Component configurations.

Figure 21 shows the timing diagrams for device startup and configuration. The example in the middle diagram is for PSoC 3; similar processes exist for PSoC 4 and PSoC 5LP. For more information, see AN60616, PSoC 3 and PSoC 5LP Startup Procedure.

Figure 21. Device Startup Process Diagrams



A.3 Effect on Other Functions

At device reset, UDB registers are reset, so all UDB-based Components cease to exist and their functions are stopped. The same is also true for analog Components based on the configurable SC/CT blocks in PSoC 3 and PSoC 5LP.

All fixed peripherals – digital and analog – are reset to their idle states. This includes the DMA, DFB, timers (TCPWM), I^2C, USB, CAN, ADCs, DACs, comparators, and opamps. All clocks are stopped except the IMO.

All digital and analog routing control registers are reset. This causes all digital and analog switches to be opened, breaking all connections within the device. This includes all connections to the I/Os except the NVLs.

All hardware-based functions are restored after configuration (see Figure 21). All firmware functions are restored when the project's main() function starts executing.

A.4 Example: Fan Control

Let us examine how a bootloader and its associated device reset can be integrated into a typical application such as fan control. PSoC Creator provides a Fan Controller Component, which encapsulates all necessary hardware blocks including PWMs, tachometer input capture timer, control registers, status registers, and a DMA channel or interrupt. For more information, see the Fan Controller Application page.

The fan control application is in a bootloadable project. Optionally, the bootloader may be customized to keep the fan running while bootloading.

The fan can also be kept running while the device is reset, during the transfer between the bootloader to the bootloadable, as Table 4 shows.

Table 4. PSoC I/O Pin Drive Modes During Device Reset for Fan Controller

I/O Pin Drive Mode	Comment
HI-Z Analog	Optionally, add external pull-up or pull-down resistor to the PWM pin for 100% duty cycle. This may not be needed because the fan may keep spinning due to inertia.
NVL setting: HI-Z Analog, Pull-up, or Pull-down	Optionally, set the PWM Pin Component reset value to Pull-up or Pull-down for 100% duty cycle. This may not be needed because the fan may keep spinning due to inertia.
PSoC Creator project configuration	Set the PWM Pin Component drive mode and initial state for 100% duty cycle. The PWM Component becomes active but does not run.
Main() starts executing	When PWM_Start() is called, the PWM starts driving the PWM pin at the Component's default duty cycle. Firmware can read the tachometer data and start actively controlling the duty cycle.

Document History

Document Title: AN73854 - PSoC® 3, PSoC 4, and PSoC 5LP Introduction to Bootloaders

Document Number: 001-73854

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3434408	MKEA	11/09/2011	New application note.
*A	3672780	MKEA	07/11/2012	Updated for PSoC Creator 2.1
*B	3720294	MKEA	08/22/2012	Updated Figure 13.
*C	3817214	MKEA	11/20/2012	Updated for PSoC 5LP and PSoC Creator 2.1 SP1.
*D	4192464	MKEA	11/14/2013	Updated for PSoC 4. Changed System Reference Guide reference to Component Author Guide. Added a note to clean and build bootloadable projects when a bootloader project is changed. Updated to latest application note template spec.
*E	4435010	MKEA	07/17/2014	Added Appendix A – Bootloader and Device Reset
*F	4507132	MKEA	09/18/2014	Expanded and clarified Table 1 on flash protection. Added a note that bootloader flash protection settings take precedence and bootloadable settings are ignored. Added sections on bootloader memory usage and debugging bootloadable application projects. Other minor edits and formatting changes.
*G	4675937	RNJT	03/18/2015	Updates for PSoC 4200M: Updated Table 1, Figure 7, and Figure 13. Updated the Related Application Notes section. Added notes to indicate changes in PSoC Creator 3.2 for selecting the application type. Added a note to explain the method to restrict external reads for a PSoC 4 device. Updated the Introduction section.
*H	4827135	MKEA	07/08/2015	Added sections PSoC Resources, PSoC Creator, and Dual-Application Bootloader Considerations. Updated flash row size statements in various sections. Updated format to latest template. Miscellaneous minor edits.
*I	4883371	RNJT	09/22/2015	Updated the following for PSoC 4200L: Updated the PSoC Resources section to include CY8CKIT-046. Updated the Building a Bootloader section. Updated the Converting a Normal Application Project to a Bootloadable Project sub section. Updated Steps 1 and 2 in the Dual-Application Bootloader Considerations section. Updated Figure 11, Figure 12 and Figure 19.

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

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

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

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com