

MSPBoot – Main Memory Bootloader for MSP430™ Flash Microcontrollers



Luis Reynoso, Caleb Overbay

MSP430 Apps

ABSTRACT

This application note describes the implementation of a bootloader that resides in the main memory of MSP430™ flash-based microcontrollers (MCUs) using either Inter-Integrated Circuit (I²C), universal asynchronous receiver/transmitter (UART), or a serial peripheral interface (SPI) bus and CC110x RF transceivers to accomplish over-the-air download (OAD). While highly flexible and modular, this bootloader has a small footprint, which makes it a very cost-effective solution, and supports the large memory model (devices with a memory footprint greater than 64KB).

A software package that includes examples and source code for both host and target devices can be downloaded from the following URL: <https://www.ti.com/tool/download/MSPBOOT>

For a step-by-step procedure that explains how to run the examples, see [Section 4.2.5](#).

Do not confuse this bootloader with the MSP430 Bootloader (BSL), which resides in protected memory (ROM or flash) in some MSP430 MCUs. For more information on the MSP430 BSL, see the [MSP430™ Flash Device Bootloader \(BSL\) User's Guide](#) and [Creating a Custom Flash-Based Bootloader \(BSL\)](#).

Note

[MSP430FRBoot](#) is an extension to MSPBoot and implements a main memory resident bootloader for MSP430 FRAM MCUs using various communication interfaces and over-the-air download (OAD) capabilities. MSP430FRBoot supports the large memory model (devices with a memory footprint greater than 64KB) as well as dual image and interrupt redirection options, making it a good customizable alternative to the built-in BSL on MSP430 FRAM MCUs.

Table of Contents

1 Introduction	3
1.1 Glossary.....	3
1.2 Conventions.....	4
2 Implementation	4
2.1 Main.....	4
2.2 Application Manager.....	5
2.3 Memory Interface (MI).....	10
2.4 Communication Interface (CI).....	11
3 Customization of MSPBoot	17
3.1 Predefined Customizations.....	19
4 Building MSPBoot	19
4.1 Starting a New Project.....	19
4.2 Examples.....	26
5 References	32
6 Revision History	32

Trademarks

MSP430™, Code Composer Studio™, LaunchPad™, and BoosterPack™ are trademarks of Texas Instruments. All trademarks are the property of their respective owners.

1 Introduction

MSP430 MCUs are equipped with the useful Bootloader (BSL) that allows for a very simple way to do field upgrades. For more information about the MSP430 BSL, see the [MSP430™ Flash Device Bootloader \(BSL\) User's Guide](#) and [Creating a Custom Flash-Based Bootloader \(BSL\)](#). The BSL is customizable in MSP430F5xx and MSP430F6xx devices, because it resides in flash.

Other families (for example, MSP430G2xx) have a ROM-resident BSL that supports only UART and cannot be modified to support I²C or other interfaces. Given these limitations, it becomes necessary to create a bootloader that resides in main memory and allows for an easy implementation of the application.

This application report describes the implementation of the bootloader named MSPBoot with the following characteristics:

- Small footprint (less than 4KB in size required)
- 20-bit and 16-bit incorporation for large memory model and small memory model devices, respectively
- Supports USI, USCI, and eUSCI peripherals
- UART communication offers the most simple wired interface using a small memory space
- SPI bus offers over-the-air downloads (using the CC110x) at a slightly larger footprint
- Different options allow for customizable levels of robustness
- Optional dual-image support in case of communication interruption
- Allows for use of all interrupts in application
- Application can optionally reuse the low-level communication drivers from the bootloader or implement its own drivers
- Configurable entry sequence
- Optional validation of application using CRC-CCITT
- Source code is available, allowing for additional customizations

Source code for the bootloader with different sample configurations, application examples, and host examples is included to allow for an easy testing, customization, and implementation. Knowledge of I²C, UART, and SPI specifications, as well as sub-1 GHz RF communication protocol, is assumed.

1.1 Glossary

BOR	Brownout reset
BSL	MSP430 bootloader
CI	MSPBoot communication interface
CRC	Cyclic redundancy check
eUSCI	Enhanced universal serial communication interface
I ² C	Inter-Integrated Circuit
MCU	Microcontroller
MI	MSPBoot memory interface
MSPBoot	The bootloader described by this application report
MSP430FRBoot	The bootloader described by MSP430FRBoot - Main Memory Bootloader and Over-the-Air Updates for MSP430™ FRAM Large Memory Model Devices
OAD	Over-the-air download
OSI	Open systems interconnection
PUC	Power-up clear reset
ROM	Read-only memory
SPI	Serial peripheral interface
UART	Universal asynchronous receiver/transmitter
USCI	Universal serial communication interface
USI	Universal serial interface

1.2 Conventions

This document contains I²C transfer examples that use the following conventions:

	Master to Slave	S: Start
	Slave to Master	P: Stop
		Sr: Repeated Start
		A: Acknowledge (SDA low)
		/A: Not Acknowledge (SDA high)
		R: R/W bit =1
		W: R/W bit = 0

UART transfer examples use the following form:

	Host to Target	St: Start
	Target to Host	SP: Stop

SPI transfer examples use the following form:

	Master to Slave	X: Don't care
	Slave to Master	

2 Implementation

A modular approach was used to allow for an easy migration between MSP430 devices and allow for customization of each layer. [Figure 2-1](#) shows the software layers.

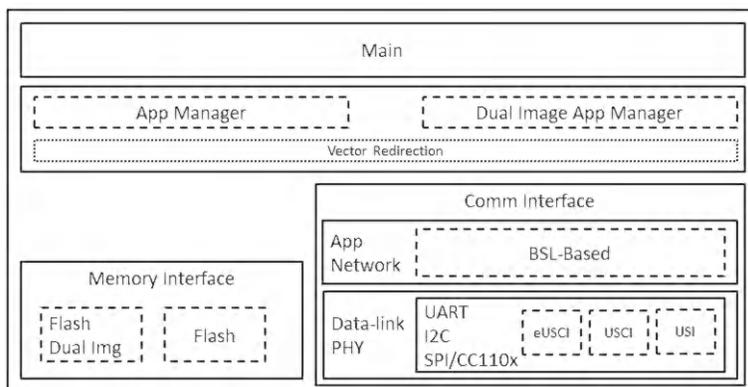


Figure 2-1. MSPBoot Software Architecture

Each module is described in more detail in the following sections.

2.1 Main

The main routine has the following purpose:

- Initialize basic functionality of the MSP430 MCU.
- Initialize the other MSPBoot layers.
- Implement the main loop that polls the communication interface and processes commands.

Figure 2-2 shows the state diagram of the main routine:

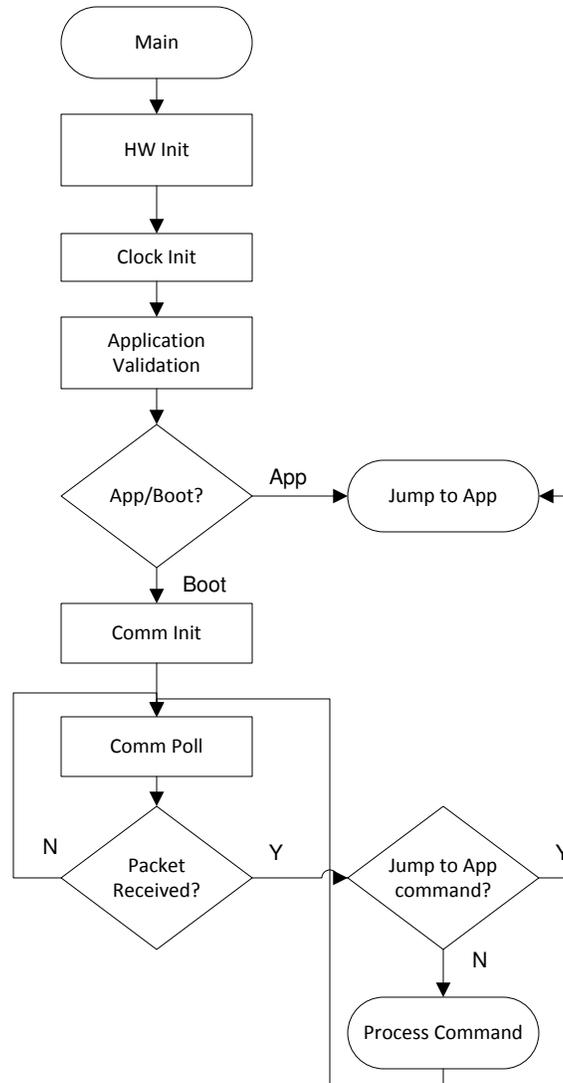


Figure 2-2. Flow Diagram of Main

2.2 Application Manager

The application manager has the main purposes of:

- Detecting when the device should be in bootloader mode or application mode
- Validating the application
- Redirecting interrupt vectors
- Jumping from bootloader to application
- Recovering a valid image when in Dual-Image mode

2.2.1 Boot and Application Detection

The Application Manager detects if the bootloader or the application should be executed by applying the following rules:

- Application is executed if:
 - The application is valid (see [Section 2.2.1.2](#))
- AND
- The bootloader is not forced by an external event or by application (see [Section 2.2.1.1](#)).
- Bootloader is executed if:
 - It is forced by an external event or by the application
- OR
- The application is invalid

Figure 2-3 shows this decision process.

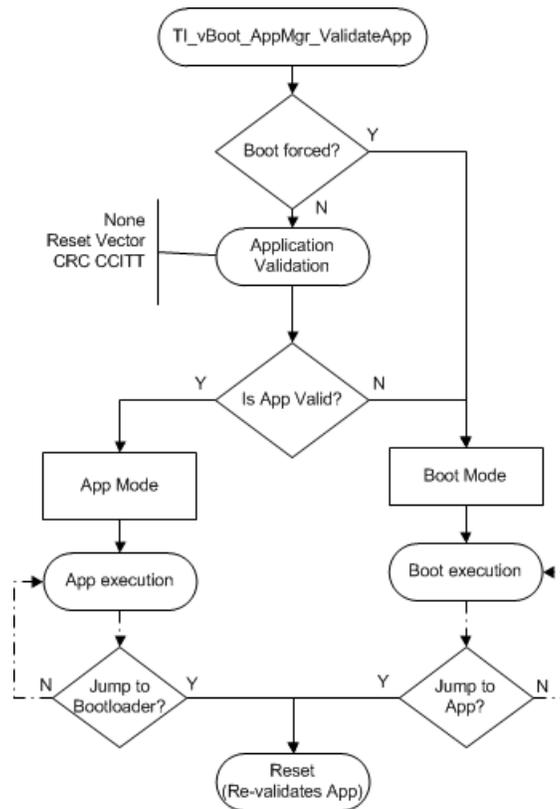


Figure 2-3. Application Validation by AppManager

2.2.1.1 Force Bootloader Mode

Even with a valid application, bootloader mode can be forced by these options:

- Option 1: An external event such as the state of a GPIO after reset.

By default, the software checks if the following GPIOs are low after reset to force bootloader mode:

- P1.3 in MSP430G2xxx (S2 button in MSP-EXP430G2)
- P1.1 in MSP430F5529 (S2 button in MSP-EXP430F5529)

This event can be modified as needed in `TI_MSPBoot_AppMgr_BootisForced()`.

- Option 2: An application calls execution of bootloader mode.

The variables StatCtrl and PassWd are reserved and shared between application and bootloader. To force bootloader mode, the application sets these variables to:

```
PassWd = 0xC0DE
```

```
StatCtrl.BIT0 = 1
```

2.2.1.2 Application Validation

The application validation mechanism allows the bootloader to validate the application before executing it. Three methods are implemented to allow for different levels of code footprint and security:

- None: Application is not validated and assumed to be always valid. An external event can be used to force bootloader mode. Not recommended.
- Reset vector: If the reset vector is different from 0xFFFF (erased state), the application is assumed to be valid and is executed.
- CRC_CCITT: A CRC CCITT is calculated for the whole application image and compared to an expected value. Note that the BSL-based protocol (see [Section 2.4.2.1](#)) uses CRC CCITT, so this validation method is recommended when using the BSL-based protocol. Additionally, this method does not take into account applications that modify the contents of flash such as data logging. When this type of application executes, the CRC CCITT stored in flash becomes incorrect and the bootloader will not allow the application to run even though it may be valid. When developing an application that modifies flash, it is recommended to use the reset vector validation method.

Note that the validation methods can prevent execution of corrupted applications, but they do not ensure the integrity and functionality of the application, which is the responsibility of the user. If the application does not have the intended functionality, the MSP430 device can still be recovered using a hardware entry sequence.

2.2.1.3 Jump to Application

MSPBoot forces a reset when the Communication Protocol detects that the download is complete and the device should jump to the application.

Devices that support software BOR (for example, MSP430F5529) use this method to force a reset, which is an efficient method to restore the MSP430 to a default state. This method is enabled when HW_RESET_BOR is defined.

Devices without this mechanism (for example, MSP430G2xx) use a PUC reset, which also forces a reset but does not clear all registers. The bootloader clears relevant registers when HW_RESET_PUC is defined.

2.2.2 Vector Redirection

MSPBoot cannot erase or reprogram the bootloader area. This limitation provides a more secure implementation, because the bootloader is always accessible, and the MCU can be recovered by forcing bootloader mode.

The reset vector is an integral part of the bootloader, because it forces the MCU to always jump to the bootloader entry sequence, and thus it should not be erased. Because the reset vector resides in the top of 16-bit flash space (0xFFFFE), the bootloader code is placed in the contiguous locations (see [Figure 2-4](#)).

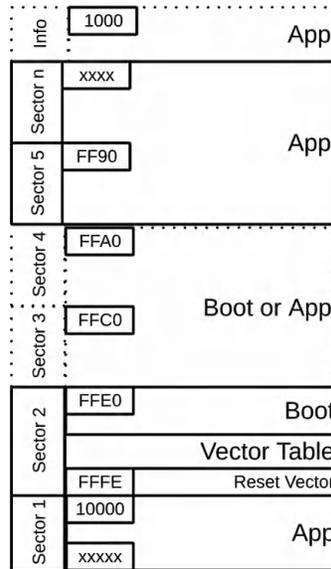


Figure 2-4. Memory Assignment

The interrupt vector table also resides in protected Boot area. Because the value of the interrupt table is expected to change based on the application, this means that special considerations must be followed to allow for application interrupts. Additional 20-bit space is available for the application in large memory model devices (0x10000 and above).

2.2.3 Interrupt Vectors in Flash Devices

The minimum size for a flash erase is a segment, which is 512 bytes in MSP430 MCUs. Given this consideration, the whole interrupt table is protected from erases. To allow interrupts on an application level, a software vector redirection method is implemented to fix the contents of the default vector table and point to a proxy table that resides in application space.

Figure 2-5 shows the concept behind this implementation:

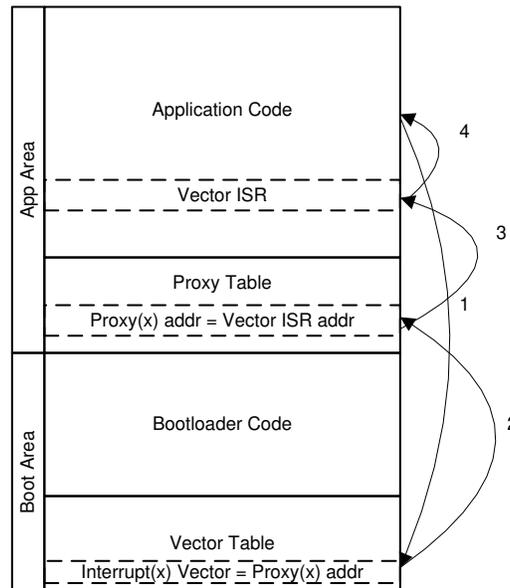


Figure 2-5. Vector Redirection Implementation

1. The application receives an interrupt request, the current address is pushed into the stack, and the CPU fetches the address from the vector table.
2. The vector table contains the address of the proxy location for each interrupt. The CPU fetches the address of the corresponding entry in the proxy table and jumps to it.
3. The proxy table contains branch instructions (BRA) followed by the actual address of each ISR. The CPU executes the BRA instruction and jumps to the corresponding application vector ISR.
4. Upon completion of the ISR, the RETI instruction is executed, and the previous address is popped from the stack.

This process is almost transparent for the implementation of an application, but it is important to note that there is added latency due to the additional jump from the proxy table to the application ISR.

Application examples that show how to implement interrupts are included in the sample code to demonstrate this functionality.

Note that some MSP430 MCUs support redirecting vectors to RAM in hardware (SYSRIVECT), which could be a good alternative for devices with sufficient RAM.

2.2.4 Dual Image Support

The Application Manager can also support dual image mode. In this mode, a valid application is always expected to reside in main memory even if an image download is interrupted or if a newly downloaded image is corrupt. This mode is critical for OAD where communication can be interrupted unexpectedly.

In dual image mode, the main memory is divided, creating a download area and application area as explained in [Section 2.3.1](#). The application validation process in this mode is different from the usual procedure, as shown in [Figure 2-6](#).

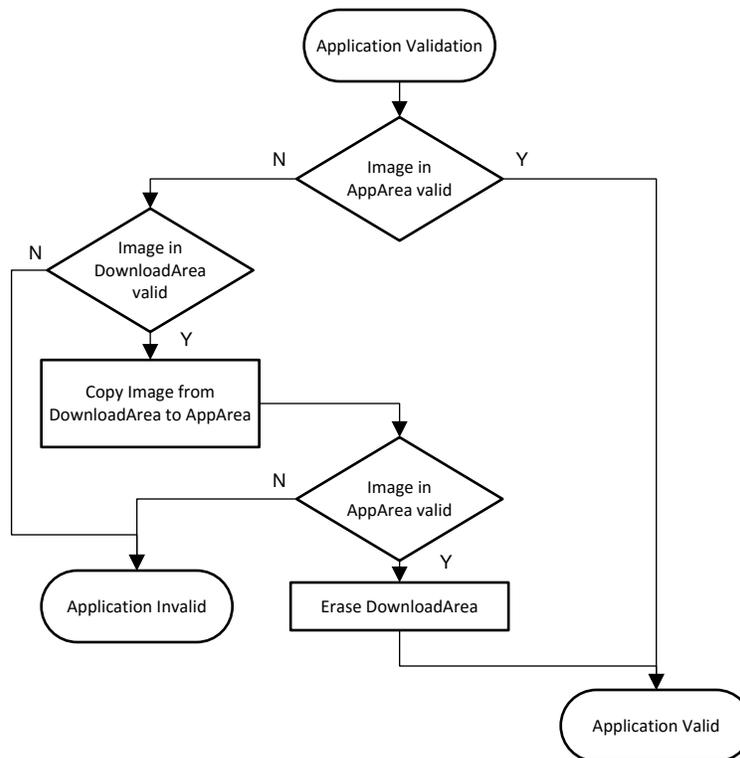


Figure 2-6. Dual Image Application Validation

2.2.4.1 Jumping to Application in Dual Image Mode

When an application download process is completed, MSPBoot performs the following steps before jumping to the new application:

1. Validate new image in download area
 - a. If invalid, exit (a reset forces bootloader again and executes the application only if the original image is valid)
 - b. If valid, continue
2. Replace application area with download area
3. Validate image in application area
 - a. If valid, erase the download area (a reset should execute application because the image in the application area is valid).
 - b. If invalid, exit (unexpected state, but a reset would revalidate both images again)

2.3 Memory Interface (MI)

To protect the bootloader area, the memory is logically partitioned in two sections:

- Application area: Writable section with user application and redirected vector table
- Bootloader area: Nonwritable section with bootloader and vector table

The size of each sector is defined in the project linker file. Examples showing different memory sizes are available in the example projects for the Code Composer Studio™ IDE (CCS). The memory interface provides an API that is used to program and erase the application memory area and protect the bootloader area. This memory protection is implemented as follows for flash devices:

- A mass erase is not performed, and the application is erased using segment erases.
- The address being erased or programmed is validated to avoid accidental corruption of bootloader area

Note

MSPBoot does not allow write or erase access to the bootloader area when executing updates, but it cannot protect against accidental erase when executing an application.

2.3.1 Dual Image Support

When dual image support is enabled, the Memory Interface module partitions the MSP430 application area in two subsections, resulting in the following logical memory map:

- Non-Boot Area
 - Download area: Section used as temporary buffer to store a new application image. Physical addresses in this area are inaccessible to the host, but this area is written when the host attempts to download to logical addresses in the application area.
 - Application area: Section used to execute the current application image. Logical addresses in this area are available to the host, but the host cannot write to the physical addresses. The bootloader updates this area when a new image in download area is validated. This procedure is explained in [Section 2.2.4](#).
- Boot area
 - Read-only section with bootloader and vector table.

The size of each sector is defined in the project linker file. Examples that show different memory sizes are available in the example projects for CCS.

2.4 Communication Interface (CI)

The purpose of the communication interface is to:

- Receive data from and send data to a host
- Implement a communication protocol
- Parse the data, validate a packet, and execute the appropriate command
- Based on the output of the function, generate a response

Following the open systems interconnection (OSI) model, the CI is divided in two modules:

- Physical-DataLink (PHY-DL)
- Network-Application (NWK-APP)

2.4.1 Physical-DataLink (PHY-DL)

The PHY-DL layer provides a hardware abstraction layer (HAL) to simplify the migration process to a different MSP430 derivative or peripheral. The PHY-DL layer provides a stable channel for sending data to and receiving raw data from the host. The current bootloader was originally implemented using I²C, UART, or SPI. It currently supports the USI, USCI, and eUSCI modules, but other options could be included if desired.

The PHY-DL layer is initialized by providing a pointer to a structure with the callback function in [Table 2-1](#).

Table 2-1. PHY-DL Callback Structure

t_CI_Callback	Structure Type Definition
.RxCallback	Called when a new byte is received
.TxCallback	Called when a byte needs to be transmitted
.ErrorCallback ⁽¹⁾	Called when an error is detected in PHY-DL (a time-out)
.StartCallback ⁽¹⁾	Called when the start of a packet is detected
.StopCallback ⁽¹⁾	Called when the end of packet is detected

(1) Callback is optional. The protocol or CI may not require a callback.

A higher level layer (NWK-APP) uses the callback functions to implement the communication protocol. Depending on the protocol, some callbacks are not required and they can be disabled in the PHY-DL layer to reduce the footprint. NWK-APP layer is described in [Section 2.4.2](#).

2.4.1.1 I²C

The I²C bus has been one of the most common communication protocols in embedded applications for many years, thanks to its low cost of implementation, robustness, and flexibility. It is a useful interface for low-cost applications that require a simple communication link. It is a perfect match for MSP430 microcontrollers, which combine high performance, high integration, and ultra-low power in a cost-effective solution. The I²C interface is implemented using 7-bit addressing with a predefined address for MSP430 MCUs:

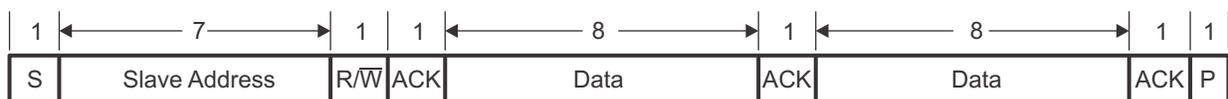


Figure 2-7. I²C 7-Bit Addressing Format

The predefined MSP430 MCU address is defined as:

CONFIG_CI_PHYDL_I2C_SLAVE_ADDR = 0x40

Files are included to support USI, G2x3 USCI, F5xx/6xx USCI, and eUSCI.

2.4.1.1.1 Time-out Detection

During I²C communication, it is valid for a slave to hold the clock line low when it needs more time to process a packet. This mechanism is known as clock stretching and, although it is very useful, it can also cause devices to hold the bus indefinitely, thus stalling the bus.

The PHY-DL layer can optionally detect when the lines are being held for too long and in such case, the PHY-DL layer can reset the interface.

This feature is enabled depending on CONFIG_CI_PHYDL_TIMEOUT. USCI and USI implementations use TA1 to implement this feature, while eUSCI includes hardware support for it.

2.4.1.2 UART

The UART interface is implemented using 8-N-1 format (8 data bits, no parity bit, 1 stop bit) (see Figure 2-8).

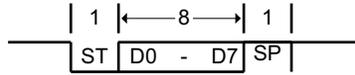


Figure 2-8. UART 8-N-1 Format

The default baud rate is defined as CONFIG_CI_PHYDL_UART_BAUDRATE = 9600.

2.4.1.3 SPI

The SPI interface used for CC110x communication is implemented using the following configuration (also see Figure 2-9):

- 8-bit data
- MSB first
- Clock polarity = 0 (inactive state is high)
- Clock phase = 1 (data changed on first clock edge, captured on following edge)
- 3-pin configuration with SS implemented using GPIO

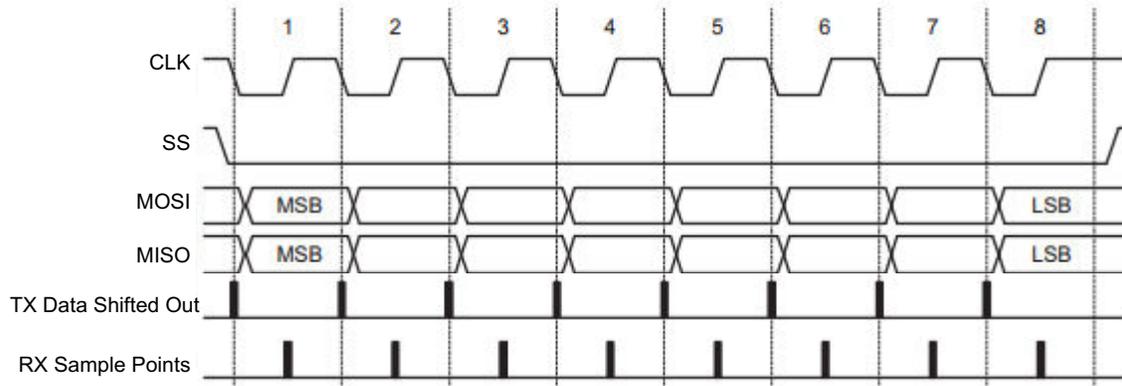


Figure 2-9. SPI Format

2.4.1.4 CC110x

The CC110x devices send data using the packet structure in [Table 2-2](#).

Table 2-2. CC110x Data Packet Structure

Header	Length ⁽¹⁾	Command	Address ⁽²⁾	Data ⁽²⁾	Checksum
0x80	N	1 byte	3 bytes	N-6 bytes	2 bytes

(1) The maximum packet length is 24 bytes; therefore the most data bytes (N) allowed per packet is 16.

(2) If the length is equal to one (command only), these subsections are not included in the packet.

The configuration for the CC110x in MSPBoot is as follows:

- 250kbps data speeds
- Carrier frequency of 902750Hz

The data speed can be set to either 1.2 or 38.4kbps through the variable sent by the `radio_init` function inside of `TI_MSPBoot_CI_PHYDL_CC1101.c`. Radio frequency can be altered in `TI_MSPBoot_Config.h`. Changes must be made to both the target and host firmware project. See the [CC1101 Low-Power Sub-1 GHz RF Transceiver](#) data sheet for more information regarding common CC1101x command and other communication details.

The packet structure is identical to the BSL-based protocol, therefore it can be directly transferred from the PHY-DL to the NWK-APP layer with the expected formatting. This is also described in [Section 2.4.2.1.2](#).

2.4.1.5 Comm Sharing

The user application can use the communication interface as desired (I²C, UART, GPIO, or other purpose), because the resources are released when the MCU jumps to the application. Optionally, the CI PHY-DL can be shared with the application, which allows it to use the same communication interface and reduce the application footprint. When this feature is enabled, the bootloader shares the function pointers from [Table 2-3](#).

Table 2-3. Boot2App_Vector_Table Definition

Boot2App_Vector_Table	Table With Addresses of Shared CI PHY-DL Functions
<code>TI_MSPBoot_CI_PHYDL_Init</code>	Function used to initialize PHY-DL passing a pointer to an application <code>t_CI_Callback</code> .
<code>TI_MSPBoot_CI_PHYDL_Poll</code>	This function checks all relevant flags and calls corresponding callbacks when required
<code>TI_MSPBoot_CI_PHYDL_TxByte</code> ⁽¹⁾	Function used to write the TX buffer

(1) Callback is implemented for SPI and UART only, and is not required for I²C.

The application must declare its own callbacks, which are passed during initialization of CI PHY-DL and called when the corresponding event is detected. The PHY-DL layer is designed with small footprint being a top priority. The application can always implement its own drivers if the PHY-DL implementation is inadequate. The Application 2 examples in the accompanying software package show how to share CI PHY-DL.

2.4.2 NWK-APP

The CI Network-Application layer implements the communication protocol, interpreting the raw data from PHY-DL, and validates such data before executing the appropriate commands. For means of simplicity, MSPBoot uses only the BSL-based protocol.

2.4.2.1 BSL-Based Protocol

The MSP430 BSL is the standard bootloader that is included in MSP430 MCUs. It is described in detail in the [MSP430™ Flash Device Bootloader \(BSL\) User's Guide](#).

The BSL-based protocol that is implemented in MSPBoot maintains robustness, but it does not implement all of the commands and exactly the same format as the BSL protocol to reduce its footprint. The protocol is *packet based* and has the format in [Table 2-4](#).

Table 2-4. BSL-Based Protocol Command Format

Header	Length	Payload	Checksum [L]	Checksum [H]
0x80	1 to PAYLOAD_MAX_SIZE ⁽¹⁾	1 to PAYLOAD_MAX_SIZE Bytes	1 Byte	1 Byte

(1) PAYLOAD_MAX_SIZE is set to 20 by default (1 CMD + 3 Addr + 16 Data)

Header: Fixed to 0x80

Length: 1 byte with the length of the payload. Valid values are 1 to PAYLOAD_MAX_SIZE.

Payload: One to PAYLOAD_MAX_SIZE bytes containing a command, optional address, and data (optional depending on command type).

Checksum: 16-bit CRC CCITT of the payload

The commands in [Table 2-5](#) are implemented as a payload.

Table 2-5. BSL-Based Protocol Commands

Command	CMD	Byte ₁	Byte ₂	Byte ₃	Byte ₄	...	Byte _{length-1}
ERASE_SEGMENT	0x12	ADDR[L]	ADDR[M]	ADDR[H]	X	X	X
ERASE_APP	0x15	X	X	X	X	X	X
RX_DATA_BLOCK	0x10	ADDR[L]	ADDR[M]	ADDR[H]	DATA0	X	DATA _n
TX_VERSION	0x19	X	X	X	X	X	X
JUMP2APP	0x1C	X	X	X	X	X	X

ERASE_SEGMENT

Erases the memory segment (512 bytes in flash) addressed by ADDR.

ERASE_APP

Erases the application area.

RX_DATA_BLOCK

Programs *n* bytes of data starting at address ADDR.

TX_VERSION

Requests the MSPBoot version from the target.

JUMP2APP

Instructs the target to jump to the application image (after validation). Responses from the target are always a single byte ([Table 2-6](#) lists the valid values).

Table 2-6. BSL-Based Protocol Slave Response

Response	Value	Description
OK	0x00	Previous command executed correctly
HEADER_ERROR	0x51	Frame had incorrect header
CHECKSUM_ERROR	0x52	Frame checksum incorrect
PACKETZERO_ERROR	0x53	Length of packet = 0
PACKETSIZE_ERROR	0x54	Length of packet > MAX_LEN
UNKNOWN_ERROR	0x55	Error in Protocol
INVALID_PARAMS	0xC5	Parameters received for command are incorrect
INCORRECT_COMMAND	0xC6	Received Command is not valid
MSPBOOT_VERSION	0 to 0xFF	Sent as response for TX_VERSION command (default is 0xA0)

2.4.2.1.1 Security

The contents of each packet are validated with a 16-bit CRC that provides additional robustness to the bootloader. The host can check the result of each command and retry if the previous command was unsuccessful.

The ERASE_SEGMENT and RX_DATA_BLOCK commands can erase and write any area within the 16-bit memory map, thus potentially corrupting the bootloader. To avoid this possibility, TI recommends including the CONFIG_MI_MEMORY_RANGE_CHECK MI definition to validate the address before a program or erase

operation. The application areas can be corrupted if the process is interrupted, so TI recommends using one of the application validation methods described in [Section 2.2.1.2](#) or use the dual-image approach.

2.4.2.1.2 BSL-Based Protocol using CC110x

The CC110x implementation of this protocol follows the same guidelines used for UART, but includes slight changes since the information is received in complete packets instead of bytes. Because the incoming CC110x packet outlined in [Table 2-2](#) is the same as the expected BSL-based protocol in [Table 2-4](#), data from the PHY-DL layer can be directly transferred to the NWK-APP without the need for conversion.

2.4.2.1.3 Examples Using I²C

The following considerations apply when using I²C with BSL-based protocol:

- The host always starts a transfer (R or W).
- The packet must contain the address of the slave device. All other addresses will be ignored.
- If the slave device is not ready to process the data or send a response, it will hold the clock low (known in I²C as "stretching the clock")
- Note that different commands have different processing times
- Example: Host reads version from the MCU

S	0x40	W	A	0x80	A	0x01	A	0x19	A	0xE8	A	0x62	A	P
Addr		Header			Length		TX_VERSION		Checksum_L		Checksum_H			

S	0x40	R	A	0xA0	/A	P
Addr		Version				

- Example: Host writes 16 bytes to address 0xC000.

S	0x40	W	A	0x80	A	0x14	A	0x10	A	0x00	A	0xC0	A	0x00	A	0x03	A	0xEE	A
Addr		Header			Length		RX_DATA_BLOCK		AddrL		AddrM		AddrH		Data0		Data1		

0x47	A	0xFF	A	0xB2	A	0x40	A	0x80	A	0x5A	A	0x20	A	0x01	A	0xD2	A
Data2		Data3		Data4		Data5		Data6		Data7		Data8		Data9		Data10	

0xD3	A	0x22	A	0x00	A	0xD2	A	0xD3	A	0x15	A	0xE4	A	P
Data11		Data12		Data13		Data14		Data15		Checksum_L		Checksum_H		

S	0x40	R	A	0x00	/A	P
Addr		OK				

2.4.2.1.4 Examples Using UART or CC110x

The following considerations apply when using UART with BSL-based protocol:

- Address is not required since communication is expected to be point-to-point.
- All bytes in UART are in 8-N-1 format as described in [Section 2.4.1.2](#).
- The target responds with the result of the command when ready and not when requested by the host
- The host should wait for the response from the target after sending a command, preferably with a time-out.
- Different commands have different processing times.
- Example: Host erases the MCU application area

0x80	0x01	0x15	0x64	0xA3
Header	Length	ERASE_APP	Checksum _L	Checksum _H

The target device will process the command and respond with the result when ready.

0x00
OK

The same considerations apply when using a CC110x with BSL-based protocol. The exception to this is [Section 2.4.2.1.2](#) where it is stated that all bytes received in packets from the CC110x are in the same format expected for BSL-based protocol; therefore, they can be directly transferred from the PHY-DL to the NWK-APP. Although processing times are the same, over-the-air communication can be expected to be slightly slower than UART, and host wait times should be lengthened to compensate.

3 Customization of MSPBoot

MSPBoot was designed with low cost and a small footprint as top priorities; however, some applications require or can benefit from having a higher level of security and robustness. Based on the application requirements, different levels of customizations are available in the MSPBoot code, and they can be adjusted to particular needs. These options are either selected by adding the appropriate files or by enabling and disabling certain pre-processor definitions. [Table 3-1](#) lists the options that can be configured in TI_MSPBoot_Config.h.

Table 3-1. Optional Configurations

Value	Description	Effect on Code Size
NDEBUG		
Defined	ASSERT_H functions are ignored. Watchdog is enabled.	–
Undefined	Used during debugging. ASSERT_H functions are checked. Watchdog is disabled.	Adds approximately 20 bytes
CONFIG_MI_MEMORY_RANGE_CHECK		
Defined	The address being erased or programmed is validated to be within the Application area.	Adds approximately 44 bytes
Undefined	Address being erased or programmed is not validated. Host must send correct address.	–
CONFIG_APPMGR_APP_VALIDATE		
1	Application is not validated	-
2	Application is validated by checking its CRC-CCITT.	Adds approximately 6 bytes
CONFIG_CI_PHYDL_COMM_SHARED		
Defined	Communication Interface PHY-DL layer is shared with application.	Adds approximately 28 bytes
Undefined	CI PHY-DL is not shared with application.	–
CONFIG_CI_PHYDL_I2C_TIMEOUT		
Defined	Detect time-out in CI PHY-DL.	Adds approximately 48 to 62 bytes
Undefined	CI PHY-DL does not detect time-out.	–

Table 3-1. Optional Configurations (continued)

Value	Description	Effect on Code Size
CONFIG_CI_PHYDL_START_CALLBACK		
Defined	A callback function is called when Start is detected (required only for some protocols or Communication interfaces).	Adds approximately 12 bytes
Undefined	Callback function is not called when Start is detected.	–
CONFIG_CI_PHYDL_STOP_CALLBACK		
Defined	A callback function is called when Stop is detected (required only for some protocols or Communication interfaces).	Adds approximately 38 to 54 bytes
Undefined	Callback function is not called when Stop is detected.	–
CONFIG_CI_PHYDL_ERROR_CALLBACK		
Defined	A callback function is called when a time-out error or an error is detected (only for some protocols or Communication interfaces).	Adds approximately 16 to 20 bytes
Undefined	Callback function is not called when a time-out error is detected.	–
CONFIG_CI_PHYDL_CC1101_FREQUENCY		
Defined	Defines the frequency of CC110x communication	-
Undefined	-	-
CONFIG_CI_PHYDL_UART_BAUDRATE		
Defined	Defines the baud rate of UART communication	-
Undefined	-	-
CONFIG_CI_PHYDL_I2C_SLAVE_ADDR		
Defined	Defines the address the MSP430 will respond to when using I ² C communication	-
Undefined	-	-

Other customizations are selected by adding and using the appropriate files in the project. [Table 3-2](#) lists the files that are interchangeable in the project.

Table 3-2. Customization Files

CI PHY-DL	
TI_MSPBoot_CI_PHYDL_USI_I2C_Slave.c	Use USI as I ² C slave
TI_MSPBoot_CI_PHYDL_USCI_I2C_Slave_x2xx.c	Use USCI as I ² C slave on x2xx devices
TI_MSPBoot_CI_PHYDL_USCI_I2C_slave.c	Use USCI as I ² C slave
TI_MSPBoot_CI_PHYDL_USI_I2C_slave.c	Use USI as I ² C slave
TI_MSPBoot_CI_PHYDL_eUSCI_I2C_slave.c	Use eUSCI as I ² C slave
TI_MSPBoot_CI_PHYDL_USCI_UART_x2xx.c	Use USCI as UART on x2xx devices
TI_MSPBoot_CI_PHYDL_USCI_UART.c	Use USCI as UART
TI_MSPBoot_CI_PHYDL_eUSCI_UART.c	Use eUSCI as UART
TI_MSPBoot_CI_PHYDL_CC1101.c	Use CC110x
MI	
TI_MSPBoot_MI_Flash_20Bit.c	API used to program application flash in large memory model devices
TI_MSPBoot_MI_FlashDualImg_20Bit.c	API that implements dual-image in flash in large memory model devices
TI_MSPBoot_MI_Flash_16Bit.c	API used to program application flash in small memory model devices
TI_MSPBoot_MI_FlashDualImg_16Bit.c	API that implements dual-image in flash in small memory model devices
App Manager	
TI_MSPBoot_AppMgr.c	Standard app manager
TI_MSPBoot_AppMgrDualImg_20Bit.c	App manager that supports dual image in large memory model devices
TI_MSPBoot_AppMgrDualImg_16Bit.c	App manager that supports dual image in small memory model devices

3.1 Predefined Customizations

The software package includes projects for CCS that supports two devices (MSP430F5529 and MSP430G2553) with three communication interfaces (UART, I²C, or SPI with CC110x), and two predefined configurations (single image, dual image) per device. In the provided CCS examples, devices and communication interfaces are separated by project selection and the predefined configurations can be chosen under Project → Build Configurations → Set Active.

4 Building MSPBoot

This section is a step-by-step guide that describes how to build the bootloader and demo applications for a target device.

[Section 4.2](#) describes how to build and use the example applications to run a demo.

4.1 Starting a New Project

The software package includes the following:

- **Host_Examples:** Examples showing the MSP-EXP430F5529 or the MSP-EXP430G2 being used as a host that programs an MSP430 flash-based target through a UART, I²C, or SPI with CC110x communication interface.
- **MSP430F5529_Examples:** Examples showing the MSP430F5529 being used as an MSPBoot target. This includes the bootloader code and two example applications for each respective communication interface (UART, I²C, or SPI with CC110x).
- **MSP430G2553_Examples:** Examples showing the MSP430G2553 being used as an MSPBoot target. This includes the bootloader code and two example applications for each respective communication interface (UART or I²C).
- **Utilities:**
 - **430 TXT Converter:** Perl script used to convert CCS output files to Host TargetApps. See [Section 4.1.2.1](#).
 - **Project_Creator:** Script used to automatically generate a CCS project script file used to create a new MSPBoot project, see [Section 4.1.1.1](#).

This software package was built and tested using CCS 7.2.0. Other IDE versions and compilers may not directly support the resources as provided and could require slight modifications.

4.1.1 Creating a New MSPBoot Project

While the examples provided in the accompanying software package are a great starting place when introduced to MSPBoot, there may be a desire to use a different flash based MSP430 device variant. The accompanying software package includes a project generation script that generate a CCS project script file that can be used to create a new MSPBoot project. After creation, several files need to be modified with user input to ensure the newly created source code will work with the MSP430 device variant being used (See [Section 4.1.1.3](#)).

4.1.1.1 MSPBootProjectCreator.pl

MSPBootProjectCreator: Generates all bootloader and application source files necessary to start creating an MSPBoot project in CCS.

Note

A Perl interpreter is required to run this script. Visit <https://www.perl.org/> to download an interpreter if needed.

Location: Utilities/Project Creator/MSPBootProjectCreator.pl

Syntax:

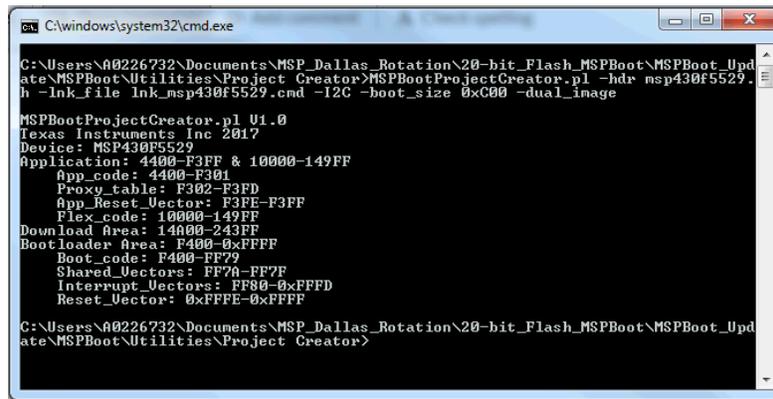
```
[ ] denotes an optional field
All numbers should be in hexadecimal format
MSPBootProjectCreator.pl
    [-help]
    -hdr <header_file>
    -lnk_file <lnk_msp430.cmd>
    [-boot_size <size>]
    [-shared_vectors <number>]
    [-dual_image]
    [-I2C]
    [-UART]
```

Where (all numbers are hex values):

- -hdr <header_file> = Specifies the header file for the MSP430 device variant. Needs to be in the same directory as MSPBootProjectCreator.pl. The default path to this file is C:\ti\ccsvx\ccs_base\msp430\include, where ccsvx is the ccs version being used.
- -lnk_file <lnk_msp430.cmd> = Specifies the default linker file for the device being used. Needs to be in the same directory as MSPBootProjectCreator.pl. The default path to this file is C:\ti\ccsvx\ccs_base\msp430\include, where ccsvx is the ccs version being used.
- -boot_size <size> = Optional parameter. Specifies the size of the bootloader area. Only increments of 0x400 are allowed. If omitted, the default bootloader size will be used.
- -shared_vectors <number> = Optional parameter. Specifies the number of shared vectors (in hex). If not specified, default value is 3 vectors.
- -dual_image = Optional parameter. Specifies that the files created should enable a dual image bootloader. If not specified, single image is assumed.
- -I2C, -UART = Specifies the communication interface for the bootloader. One of these must be specified.

Note

SPI with CC110x is not supported when using the generator script because it has many device specific dependencies. If trying to develop an OAD application, see the examples provided in the accompanying software package.



```

C:\windows\system32\cmd.exe

C:\Users\A0226732\Documents\MSP_Dallas_Rotation\20-bit_Flash_MSPBoot\MSPBoot_Update\MSPBoot\Utilities\Project_Creator>MSPBootProjectCreator.pl -hdr msp430f5529.h -lnk_file lnk_msp430f5529.cmd -l2c -boot_size 0xC00 -dual_image

MSPBootProjectCreator.pl U1.0
Texas Instruments Inc 2017
Device: MSP430F5529
Application: 4400-F3FF & 10000-149FF
  App_code: 4400-F301
  Proxy_table: F302-F3FD
  App_Reset_Vector: F3FE-F3FF
  Flex_code: 10000-149FF
Download Area: 14000-243FF
Bootloader Area: F400-0xFFFF
  Boot_code: F400-FF79
  Shared_Vectors: FF7A-FF7F
  Interrupt_Vectors: FF80-0xFFFFD
  Reset_Vector: 0xFFFE-0xFFFF

C:\Users\A0226732\Documents\MSP_Dallas_Rotation\20-bit_Flash_MSPBoot\MSPBoot_Update\MSPBoot\Utilities\Project_Creator>
  
```

Figure 4-1. Example Command

This script uses templates that are located in the Linker_Templates, Vector_Templates, and ProjectSpec_Templates folders. The contents of these templates can be modified as needed but they are required to run the script. This script also uses the code located in the Src folder. Again, the contents of these files can be modified as needed, but their names and file paths must remain the same for proper operation of the script.

4.1.1.2 Importing Project Spec File in CCS

Once MSPBootProjectCreator has been executed and your project spec file created, you'll need to use that file in CCS to create a project.

1. Start by opening CCS and navigating to Project → Import CCS Projects...
 - a. Browse to the project spec file that was generated by MSPBootProjectCreator.pl
 - b. Determine which application template you will be using and select the appropriate projects
 - i. App_Shared_Comm: Shows how to share a communication interface with the bootloader
 - ii. App_Simple: Does not share a communication interface with the bootloader
 - c. click Finish

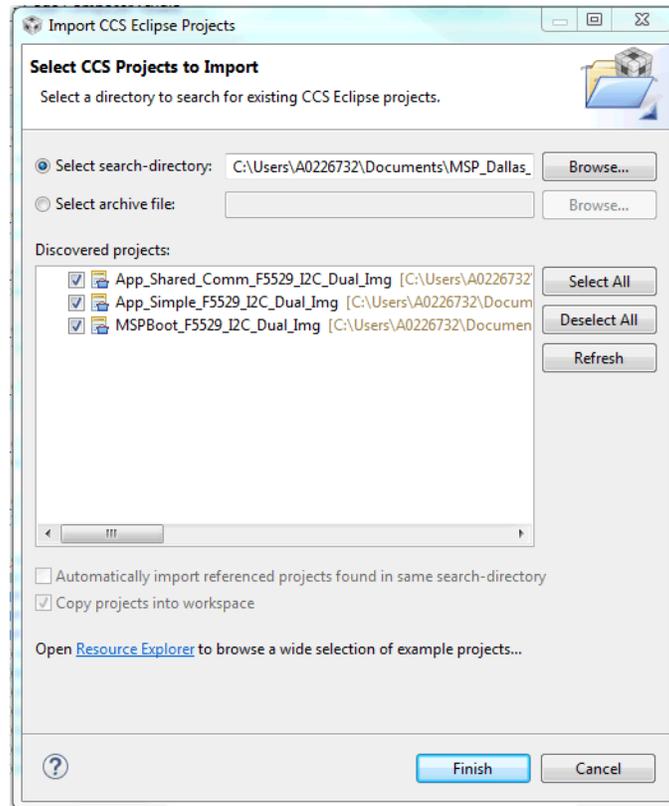


Figure 4-2. Importing Project Spec File in CCS

2. Delete the default linker command file from each project. This file will be titled `Ink_<device variant>.cmd`.
3. After modifying the appropriate source code as specified in [Section 4.1.1.3](#), you can build the project and program it on the MSP430 device.

4.1.1.3 Modifying Generated Source Code

When using MSPBootProjectCreator, several files require user modification after their creation. These files include:

- In MSPBoot:
 - main.c
 - TI_MSPBoot_Config.h
 - Comm/PHY_DataLink/TI_MSPBoot_CI_PHYDL_XXXX_XXX.c
 - AppMgr/TI_MSPBoot_AppMgr.c
- In Application(s):
 - main.c
 - TI_MSPBoot_Mgr_Vectors_XXXX.c

The portions of these files that need to be modified are marked with the //TODO: tag that is easily searchable and stands out in CCS. Each TODO also shows up in the CCS task list, which can be accessed through View → Other... → General → Tasks. Each task includes instructions on what must be modified and where to look for examples. This modification is required for proper operation of MSPBoot and application code when using the project generation script.

4.1.1.3.1 Modifying MSPBoot Main.c

There are three TODO items in the MSPBoot main.c that require user modification:

- Define a Debug Interface
 - For example, this could be an LED that turns on when in the bootloader, or a GPIO that enters a specific state
- Initialize the Clock System
 - By default, the examples includes with MSPBoot setup the system frequency to 8 MHz. MSPBoot can operate at 8MHz, 4MHz, or 1 MHz. Refer to the examples provided in the accompanying software package and the device specific user's guide for more details.
- Initialize Hardware
 - When the device enters the bootloader code, peripheral registers that are not initialized to their default values can affect operation. For example, a timer that was setup by the application to interrupt every second would affect MSPBoot operation. For this reason, TI highly recommends initializing all peripheral registers to their default values. If the bootloader was entered after a BOR, all peripheral registers are already set to their default values.

4.1.1.3.2 Modifying TI_MSPBoot_Config.h

There are four TODO items in TI_MSPBoot_Config.h that require user modification:

- Define the MCLK Frequency
 - After modifying main.c other portions of the code need to know the MCLK frequency to operate properly.
- Setup Watchdog Interval
 - In some applications, there may be a need to reset the device if the bootloader becomes unresponsive. The interval length is most dependent on the size of flash for the device being programmed. The more flash space, the longer the interval must be. For example, in the MSP430F5529 that contains 128KB of flash an interval greater than 6 seconds. If using the watchdog timer, the NDEBUG define also must be uncommented.
- Setup the Hardware Entry Condition
 - Typically a bootloader offers a way to enter the code through a hardware condition. In MSPBoot, the user can decide based on the status of a pin being high or low.

- Uncomment Configuration Options
 - There are many different configuration options that need to be selected based on the communication interface being used. You can also find more information on this in [Table 3-1](#).
 - When using I²C:
 - Uncomment CONFIG_CI_PHYDL_START_CALLBACK and CONFIG_CI_PHYDL_I2C_SLAVE_ADDR
 - When using UART:
 - Uncomment CONFIG_CI_PHYDL_UART_BAUDRATE
 - When using CC110x:
 - Uncomment CONFIG_CI_PHYDL_CC1101_FREQUENCY

4.1.1.3.3 Modifying TI_MSPBoot_CI_PHYDL_xxxx_xxx.c

There are three TODO items in TI_MSPBoot_CI_PHYDL_xxxx_xxx.c that require user modification:

- Define the communication module
 - Depending on the device and communication protocol, several peripherals in the MSP430 can achieve the same result. For example in the MSP430F5529 the USCI_A0 and USCI_A1 modules can both perform UART communication. The user needs to modify the defines associated with this TODO so the proper peripheral registers are being used for communication based on the application requirements.
- Define the timer module (USCI and USI I²C Only)
 - In I²C communication an optional time-out can be configured that takes advantage of the timer module within various MSP430 MCUs. The user needs to modify the defines associated with this TODO so an appropriate timer is being used based on the application requirements.
- Set up GPIO pins
 - Depending on the communication interface and the specific module being used, the user needs to set up the GPIO to the proper peripheral functionality. For more information on how to set up a pin for a communication interface, see the device-specific data sheet.

4.1.1.3.4 Modifying TI_MSPBoot_AppMgr.c

There is one TODO item in TI_MSPBoot_AppMgr.c that require user modification. In the TI_MSPBoot_Restore_DefaultClockSettings function, it is recommended that the user set the clock settings to default as if the part had just reset. This function is called just before jumping to the application code and will ensure that the MSPBoot clock settings are not carried over to the application.

4.1.1.3.5 Modifying Application Main.c

There are a varying number of TODO items depending on whether the simple application template or the shared communication application template is used. However, each includes TODO items for adding user defined values, functions and their prototypes, and interrupt service routines. These TODO items simply need to be replaced with the corresponding user code. Each template also provides access to the function TI_MSPBoot_JumpToBoot which allows the application code to jump to the bootloader.

4.1.1.3.6 Modifying TI_MSPBoot_Mgr_Vectors_xxxx.c

There are two TODO items in TI_MSPBoot_VecRed_xxxx_App.c that require user modification:

- Add ISR Prototypes
 - For vector redirection to work properly, add extern prototypes of every ISR used in the application. These are used in the next TODO.
- Update Proxy Vector Table
 - To update the proxy interrupt vector table with the ISRs used in your application, replace the "Dummy_ISR" text with the corresponding ISR function name. The comments next to each line indicate what peripheral the ISR is associated with.

4.1.2 Loading Application Code With MSPBoot

When creating custom applications to load onto the MSP430 using MSPBoot, perform the following steps for best results:

1. Develop the application without using MSPBoot.
 - a. This includes creating a project, using the default linker file, and developing code as if a main memory bootloader will not be used.
2. Once you have developed the application, transfer the code to one of the application templates.
 - a. App_Simple: Does not share a communication interface with the bootloader
 - b. App_Shared_Comm: Shows how to share a communication interface with the bootloader
3. Modify the vector redirection file as described in [Section 4.1.1.3.6](#).
4. Edit the project properties to output a TI-TXT hex format file.
 - a. Project Properties → MSP430 Hex Utility → Enable MSP430 Hex Utility
 - b. Project Properties → MSP430 Hex Utility → Output Format Options → Output TI-TXT hex format (--ti_txt)
5. Build the project.
6. Generate a C file that can be loaded from a host processor using the TI-TXT file found in the project Debug folder (see [Section 4.1.2.1](#)).
7. Load MSPBoot onto the target device.
8. If the target device is not already executing bootloader code, the target device must be forced to enter the bootloader. This can be done by setting up the application code to jump to the bootloader when a certain command is received. See the examples provided for more information on how to accomplish this.
9. Load the application C file onto the target device.
 - a. See the example host projects included in the accompanying software package for more information.

4.1.2.1 Convert Application Output Images

The CCS projects can be setup to generate outputs in MSP430 .txt format or Intel .hex format by following Project Properties → MSP430 Hex Utility for more information. Neither file includes a CRC, but one can either be calculated by the host processor or needs to be added manually to the generated file. Regardless of how the CRC is calculated, the application image must be converted to a format useable by the host processor. To make this easier, the software package includes image2C, a Perl script used to convert an MSP430 .txt file or Intel .hex file to a C array.

Location: MSPBoot\Utilities\430 Image Converter\image2C.pl

Syntax:

```
[ ] denotes an optional field
image2C.pl
           [-help]
           -src <src_file>
           -dst <dest_file>
           -struct <array_name>
           [-20_bit]
```

- -src <src_file> = Specifies the source file in .txt or .hex format.
- -dst <dest_file> = Specifies the destination file in .c format.
- -struct <array_name> = Name of the array in C file. If using the host examples provided in the accompanying software package, TI recommends naming the struct App1 or App2.
- -20_bit = Optional parameter. Specifies that the files created are compatible with large memory model (20 bit) MSP430 devices. If using a large memory model device, this command is required for the file to be generated correctly. If not specified, single image is assumed.

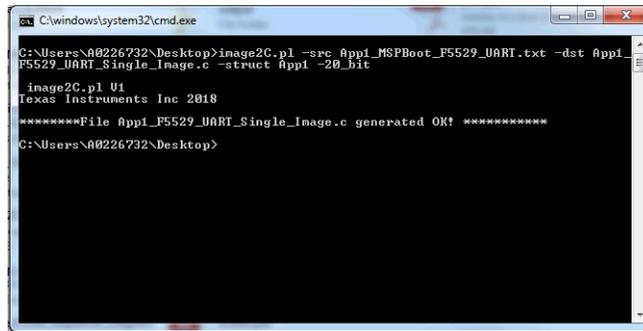


Figure 4-3. image2C Example

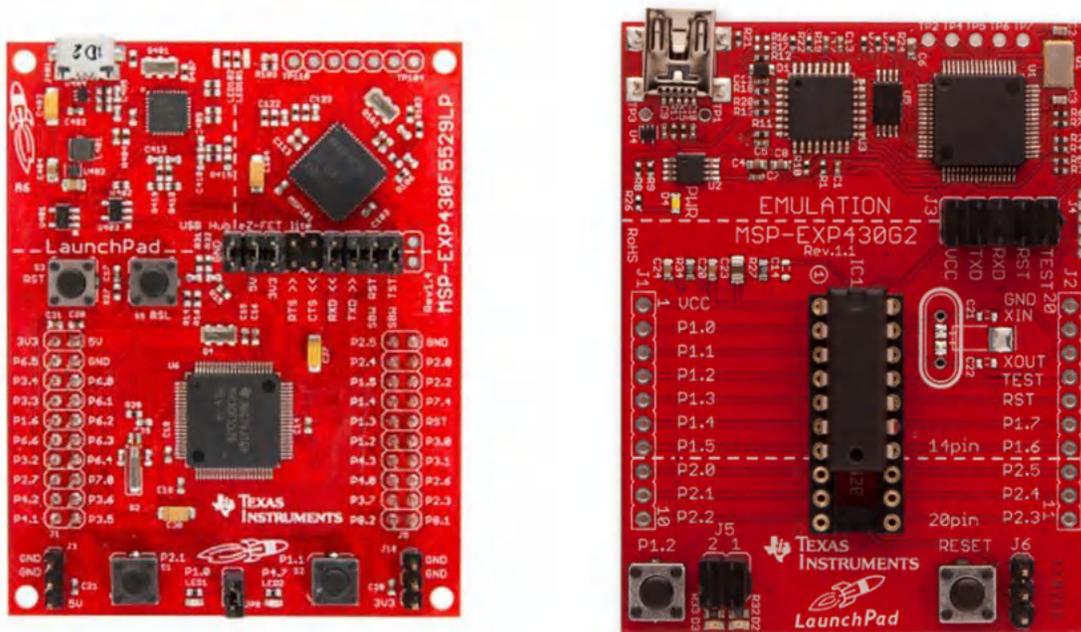
Note

A Perl interpreter is required to run this script. Visit <https://www.perl.org/> to download an interpreter if needed.

4.2 Examples

This software package includes examples for MSP430G2553 and MSP430F5529. Any MSP430 board can be used, but the LaunchPad™ development kit MSP-EXP430G2 and MSP-EXP430F5529 are the examples used in this application report.

4.2.1 LaunchPad Development Kit Hardware



- A. P2.0 is not connected to LED2 by default in MSP-EXP430G2. An external connection can be added for demo purposes when using I²C communication
- B. Jumper J5 that connects P1.6 to LED2 must be removed in MSP-EXP430G2 when using I²C communication.

Figure 4-4. Target Boards: MSP-EXP430F5529 and MSP-EXP430G2

The bootloader and demo applications use the same LED (LED1 and LED2) notations across all variants of the LaunchPad development kits. The pin assignments that correspond to these I/O peripherals are different for each board derivative. For each use, the examples have been designed so that the host and target LaunchPad development kits can be the same derivative, although there are examples and this can be modified for different configurations if desired.

4.2.2 CC110x Hardware

Two hardware options are available for using the CC110x communication with the MSPBoot examples. The first is a combination of the CC1101EMK868-915 and BOOST-CCEMADAPTER, but the simplest solution is with a 430BOOST-CC110L. Figure 4-5 shows both options.

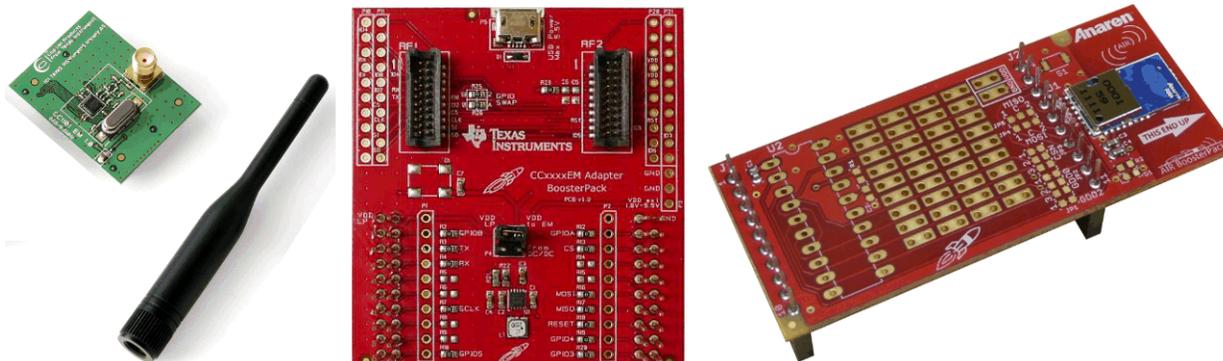


Figure 4-5. CC1101EMK868-915, BOOST-CCEMADAPTER, and 430BOOST-CC110L

Two units of either option are required, one for the host device and the other for the target. Both solutions are compatible across all LaunchPad development kits and are directly connected such that no other hardware is required to run the provided examples. More information about the ecosystem for the LaunchPad development kits and BoosterPack™ plug-in modules can be found at [TI LaunchPad Development Kits](#).

CAUTION

The 430BOOST-CC110L and the CC1101EMK868-915 use different reference crystals to generate the appropriate frequencies. When using the CC1101EMK868-915 with the BOOST-CCEMADAPTER the reference crystal definition found in `hal_spi_ref_exp5529.h` must be modified from 27000 to 26000 to compensate for the change from a 27-MHz crystal on the 430BOOST-CC110L to a 26-MHz crystal on the CC1101EMK868-915.

4.2.3 Building the Target Project

1. Select a target processor: MSP430F5529 or MSP430G2553.
2. Select a communication interface: I²C, UART, or SPI with CC110x.
3. Open CCS and select or create a workspace.
4. Import the MSPBoot CCS projects into the workspace. The projects are located in `MSPBoot\<target>_Examples\<communication_interface>`
 - a. Select the Copy projects into workspace checkbox to ensure you're working on the project located within your workspace instead of somewhere else on your PC.

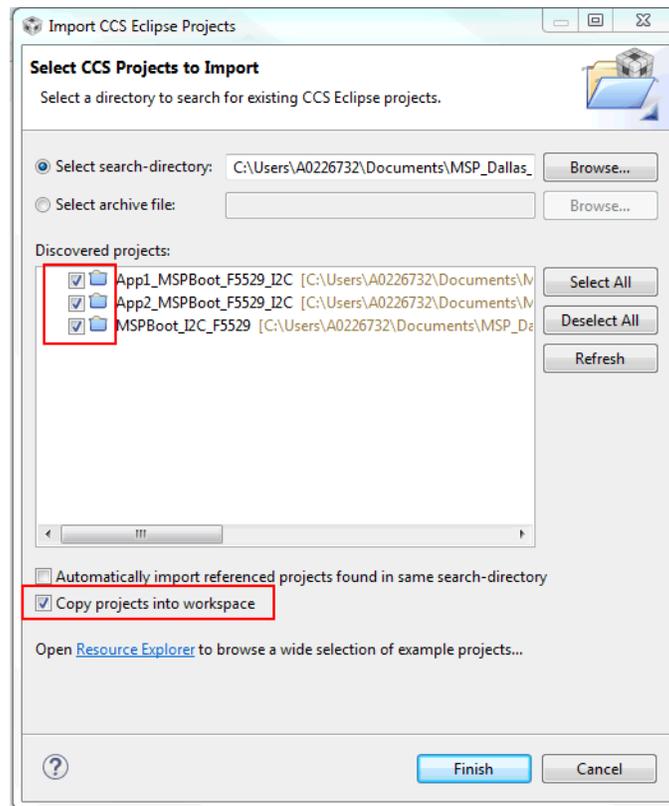


Figure 4-6. Import MSPBoot CCS Projects

5. Build the bootloader
 - a. Select the MSPBoot project
 - b. Select the proper target configuration (single image or dual image)

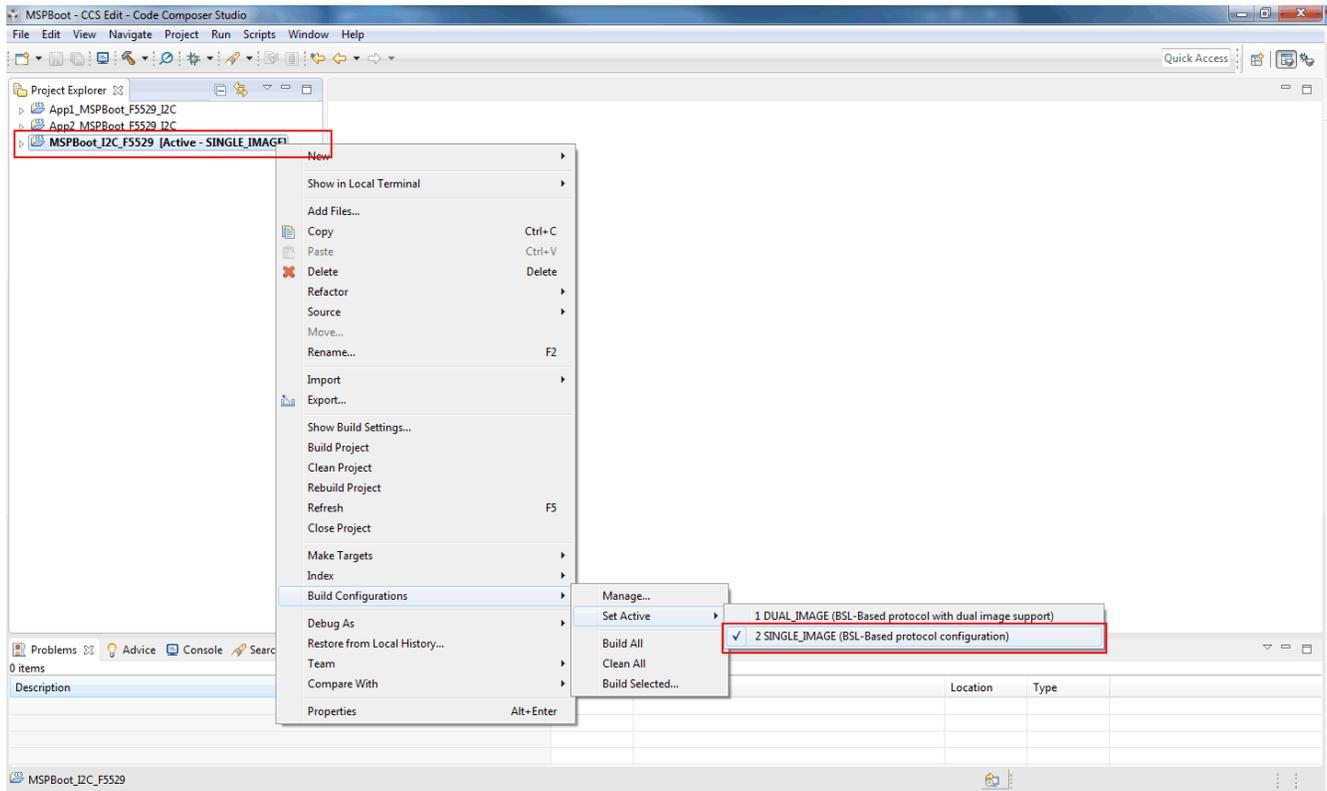


Figure 4-7. Select Target Configuration

6. Build  and Download . Only the target LaunchPad development kit should be connected to the PC
7. Build both applications.
 - a. Select the App1_MSPBoot project and select the same configuration as the bootloader.

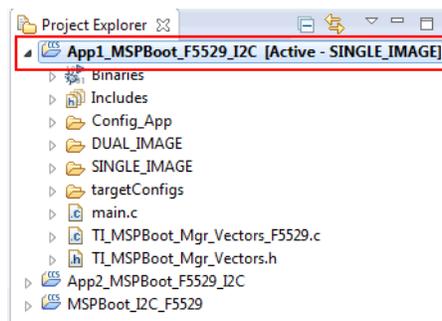


Figure 4-8. Select App1_MSPBoot Project

8. Click the Build  project. The output is generated after this step, but the output will be converted and downloaded through the host processor. [Section 4.1.2.1](#) explains how to convert the image and [Section 4.2.4](#) explains how to download it using a host demo.
9. Repeat Step 6 for App2_MSPBoot

4.2.4 Building the Host Project

The host project can be built following the next steps:

1. Import the project into CCS. The project files are located in MSPBoot\Host_Examples\Host_Examples_for_<target>_target\<communication interface>\.
2. Find the TODO in main.c located above the definition of several addresses including the CRC address
 - a. Update the values to match the addresses defined in the target device linker command file.
3. Add the target application C file generated from step 8 of [Section 4.2.3](#) to the excluded TargetApps Folder in the host project
4. Find the TODO in main.c located above the definition of the target application files
 - a. Update the name of these files to match the target application C file referenced in step 3.
5. Build the host project
 - a. Select the host project
 - b. Select the proper target configuration (single or dual image)

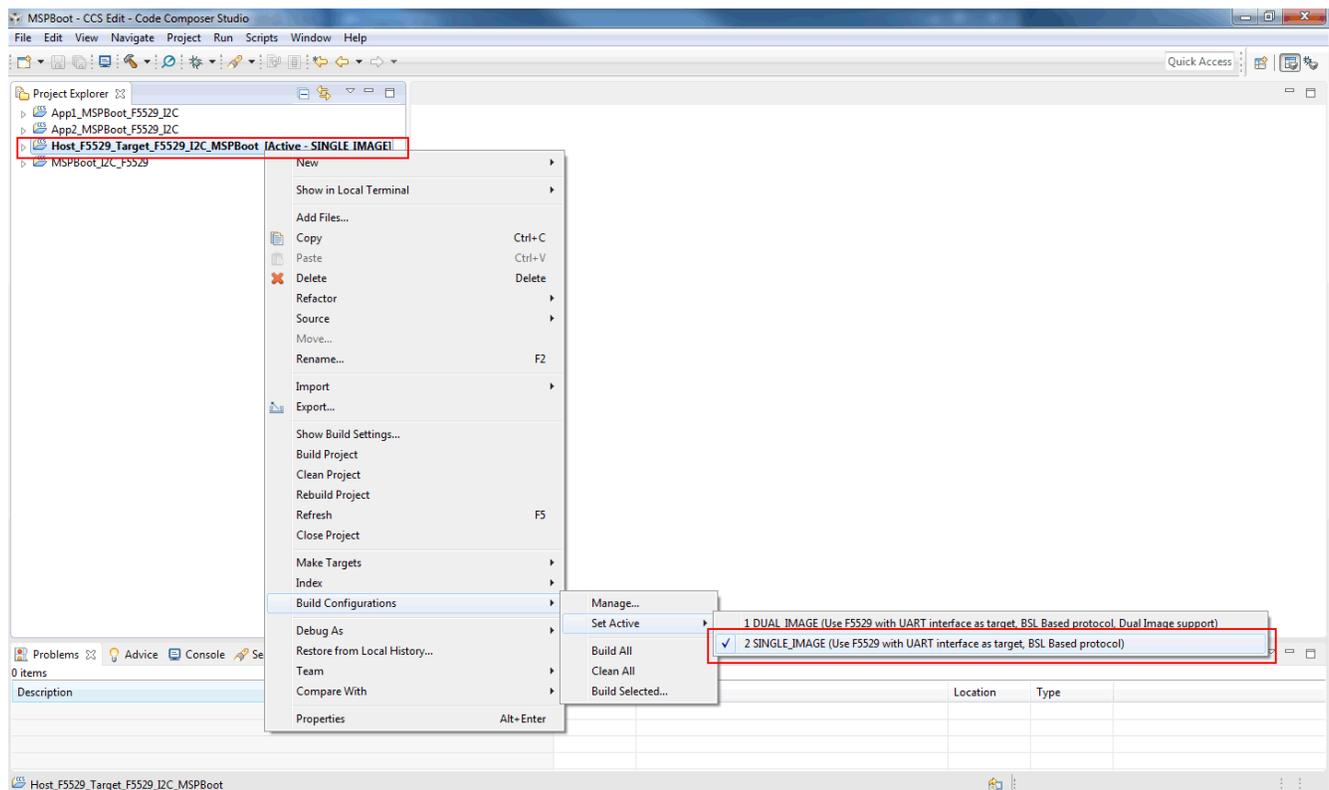


Figure 4-9. Target Selection for Host Project in CCS

6. Build  and Download . Only the host LaunchPad development kit should be connected to the PC.

The project uses the application images located in the following folder:

<Project_Dir>\Target_Apps

Where Project_Dir is the directory where the host project is located. This folder should also be excluded from the host build by default. Prebuilt images are included, but target applications can be replaced or updated by following the procedure described in [Section 4.2.3](#) and [Section 4.1.2.1](#).

4.2.5 Running the Examples

The host project sends two different images to the target device, using a push button for user interaction. USB connection to a computer is not required on either LaunchPad development kit to run the demo; however, each kit should be powered either by a USB connection through the eZ-FET or with steady 3.3-V external power

supply to V_{CC} and GND pins (ensure that the eZ-FET is disconnected in this instance). The demo is run using these steps regardless of communication type or image model used:

1. Build and download the MSPBoot as described in [Section 4.2.3](#) and build App1 and App2.
2. Convert App1 and App2 according to [Section 4.1.2.1](#).
3. Build and download the host application as described in [Section 4.2.4](#).
4. Connect the boards according to the desired communication type (I²C, UART, or SPI with CC110x).
 - a. Each host project contains a commented diagram at the start of code describing the proper connections.
5. Reset and execute code in both devices.
6. To enter the target bootloader mode (indicated by both LED1 and LED2 remaining on):
 - a. If the target does not have a valid application (default), the target stays in bootloader mode.
 - b. Bootloader mode can be forced in hardware by pressing and holding the S2 button on the target device while pressing and releasing the reset button.
 - c. If running an application:
 - i. APP1 jumps to bootloader mode when the S2 button is pressed on the target device.
 - ii. APP2 jumps to bootloader mode when it receives the Force Boot command (supported only if CI PHY-DL is shared).
7. Press the S2 button on the host board. The host device performs the following sequence of commands:
 - a. Toggles LED1 twice.
 - b. Sends “Force Boot” command (0xAA).
 - i. If the target device is already in bootloader mode, it discards the packet, because the CRC is incorrect.
 - ii. If the target is running APP2, the target device enters bootloader mode.
 - c. Requests the bootloader version (sends the TX_VERSION command).
 - i. If the target response is 0xA1 (expected from BSL protocol), the host continues.
 - ii. If the target response is any other value, the host aborts transaction.
 - d. Erases the target application area (sends the ERASE_APP command).
 - e. Sends APP1 (uses the RX_DATA_BLOCK commands).
 - f. Programs CRC of APP1 (uses the RX_DATA_BLOCK command).
 - g. Forces the target application to run (sends the JUMP2APP command).
 - h. Toggles LED1 twice to indicate successful transfer, and keeps LED1 on to show that the host is now ready to send APP2.
8. Target starts running APP1 upon completion of transfer.
 - a. The target device blinks LED1.
 - b. LED1 blinks at a periodic interval using the timer.
 - c. Press the S2 button on the target board to enter bootloader mode.
9. With the target in bootloader mode, press the S2 button on the host board to send APP2. When finished and done toggling, LED1 of the host board stays off to indicate that APP1 is now ready to be sent.

10. Target starts running APP2 upon completion of transfer.
 - a. The target device blinks LED2.
 - b. Press the S2 button on the target board to toggle LED2.
 - c. Because the CI is initialized, the host can send a Force Boot command to force bootloader mode in the target device at the start of a new transfer sequence.
11. Press the S2 button on the host to start a new sequence sending APP1 again.

Dual-image mode contains a brief pause from the host after the transfer is complete while it validates the download area, transfers the memory into the application space, and erases the download area after the application area is validated by a CRC-CCITT check.

5 References

1. [MSP430x2xx Family User's Guide](#)
2. [MSP430x5xx and MSP430x6xx Family User's Guide](#)
3. I²C specification 2.1
4. [MSP430™ Flash Device Bootloader \(BSL\) User's Guide](#)
5. [Creating a Custom Flash-Based Bootloader \(BSL\)](#)

6 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision D (February 2018) to Revision E (January 2024)	Page
• Updated the numbering format for tables, figures, and cross references throughout the document.....	1
• Changed the link to download the software package.....	1

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to [TI's Terms of Sale](#) or other applicable terms available either on [ti.com](https://www.ti.com) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2024, Texas Instruments Incorporated