

MSPBoot – Main Memory Bootloader for MSP430™ Microcontrollers

Luis Reynoso

MSP430 Apps

ABSTRACT

This application report describes the implementation of a bootloader that resides in the main memory in an MSP430™ microcontroller and that uses Inter-Integrated Circuit (I²C) communication but supports different communication interfaces. While highly flexible and modular, this bootloader has a small footprint, which makes it a very cost-effective solution.

A software package that includes examples and source code for both host and target devices can be downloaded from the following URL:

http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSPBoot/latest/index_FDS.html

For a step-by-step procedure that explains how to run the examples, see [Section 5](#) and [Section 6](#).

This bootloader is not to be confused with MSP430 Bootloader (BSL), which resides in protected memory (ROM or Flash) in some MSP430 microcontrollers. For more information on the MSP430 BSL, see [MSP430 Programming With the Bootloader \(BSL\)](#) 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 16KB) as well as dual image and interrupt redirection options, making it a good customizable alternative to the built-in BSL on MSP430 FRAM MCUs.

Contents

1	Introduction	3
2	Implementation	5
3	Customization of MSPBoot	23
4	Building MSPBoot	26
5	Demo Using MSP-EXP430F5438 as Host	36
6	Demo Using MSP-EXP430G2 as Host	41
7	References	43

List of Figures

1	MSPBoot Software Architecture	5
2	Flow Diagram of Main	6
3	Application Validation by AppManager	7
4	Memory Assignment	9
5	Vector Redirection Implementation	9
6	Dual Image Application Validation	11
7	I²C 7-Bit Addressing Format	13
8	UART 8-N-1 Format	14
9	SPI Format	14
10	Target Boards: MSP-EXP430G2 and MSP-EXPFR5739	26

11	Select MSPBoot Project	29
12	Select Target Configuration	30
13	Select App1_MSPBoot Project	30
14	Import MSPBoot CCS Projects	31
15	Select Target Configuration	32
16	Select App1_MSPBoot Project	33
17	CRCGen430 Example	34
18	430txt2C Example	34
19	Example Command	36
20	MSP-EXP430F5438	37
21	Target Selection for Host Project in IAR	38
22	Target Selection for Host Project in CCS	38
23	Target Selection for Host Project in IAR	41
24	Target Selection for Host Project in CCS	42

List of Tables

1	PHY-DL Callback Structure	13
2	Boot2App_Vector_Table Definition	14
3	Simple Protocol Format	15
4	Simple Protocol Slave Response	15
5	BSL-Based Protocol Command Format	17
6	BSL-Based Protocol Commands	17
7	BSL-Based Protocol Slave Response	18
8	SMBus-Based Protocol Command Format	21
9	SMBus-Based Protocol Commands	21
10	SMBus-Based Protocol Slave Responses	22
11	Optional Configurations in TI_MSPBoot_Config.h	23
12	Customization Files	24
13	Configuration Files	25
14	Predefined MSPBoot Configurations	25
15	GPIOs Used by Target Devices	27
16	Hardware Connection Between Host and Slave With MSP-EXP430F5438 as Host	37
17	Button Navigation in MSP-EXP430F5438	39
18	Host Demo Commands	39
19	Hardware Connection Between Host and Slave With MSP-EXP430G2 as Host	41

Trademarks

MSP430 is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

1 Introduction

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.

Many applications, however, require not only a solution that allows easy implementation but that also allows field upgrades.

MSP430 devices 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 Programming With the Bootloader \(BSL\)](#) 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 and MSP430FR5xx) have a ROM-resident BSL that supports only UART and that 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 still allows for an easy implementation of the application.

This application report describes the implementation of the I²C bootloader named MSPBoot with the following characteristics:

- Small footprint (1 to 3 flash sectors)
- Supports USI, USCI, and eUSCI peripherals
- Optional support for SMBus (protocol and clock time-out)
- Different communication protocols vary in complexity and size
- Different options allow for different levels of robustness
- Optional dual-image support
- Allows for use of all interrupts in application
- Application can optionally re-use the low-level I²C drivers from the bootloader or implement its own drivers
- Configurable entry sequence
- Optional validation of application using CRC-8 or CRC-CCITT
- Source code available, allowing for additional customizations
- MSPBoot version 1.2 includes examples of Universal Asynchronous Receiver/Transmitter (UART) and Serial Peripheral Interface Bus (SPI) Communication Interfaces using the Universal Serial Communication Interface (USCI) peripheral

Source code for the bootloader with different sample configurations, application examples, and host examples are included to allow for an easy testing, customization, and implementation.

Knowledge of I²C and SMBus specifications is assumed. For more information, see:

- I²C specification 2.1 <http://www.nxp.com/documents/other/39340011.pdf>
- SMBus Specification 2.0 <http://smbus.org/specs/smbus20.pdf>



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 16KB) as well as dual image and interrupt redirection options, making it a good customizable alternative to the built-in BSL on MSP430 FRAM MCUs.

1.1 Glossary



BOR	Brownout reset
BSL	MSP430 Bootloader
CI	MSPBoot Communication Interface
CRC	Cyclic Redundancy Check
eUSCI	Enhanced Universal Serial Communication Interface (MSP430FR573x)
I ² C	Inter-Integrated Circuit
MCU	Microcontroller
MI	MSPBoot Memory Interface
MSPBoot	The bootloader described by this application report
OSI	Open Systems Interconnection model
PEC	SMBus Packet Error Checking
PUC	Power-Up Clear Reset
ROM	Read-Only Memory
SMBus	System Management Bus
SPI	Serial Peripheral interface Bus
UART	Universal Asynchronous Receiver/Transmitter
USCI	Universal Serial Communication Interface (MSP430G2xx3)
USI	Universal Serial Interface (MSP430G2xx2)

1.2 Conventions



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

	S: Start	
	P: Stop	
	Sr: Repeated Start	
	Master to Slave	A: Acknowledge (SDA low)
	Slave to Master	/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 1](#) shows the software layers.

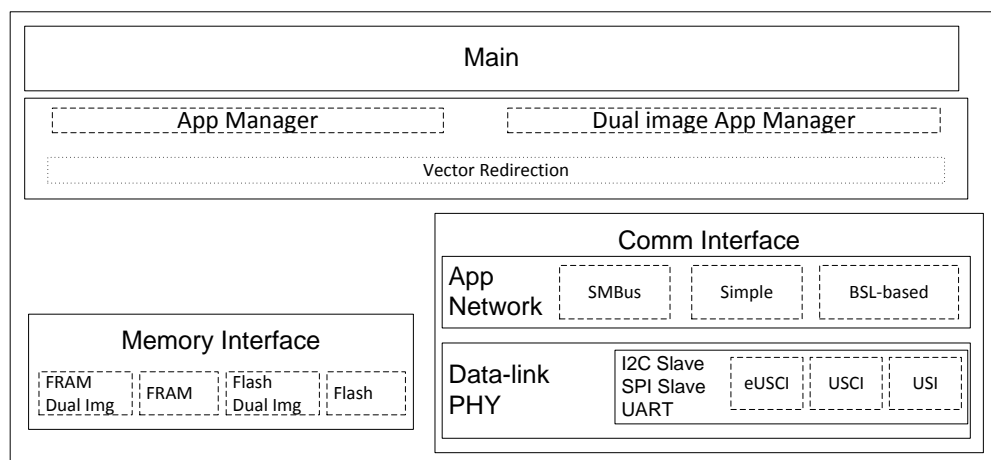


Figure 1. MSPBoot Software Architecture

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 microcontroller.
- Initialize the other MSPBoot layers.
- Implement the main loop that polls the communication interface and processes commands.

Figure 2 shows the state diagram of the main routine:

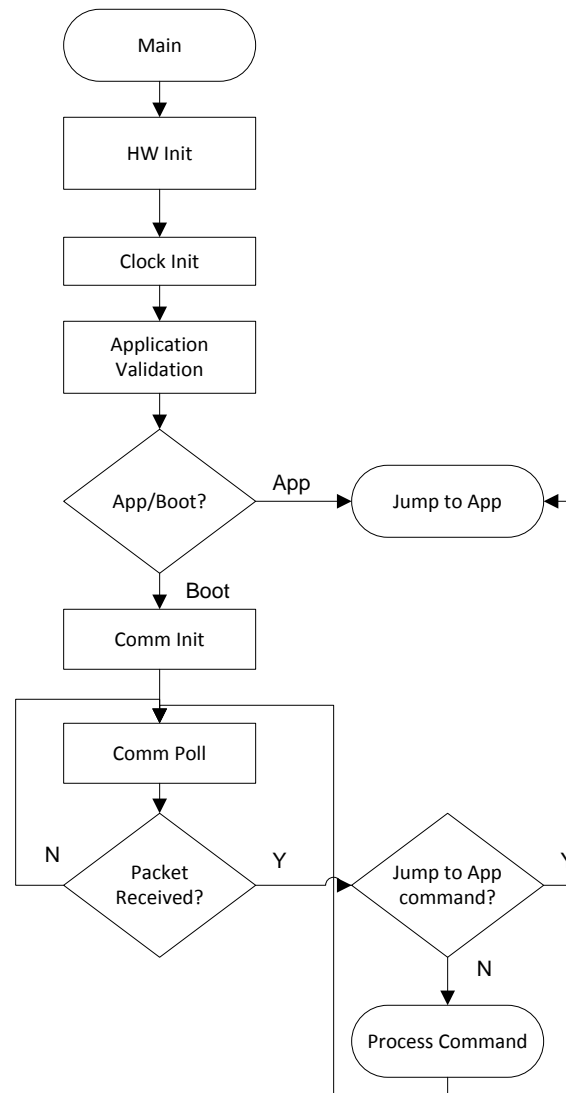


Figure 2. Flow Diagram of Main

2.2 Application Manager

The application manager has the main purposes of:

- Detecting when the device should be in bootloader mode or application mode
- Providing a mechanism to validate application
- Providing a mechanism to redirect interrupt vectors
- Providing a mechanism to jump 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 following the next rules:

- The 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](#)).
- The bootloader is executed if:
 - It is forced by an external event or by the application
 - OR
 - The application is invalid

[Figure 3](#) shows this decision process.

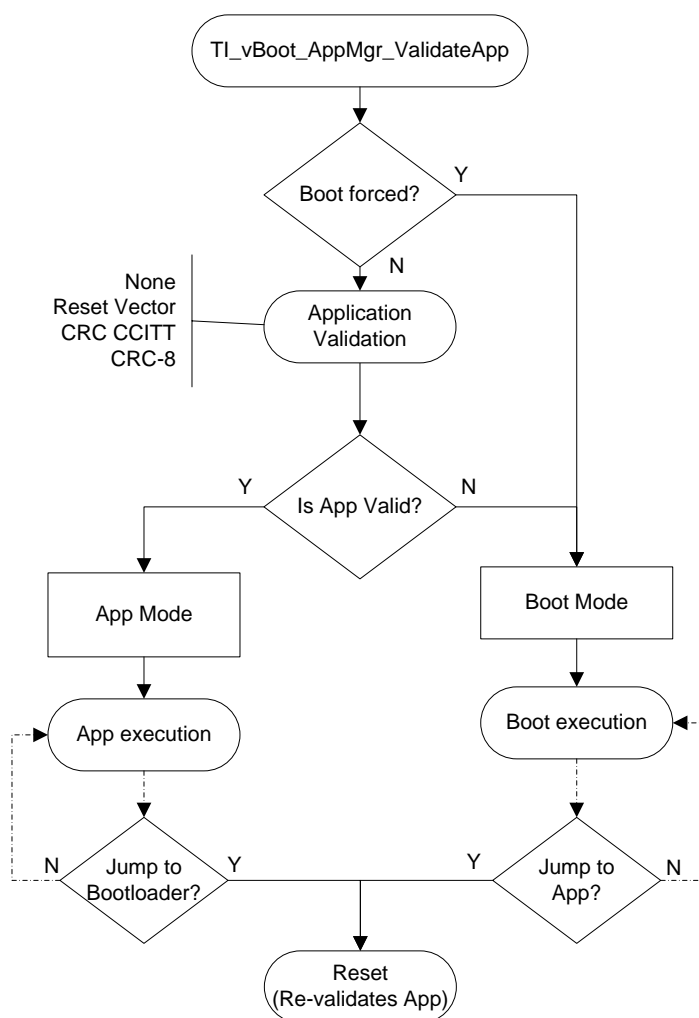


Figure 3. Application Validation by AppManager

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)
- P4.1 in MSP430FR5739 (S2 button in MSP-EXP430FR5739)

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. Four 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 BSL-based protocol (see [Section 2.4.2.2](#)) uses CRC-CCITT, so this validation method is recommended when using the BSL-based protocol.
- CRC_CRC8: A CRC-8 is calculated for the whole application image and compared to an expected value. Note that SMBus-based protocol (see [Section 2.4.2.3](#)) uses CRC-8 for PEC, so this validation method is recommended when using the SMBus-based protocol.

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 user responsibility. 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 update is complete and the device should jump to application.

Devices that support software BOR (for example, MSP430FR57xx) 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 microcontroller can be recovered by forcing bootloader mode.

The reset vector is an integral part of the bootloader, because it forces the microcontroller to always jump to the bootloader entry sequence, and thus it should not be erased. Because the reset vector resides in the top of Flash (`0xFFFFE`), the bootloader code is placed in the contiguous locations as shown in [Figure 4](#).

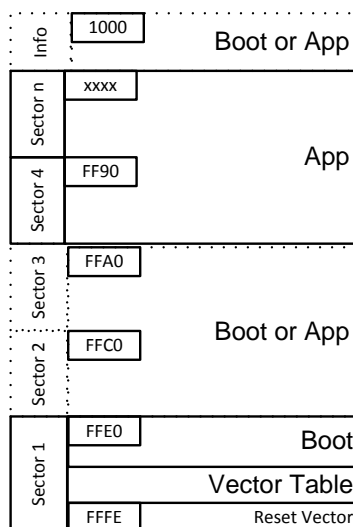


Figure 4. Memory Assignment

Note that the interrupt vector table resides in protected Boot area, too. Because the value of the interrupt tables is expected to change based on the application, this means that special considerations must be followed to allow for application interrupts.

2.2.3 Interrupt vectors in Flash devices

The minimum size for a Flash erase is a segment, which is 512 bytes in MSP430 microcontrollers. 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 5 shows the concept behind this implementation:

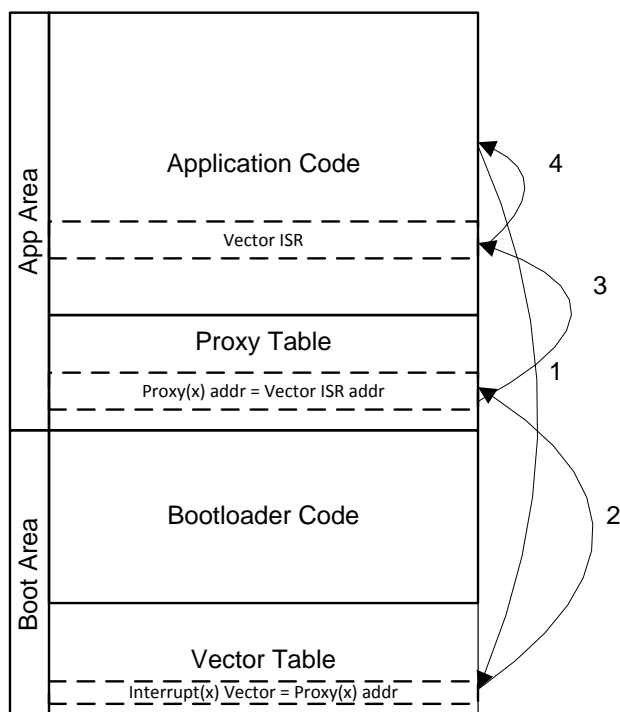


Figure 5. Vector Redirection Implementation

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 microcontrollers support redirecting vectors to RAM in hardware (SYSRIVECT), which could be a good alternative for devices with sufficient RAM.

2.2.4 Interrupt Vectors in FRAM Devices

FRAM does not have the limitation of a minimum erase size, so all interrupts can be reprogrammed in FRAM devices without risking erasure of the reset vector. By default, MSPBoot enables protection of the bootloader area using MPU, but this feature is disabled while reprogramming interrupt vectors.

Note that some MSP430 devices support redirecting vectors to RAM in hardware (SYSRIVECT), which could be a good alternative for devices with sufficient RAM. This also allows for full protection of the bootloader using the MPU module.

2.2.5 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.

In dual image mode, the main memory is divided into a *Download* area and an *Application* area, as described in [Section 2.3.1](#).

The application validation process in this mode is different from the usual procedure, as shown in Figure 6.

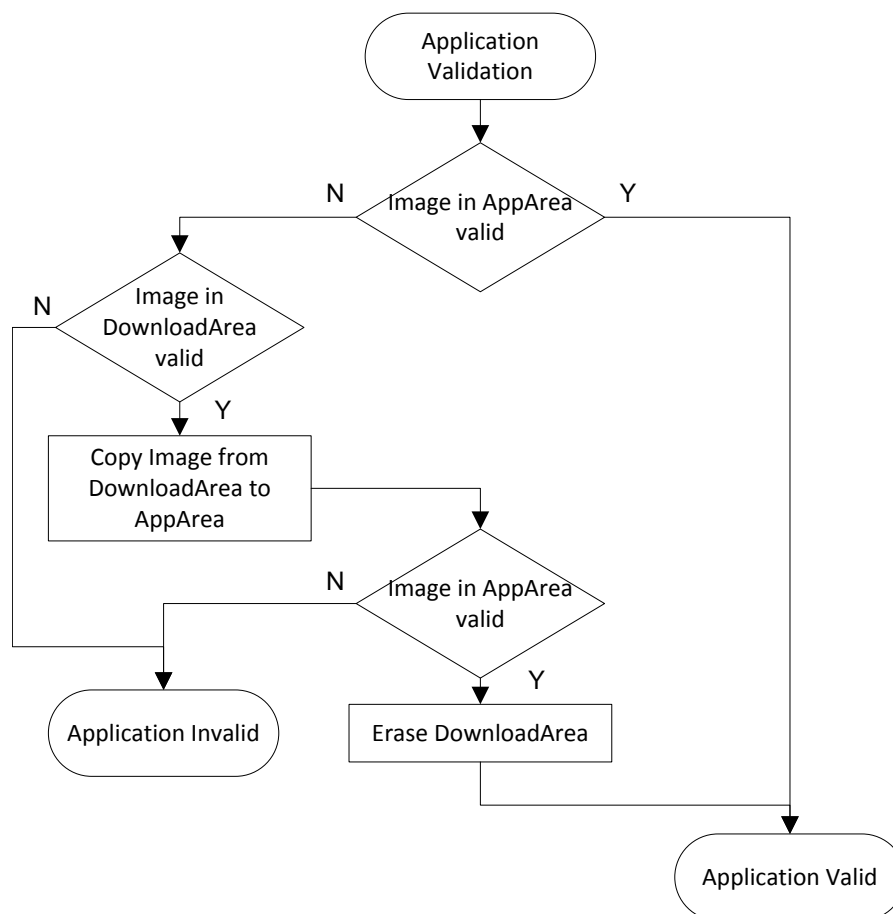


Figure 6. Dual Image Application Validation

2.2.5.1 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 by 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)

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: Non-writable 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 CCS and IAR.

The memory interface provides an API which is used to program and erase the application memory area and protect the bootloader area. This memory protection is implemented as follows:

- 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.
- FRAM devices
 - FRAM does not require erasing, but the application memory is written with 0xFF when an erase is performed to calculate a valid CRC.
 - The address being erased or programmed is validated to avoid accidental corruption of the bootloader area.
 - The MPU protects the bootloader area. The user can modify MPU settings according to the application, but it is recommended to always protect the bootloader area.
 - MPU protection is disabled only when updating interrupt vectors as described in [Section 2.2.4](#).

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. The bootloader area is hardware-protected using the MPU, if it is available.

2.3.1 Dual Image Support

When Dual Image support is enabled, the Memory Interface module partitions the 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 2.2.5](#).
- 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 and IAR.

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 PHY-DL

The Physical and Data-Link layer provides a Hardware Abstraction layer that facilitates the migration process to a different MSP430 derivative or peripheral.

The PHY-DL has the purpose of providing a stable channel for sending data to and receiving raw data from the host.

The current bootloader was originally implemented using I²C and it currently supports the USI (on the G2xx2), USCI (on the G2xx3), and eUSCI (on the FR573x) modules. Version 1.2 adds examples for UART and SPI Slave using USCI and other options could be implemented, if desired.

The PHY-DL layer is initialized by providing a pointer to a structure with these callback functions:

Table 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. Note that 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 interface is implemented using 7-bit addressing with a predefined address for MSP430 MCUs:

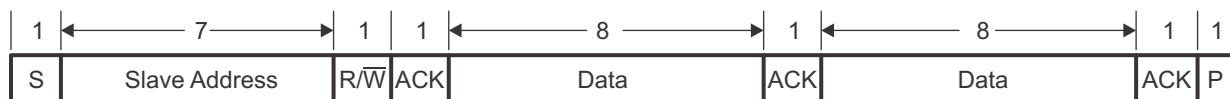


Figure 7. I²C 7-Bit Addressing Format

The predefined MSP430 MCU address is defined as:

```
CONFIG_CI_PHYDL_I2C_SLAVE_ADDR = 0x40
```

Files are included to support G2xx2 USI, G2xx3 USCI, and FR573x eUSCI.

2.4.1.1.1 Timeout 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, it can reset its interface. This is a requirement for SMBus devices.

This feature is enabled depending on CONFIG_CI_PHYDL_TIMEOUT. USI and USCI 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) as shown in Figure 8.

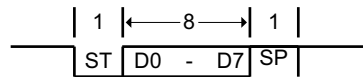


Figure 8. UART 8-N-1 Format

The default baudrate is defined as:

CONFIG_CI_PHYDL_UART_BAUDRATE = 9600

Files are included to support G2xx3 USCI.

2.4.1.3 SPI

The SPI interface is implemented using the following configuration:

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

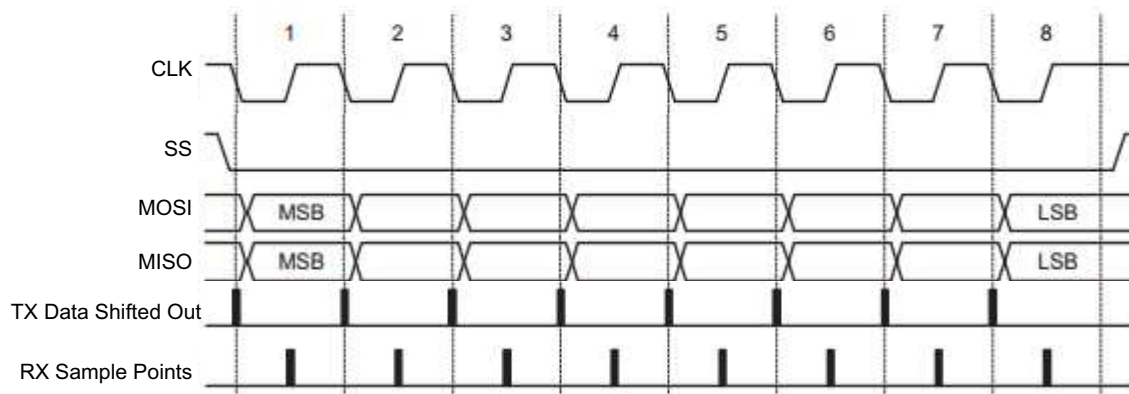


Figure 9. SPI Format

Files are included to support G2xx3 USCI.

2.4.1.4 Comm Sharing

The user application can use the communication interface as desired (for example, as I²C, GPIO, or other purpose), because the resources are released when the microcontroller 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 in Table 2.

Table 2. Boot2App_Vector_Table Definition

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

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

The application must declare its own callbacks that are passed during initialization of CI PHY-DL and called when the corresponding event is detected.

Note that the PHY-DL layer was designed with a small footprint being a top priority. An application can implement its own drivers if the PHY-DL implementation is inadequate.

Application examples that show how to share CI PHY-DL are included in the software that is included with this application report.

2.4.2 NWK-APP

The CI Network-Application layer implements the communication protocol, interprets the raw data from PHY-DL, validates the data, and executes the appropriate commands.

To provide flexibility and allowing for different application requirements, MSPBoot supports different communication protocols that provide different levels of robustness and footprint sizes:

- Simple Protocol
- BSL-based protocol
- SMBus-based protocol

The bootloader must make use of one and only one of these protocols.

2.4.2.1 Simple Protocol

The Simple Protocol has a small footprint and relatively low level of robustness, but it still provides some inherent security.

The protocol is *byte based* and uses the format shown in [Table 3](#).

Table 3. Simple Protocol Format

SYNC	BYTE0	BYTE1	BYTE2	...	BYTE(n-1)	BYTE(n)
0x55	App[0]	App[1]	App[2]	...	App[n-1]	App[n]

The response from the microcontroller is hard-coded as shown in [Table 4](#).

Table 4. Simple Protocol Slave Response

RESPONSE
0xB0

Data can be sent in packets of any size, because the target device process each byte as it is received.

Note that the target device expects a full image from application byte 0 to byte *n*, and it is the host's responsibility to meet this requirement.

The following steps are implemented for the Simple Protocol:

1. The target initializes the Communication and waits for an incoming reception.
2. Upon reception of a byte:
 - (a) If a received byte is different from SYNC byte, it is ignored.
 - (b) If the received byte matches the SYNC byte, the application area is erased and an address pointer is set to the hardcoded start of Application area.
3. The target now waits for the application image.
4. Upon reception of a byte, data is written to the location pointed by the address pointer and the address pointer is incremented.
 - (a) If the address pointer points to valid application area, the target waits for the next byte.
 - (b) If the address pointer is outside the application area, it means that the application image has been completely received and the target jumps to the application.

2.4.2.1.1 Security

The data contents for this protocol are not validated with a CRC or any other method, so the protocol assumes:

- Content received from the host through the peripheral interface is valid.
- Application validation method verifies the application image upon completion.

This protocol erases and writes the application area automatically, because it is hard-coded in the implementation, and thus it cannot erase the bootloader section. Therefore, it does not require the flash address validation that is provided by CONFIG_MI_MEMORY_RANGE_CHECK.

If the download process is interrupted, the application area can be corrupted, so it is recommended to use one of the application validation methods described in [Section 2.2.1.2](#).

2.4.2.1.2 Examples Using I²C

The following considerations apply when using I²C with Simple 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 holds the clock low (known in I²C as "stretching the clock").
- Note that different commands have different processing times.

Examples:

- Host reads response from microcontroller (that is, determines if the microcontroller is in bootloader mode)

S	0x40	R	A	0xB0	/A	P
Addr		Version				

- Sync byte sent by host.

S	0x40	W	A	0x55	A	P
Addr		SYNC				

The microcontroller erases the application area and prepares to receive the binary image.

- Application image sent by host.

S	0x40	W	A	Data ₀	A	Data ₁	A	...	Data _n	A	P
Addr											

Data₀ is written to the start of the application area, followed by Data₁ and so on. At the end of Data_n, the microcontroller automatically resets and starts the application validation process.

2.4.2.2 BSL-Based Protocol

The MSP430 BSL is the standard bootloader that is included in MSP430 microcontrollers. It is described in detail in [MSP430 Programming With the Bootloader \(BSL\)](#).

The BSL-based protocol that is implemented in MSPBoot shares some of the same 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 shown in [Table 5](#).

Table 5. BSL-Based Protocol Command Format

Header	Length	Payload (see Table 6)	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 19 by default (1 CMD + 2 Addr + 16 Data)

Header: Fixed to 0x80

Length: One byte that reports 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 optional data.

Checksum: 16-bit CRC CCITT of the payload

The commands shown in [Table 6](#) are implemented as payload.

Table 6. BSL-Based Protocol Commands

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

ERASE_SEGMENT

Erases the memory segment (512 bytes in flash) starting at address 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).

The response from the target is always a single byte. Valid values are shown in [Table 7](#).

Table 7. 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 255	Sent as response for TX_VERSION command (default is 0xA0)

2.4.2.2.1 Security

The contents of each packet are validated with a 16-bit CRC, which 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, it is recommended to include the CONFIG_MI_MEMORY_RANGE_CHECK MI definition, which validates the address before a program or erase operation.

If the process is interrupted, the application area can be corrupted, so it is recommended to use one of the application validation methods described in [Section 2.2.1.2](#) or to use the dual-image approach.

2.4.2.2.2 BSL-Based Protocol Using SPI

The SPI implementation of this protocol follows the same guidelines used for I²C and UART, but it includes some important changes due to the full-duplex nature of this protocol.

An important difference is the use of a dummy byte to request the response from the slave device.

This dummy byte is defined as:

DUMMY_CHAR = 0x90

If a slave detects the BSL header '0x80' as the first byte of a packet, then it processes the packet as usual; but if it receives the dummy byte '0x90' instead, it ignores the contents of the packet and only sends the response from the previous packet. Any other values in the first byte of the packet is processed as incorrect by the BSL protocol.

This BSL also adds a new Slave Response, which indicates when the slave is busy receiving or processing new packets.

Response	Value	Description
RECEIVING_PACKET	0xC0	Packet reception is in progress (slave might take some time parsing and validating the contents)
PROCESSING_CMD	0xC1	Packet was received correctly by the slave device and the command is being executed (different commands have different processing times)

The following procedure should be followed by a host sending a packet:

1. Host sends packet following BSL-based protocol
2. Host sends dummy byte checking for successful reception
 - (a) If response is an error message, exit with error.
 - (b) If response is OK or MSPBOOT_VERSION (after TX_VERSION command), packet was received correctly.
 - (c) If response is PROCESSING_CMD or RECEIVING_PACKET, retry step 2 until a time-out expires.
 - (d) If a time-out expires, the packet was not processed by the slave correctly, exit with error.

2.4.2.2.3 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

Examples:

- Host reads version from the microcontroller

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		

- Host writes 16 bytes to address 0xC000.

S	0x40	W	A	0x80	A	0x13	A	0x10	A	0x00	A	0xC0	A	0x03	A	0xEE	A
	Addr			Header		Length		RX_DATA_BLOCK		AddrL		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	0x4E	A	0x6E	A	P
Data11		Data12		Data13		Data14		Data15		Checksum_L		Checksum_H		

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

2.4.2.2.4 Examples Using UART

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.
- Note that different commands have different processing times.

Examples:

- Host erases the microcontroller 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

2.4.2.2.5 Examples Using SPI

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

- The master always starts a transfer by holding SS low and it must hold the line low for the duration of the packet.
- Address is not required since the slave is only accessed when SS is low.
- SPI is full-duplex and the target will always respond with data, but this data should be ignored until the response from the slave is ready.
- This protocol modifies the standard BSL-protocol slightly by adding a dummy byte and a new Slave response as explained in [Section 2.4.2.2.2](#).
- The master should poll the device with the dummy byte until it's ready to respond.
- Note that different commands have different processing times.

Examples:

- Host erases the microcontroller application area

MOSI	0x80	0x13	0x10	0x00	0xC0
	Header	Length	RX_DATA_BLOCK	AddrL	AddrH
MISO	X	X	X	X	X
CR					

Note that the slave response is "don't care" during the duration of the packet, but if an error is detected during the packet transfer (HEADER_ERROR), the error will be sent immediately and this response will remain on the bus until the next BSL packet is received.

The slave device sends the RECEIVING_PACKET while receiving a packet and even after a successful reception, but while still busy validating its contents (calculating CRC); then it sends the PROCESSING_CMD if the contents are valid and while the command is being executed; and it sends the result of the command after executing it.

The master should poll the slave to check the status of the command as described in [Section 2.4.2.2.2](#).

MOSI	0x80		0x90		0x90
	Dummy		Dummy		Dummy
MISO	0xC0		0xC1		0x00
CR	RECEIVING PACKET		PROCESSING CMD		OK
CS					

2.4.2.3 SMBus-Based Protocol

SMBus (System Management Bus) is an interface that adds a network layer in its specification, in addition to defining physical and data-link layers based on I²C. This network layer defines a mechanism to detect packet errors and predefined bus protocols.

For more information regarding the SMBus specification, see <http://smbus.org/specs/smbus20.pdf>.

The SMBus-based protocol implemented in MSPBoot provides a similar level of security compared to the BSL-based and adds time-out detection.

Similar to the BSL-based protocol implementation, the SMBus-based protocol implements a basic set of commands to minimize its footprint.

The protocol is *packet based* and uses the format shown in [Table 8](#).

Table 8. SMBus-Based Protocol Command Format

Command	Byte Count	Data ⁽¹⁾	PEC
1 byte	1 to 32	1 to 32 bytes	1 byte

⁽¹⁾ SMBus defines a maximum length of 32 bytes for the Block Write protocol.

Command: One byte that defines the command to execute. Bit 7 defines if PEC is enabled for the command.

Byte Count: Optional byte used in Block Write protocol to define the number of bytes of Data.

Data: Optional bytes used in Block Write (1 to 32 bytes) or Word Write (2 bytes) protocols.

PEC: One byte with Packet Error Checking (CRC-8) of the packet.

The commands shown in [Table 9](#) are implemented in this protocol.

Table 9. SMBus-Based Protocol Commands

Command	CMD	CMD With PEC	SMBus Protocol	Byte Count	Data
TX_VERSION	0x01	0x81	Send Byte	–	–
ERASE_APP	0x02	0x82	Send Byte	–	–
ERASE_SEGMENT	0x03	0x83	Write Word	–	AddrL-AddrH
RX_DATA_BLOCK	0x04	0x84	Block Write	1 to 32	AddrL-AddrH-Data ₀ -Data _n
JUMP2APP	0x05	0x85	Send Byte	–	–

TX_VERSION

Requests the MSPBoot version from the target.

ERASE_APP

Erases the Application area.

ERASE_SEGMENT

Erases the memory segment (512 bytes in flash) starting at address ADDR.

RX_DATA_BLOCK

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

JUMP2APP

Instructs the target to jump to the application image (after validation).

The response from the target is always a single byte that uses the SMBus receive byte protocol without PEC. Valid values are shown in [Table 10](#).

Table 10. SMBus-Based Protocol Slave Responses

Response	Value	Description
OK	0x00	Previous command executed correctly
NTR	0x50	Nothing to respond (previous response was sent)
PEC_ERROR	0x52	Packet Checksum Error
PACKETSIZE_ERROR	0x53	Packet size is invalid
UNKNOWN_ERROR	0x54	Error in Protocol (no command has been received)
INCORRECT_FORMAT	0x55	Incorrect format in protocol
INCORRECT_COMMAND	0x56	Invalid Command
MEM_ERROR	0x57	Error returned by MI
MSPBOOT_VERSION	0-255	Sent as response for TX_VERSION command (default is 0xC0)

2.4.2.3.1 Security

CRC-8 is implemented in SMBus, because the PEC mechanism differs from the 16-bit CRC that is implemented in the BSL-based protocol. CRC-8 provides sufficient robustness to the protocol.

SMBus defines a mechanism to detect bus time-out when one of the devices on the bus holds the line too long. This feature is implemented in the PHY-DL layer as explained in [Section 2.4.1.1.1](#). The time-out is reported to the SMBus NWK-APP layer, which forces a reset of the Communication Interface.

The ERASE_SEGMENT and RX_DATA_BLOCK commands can erase and write any area within the 16-bit memory map, and can potentially corrupt the bootloader. To avoid this possibility, it is recommended to include the CONFIG_MI_MEMORY_RANGE_CHECK MI definition, which validates the address before a program or erase operation.

If the write process is interrupted, the application area can be corrupted, so it is recommended to use one of the application validation methods described in [Section 2.2.1.2](#) or to use the dual-image approach.

2.4.2.3.2 Examples Of Using I²C

The following considerations apply when using I²C with SMBus:

- 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
- If the clock is stretched for too long, it will be recognized as a time-out and the slave will reset its interface.

Examples:

- Host reads version from microcontroller without PEC.

S	0x40	W	A	0x01	A	P
	Addr			TX_VERSION		

S	0x40	R	A	0xC0	/A	P
	Addr			Version		

- Host erases the application area with PEC.

S	0x40	W	A	0x82	A	0x31	A	P
Addr		ERASE_APP (with PEC)			PEC			

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

- Host writes 4 bytes to address 0xC000 with PEC.

S	0x40	W	A	0x84	A	0x06	A	0x00	A	0xC0	A	0x03	A	0xEE	A	0x47	A
Addr		RX_DATA_ BLOCK (with PEC)			Byte Count		Addr_L		Addr_H		Data0		Data1		Data2		

0xFF	A	0x4F	A	P
Data3		PEC		

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

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.

The options shown in [Table 11](#) can be configured in TI_MSPBoot_Config.h.

Table 11. Optional Configurations in TI_MSPBoot_Config.h

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 40 bytes
Undefined	Address being erased or programmed is not validated. Host must send correct address, or programming procedure must ensure that bootloader area is not modified (that is, Simple Protocol implementation).	–
CONFIG_APPMGR_APP_VALIDATE		
1	Application is validated by checking its reset vector.	Adds approximately 10 bytes
2	Application is validated by checking its CRC_CCITT.	Adds approximately 94 bytes
3	Application is validated by checking its CRC-8.	Adds approximately 90 bytes
CONFIG_CI_PHYDL_COMM_SHARED		
Defined	Communication Interface PHY-DL layer is shared with application.	Adds approximately 4 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 11. Optional Configurations in TI_MSPBoot_Config.h (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.	–

Other customizations are selected by adding and using the appropriate files in the project. [Table 12](#) shows the files that can be interchanged in the project.

Table 12. Customization Files

CI PHY-DL	
TI_MSPBoot_CI_PHYDL_eUSCI_I2C_slave.c	Use eUSCI as I ² C Slave (FR57xx)
TI_MSPBoot_CI_PHYDL_USCI_I2C_slave.c	Use USCI as I ² C Slave (G2xx3)
TI_MSPBoot_CI_PHYDL_USI_I2C_slave.c	Use USI as I ² C Slave (G2xx2)
TI_MSPBoot_CI_PHYDL_USCI_UART.c	Use USCI as UART (G2xx3)
TI_MSPBoot_CI_PHYDL_USCI_SPI.c	Use USCI as SPI Slave (G2xx3)
CI NWK-APP	
TI_MSPBoot_CI_NWK_APP_Simple.c	Use Simple Protocol
TI_MSPBoot_CI_NWK_APP_BSL.c	Use BSL-based protocol
TI_MSPBoot_CI_NWK_APP_SMBus.c	Use SMBus-based protocol
TI_MSPBoot_CI_NWK_APP_BSL_UART.c	Use BSL-based protocol with UART
TI_MSPBoot_CI_NWK_APP_BSL_SPI.c	Use BSL-based protocol with SPI
MI	
TI_MSPBoot_MI_Flash.c	API used to program application Flash
TI_MSPBoot_MI_FlashDualImg.c	API that implements dual-image in Flash
TI_MSPBoot_MI_FRAM.c	API that programs application FRAM
TI_MSPBoot_MI_FRAMDualImg.c	API that implements dual-image in FRAM
App Manager	
TI_MSPBoot_AppMgr.c	Standard App Manager
TI_MSPBoot_AppMgrDualImg.c	App Manager that supports dual image
Vector Redirection	
TI_MSPBoot_VecRed_G2x52.c	Supports vector redirection for G2x52
TI_MSPBoot_VecRed_G2x53.c	Supports vector redirection for G2x53

3.1 Predefined Customizations

The software package includes projects for IAR and CCS that supports three devices, three communication interfaces, and six predefined configurations.

Table 13 shows the different target devices and the files that are included for each one of them.

Table 13. Configuration Files

Device	HW_RESET_PUC, HW_RESET_BOR	HW_ENTRY_ CONDITION	TI_MSPBoot_ CI_PHYDL_XXX.c	TI_MSPBoot_ VecRed_XXX.c	TI_MSPBoot_ MI_XXX.c ⁽¹⁾
G2452_I2C	HW_RESET_PUC	Check P1.3	USI_I2C_slave	G2x52	Flash, FlashDualImg
G2553_I2C	HW_RESET_PUC	Check P1.3	USCI_I2C_slave	G2x53	Flash, FlashDualImg
FR5739_I2C	HW_RESET_BOR	Check P4.1	eUSCI_I2C_slave	N/A	FRAM, FRAMDualImg
G2553_UART	HW_RESET_PUC	Check P1.3	USCI_UART	G2x53	Flash
G2553_SPI	HW_RESET_PUC	Check P1.3	USCI_SPI	G2x53	Flash

⁽¹⁾ Dual Image files are used for BSLBased_DualImg target configuration.

Table 14 shows the available configurations and corresponding settings.

Table 14. Predefined MSPBoot Configurations ⁽¹⁾

Configuration	NDEBUG	CONFIG_MI_MEMORY_RANGE_CHECK	CONFIG_APPMGR_APP_VALIDATE	CONFIG_CI_PHYDL_COMM_SHARED	CONFIG_CI_PHYDL_TIMEOUT	CONFIG_CI_PHYDL_START_CALLBACK	CONFIG_CI_PHYDL_END_CALLBACK	CONFIG_CI_PHYDL_ERROR_CALLBACK	TI_MSPBoot_CI_NWK_APP_XXX.c	TI_MSPBoot_MI_XXX.c	TI_MSPBoot_AppMgr_XXX.c
Simple	X		1 ⁽²⁾						Simple	Flash or FRAM ⁽³⁾	–
BSLBased	X	X	2 ⁽²⁾	X		X			BSL	Flash or FRAM ⁽³⁾	–
SMBus	X	X	3 ⁽²⁾	X	X	X	X	X	SMBus	Flash or FRAM ⁽³⁾	–
BSLBased_DualImg	X	X	2 ⁽²⁾	X		X			BSL	Flash or FRAM ⁽³⁾	Dual Image
BSLBased_UART	X	X	2 ⁽²⁾	X					BSL_UART	Flash	
BSLBased_SPI	X	X	2 ⁽²⁾	X		X			BSL_SPI	Flash	

⁽¹⁾ UART and SPI configurations only implemented for G2xx3.

⁽²⁾ 1 = Reset vector, 2 = CRC-CCITT, 3 = CRC-8

⁽³⁾ Flash is used in G2xx. FRAM is used in FR57xx.

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 5](#) and [Section 6](#) describe how to build and use the host applications to run a demo.

4.1 Hardware

This software package includes examples for MSP430G2452, MSP430G2553, and MSP430FR5739. Any MSP430 board can be used, but the LaunchPad™ development kit (MSP-EXP430G2) and MSP-EXP430FR5739 Experimenter Board are the examples used in this application report.



Figure 10. Target Boards: MSP-EXP430G2 and MSP-EXPFR5739

The bootloader and demo applications use the pins shown in [Table 15](#).

Table 15. GPIOs Used by Target Devices

Pin	Target	
	MSP-EXP430G2	MSP-EXP430FR5739
I ² C: SCL	P1.6/UCB0SCL ⁽¹⁾	P1.7/UCB0SCL
I ² C: SDA	P1.7/UCB0SDA	P1.6/UCB0SDA
UART: RXD	P1.1/UCA0RXD ⁽²⁾	N/A
UART:TXD	P1.2/UCA0TXD ⁽²⁾	N/A
SPI: MISO	P1.1/UCA0SOMI	N/A
SPI:MOSI	P1.2/ UCA0SIMO ⁽²⁾	N/A
SPI:CLK	P1.4/UCA0CLK ⁽²⁾	N/A
SPI:SS	P1.5 ⁽²⁾	N/A
Demo: LED1	P1.0	PJ.0
Demo: LED2	P2.0 ⁽³⁾	PJ.1
Demo: Button S2	P1.3	P4.1

⁽¹⁾ Jumper J5 that connects P1.6 to LED2 must be removed in MSP-EXP430G2.

⁽²⁾ UART and SPI configuration only implemented for G2xx3.

⁽³⁾ P2.0 is not connected to LED by default in MSP-EXP430G2. An external connection can be added for demo purposes.

4.2 Software

The software package includes the following folders:

- **Target:** Target's bootloader and demo applications
 - **FR5739_I2C:** Projects that support the MSP430FR5739 using I²C
 - **CCS:** CCS project files
 - **MSPBoot:** CCS project files for bootloader
 - **Config:** CCS Linker files for bootloader
 - **App1_MSPBoot:** CCS project files for Application Example 1
 - **Config:** CCS Linker files for App1
 - **App2_MSPBoot:** CCS project files for Application Example 2
 - **Config:** CCS Linker files for App2
 - **IAR:** IAR project files (similar structure as CCS folder)
 - **Src:** Source code
 - **MSPBoot:** Source code for bootloader
 - **AppMgr:** App Manager source code files
 - **Comm:** CI source code files
 - **MI:** MI source code files
 - **App1:** Source code for Application Example 1
 - **App2:** Source code for Application Example 2
 - **G2452_I2C:** Projects that support MSP430G2452 using I²C (same structure as FR5739)
 - **G2553_I2C:** Projects that support MSP430G2553 using I²C (same structure as FR5739)
 - **G2553_SPI:** Projects supporting MSP430G2553 using SPI (same structure as FR5739)
 - **G2553_UART:** Projects supporting MSP430G2553 using UART (same structure as FR5739)
- **Host:** Host demo application
 - **MSP-EXP430F5438:** Project that supports the host using MSP-EXP430F5438. See [Section 5](#)
 - **CCS:** CCS project files
 - **IAR:** IAR project files
 - **Src:** Source code
 - **TargetApps:** ConvertedTargetApplication Examples
 - **MSP-EXP430G2:** Project that supports the host using MSP-EXP430G2 (same structure as MSP-EXP430F5438). See [Section 6](#).
- **430txt_converter:** Scripts and applications used to convert IAR or CCS output files to Host TargetApps. See [Section 4.2.2](#).
- **linkerGen:** Script used to automatically generate linker files for IAR/CCS, see [Section 4.2.3](#).

This software package was built and tested using CCS 5.5 and IAR 5.6. Other IDE versions and compilers may not directly support the resources as provided and could require slight modifications.

4.2.1 Building the Target Software

4.2.1.1 Building the Target Software in IAR

1. Select a target processor: MSP430G2452_I2C, MSP430G2553_I2C, or MSP430FR5739_I2C, MSP430G2553_UART or MSP430G2553_SPI.
2. Using IAR, open the workspace located in:
MSPBoot\Target\<target>\IAR\MSPBoot_Workspace_<target_device>.eww
3. Build the bootloader
 - (a) Select the MSPBoot project:

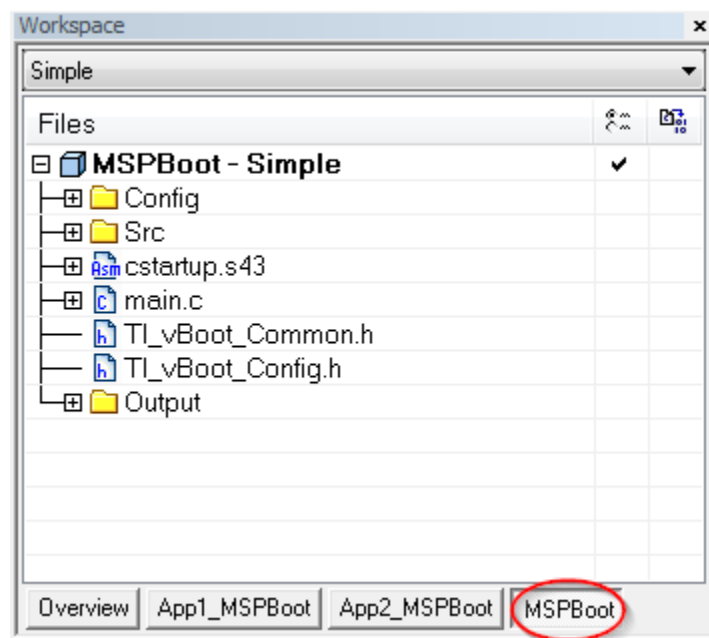


Figure 11. Select MSPBoot Project

- (b) Select a proper target configuration based on [Section 3.1](#).

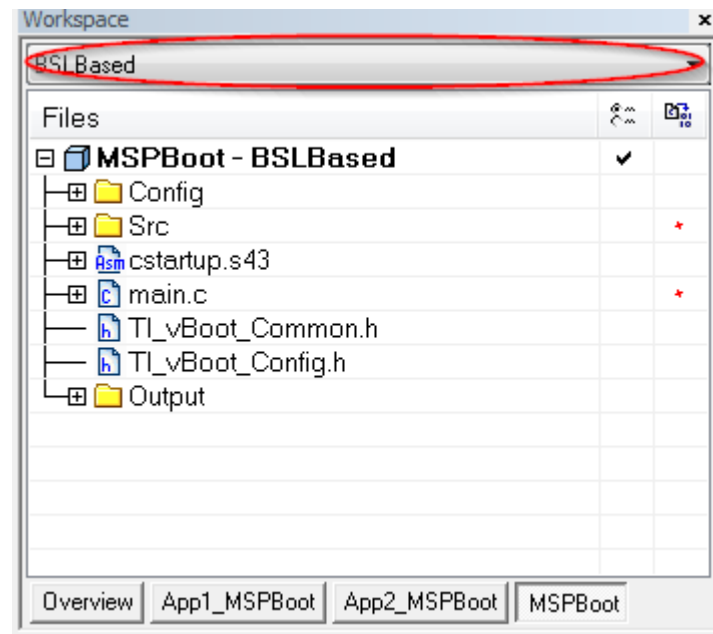




Figure 12. Select Target Configuration

- (c) Click the Build  and then Download .
4. Build the applications.
- (a) Select the App1_MSPBoot project and select the same configuration as the bootloader:

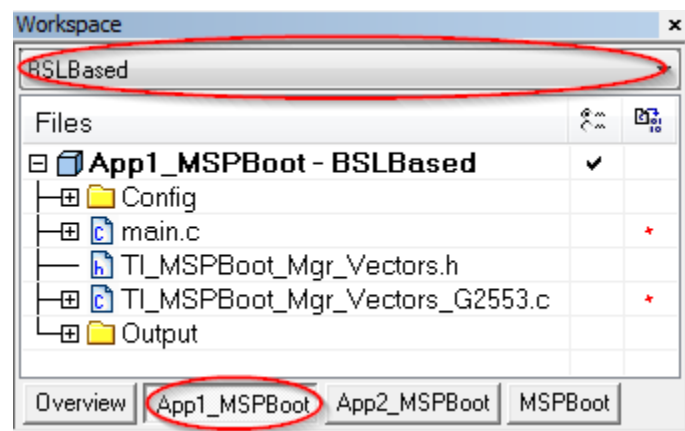



Figure 13. Select App1_MSPBoot Project

- (b) Click the Build  project. Note that the output is generated after this step, but the output needs to be converted and downloaded by the host processor. [Section 4.2.2](#) describes how to convert the image, and [Section 5](#) and [Section 6](#) explain how to download using a host demo.
- (c) Repeat step 4 for App2_MSPBoot.

4.2.1.2 Building the Target Software in CCS

1. Select a target processor: MSP430G245_I2C, MSP430G2553_I2C, MSP430FR5739_I2C, MSP430G2553_UART or MSP430G2553_SPI.
2. Open CCS and select or create a workspace.
3. Import the MSPBoot CCS projects into the workspace. The projects are located in MSPBoot\Target\<target>\CCS\.

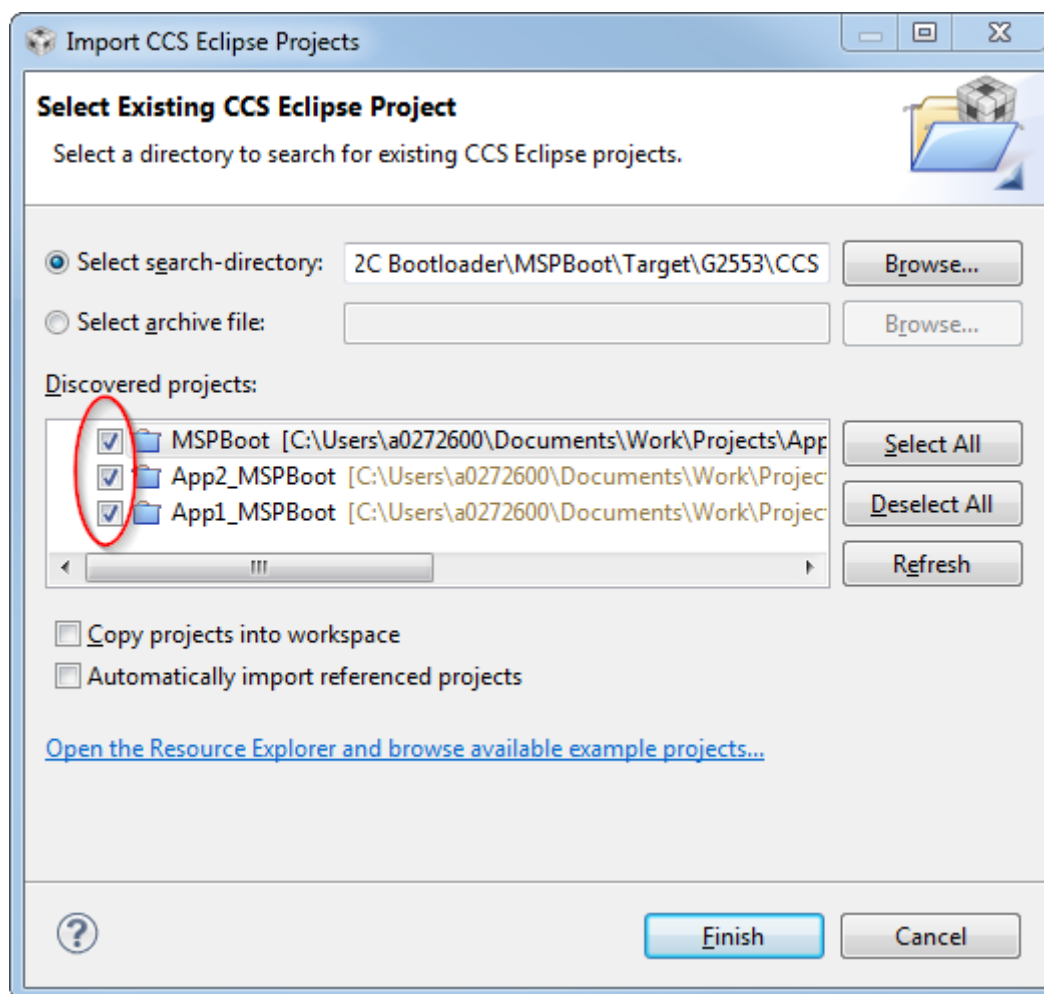


Figure 14. Import MSPBoot CCS Projects

4. Build the bootloader
 - (a) Select the MSPBoot project.
 - (b) Select the proper target configuration based on [Section 3.1](#).

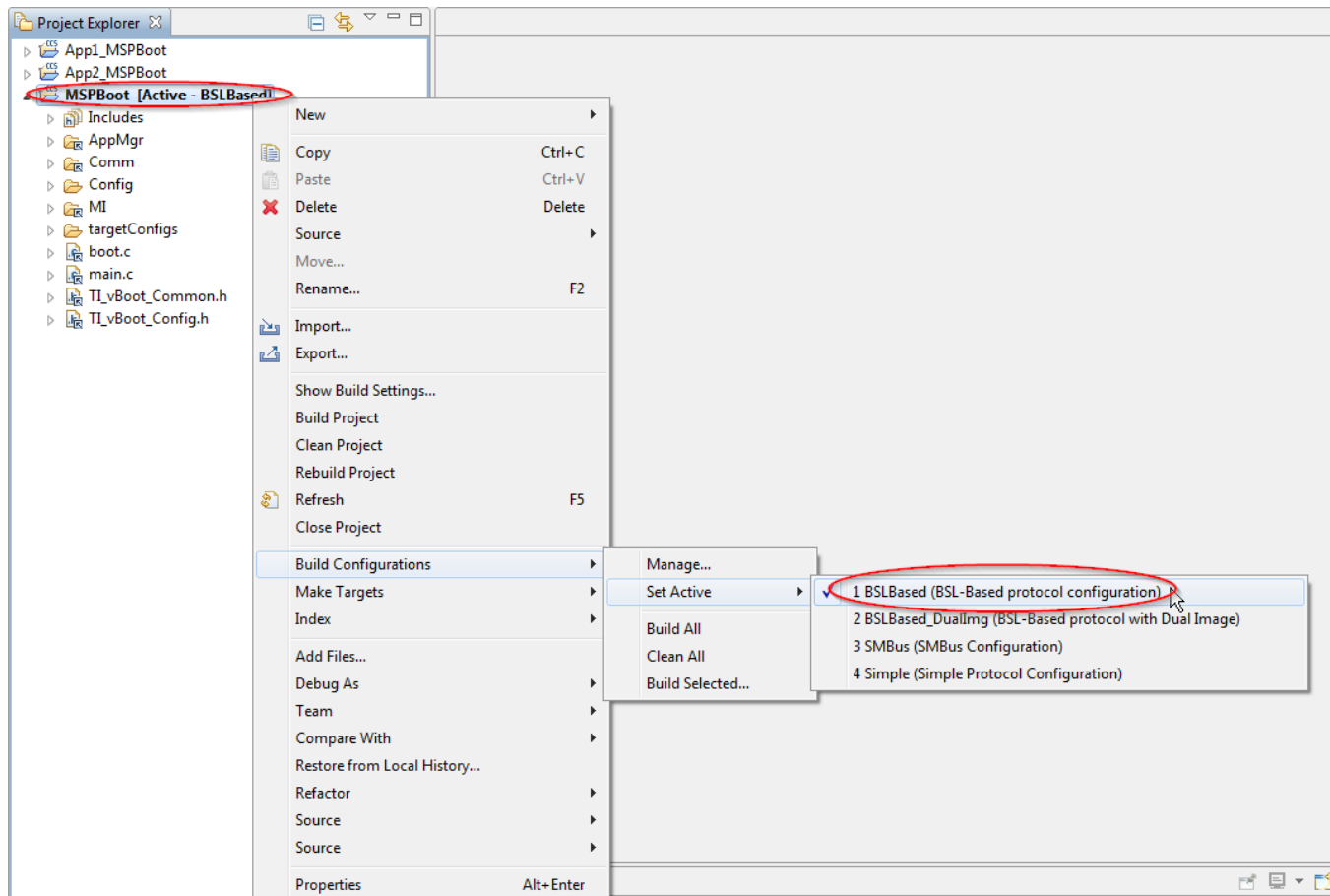




Figure 15. Select Target Configuration

- (c) Click Build  and then Download .

5. Build the applications.
 - (a) Select the App1_MSPBoot project and select the same configuration as the bootloader:

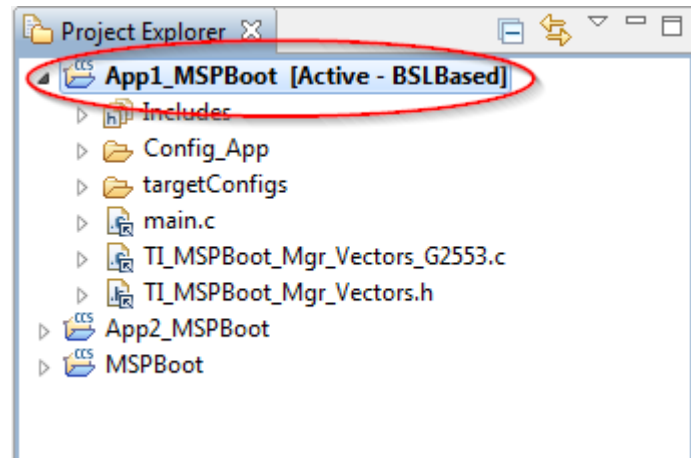



Figure 16. Select App1_MSPBoot Project

- (b) Click the Build project . Note that the output is generated by this step, but the output must be converted and then downloaded by the host processor. [Section 4.2.2](#) describes how to convert the image, and [Section 5](#) and [Section 6](#) describe how to download using a host demo.
 - (c) Repeat step 5 for App2_MSPBoot.

4.2.2 Convert Application Output Images

The CCS and IAR projects generate outputs in MSP430 .txt format in the following directories:

IAR: MSPBoot\Target\<target>\IAR\<App>\<Configuration>\Exe
 CCS: MSPBoot\Target\<target>\CCS\<App>\<Configuration>\

Where,

<target> = FR5739_I2C, G2452_I2C, G2553_I2C, G2553_SPI, G2553_UART
 <App> = App1_MSPBoot or App2_MSPBoot
 <Configuration> = Simple, BSLBased, SMBus, BSLBased_DualImg, BSLBased_SPI, BSLBased_UART

This .txt file does not include CRC and it must be converted to a format that can be used by the host project. To make this easier, the software package includes the following tools.

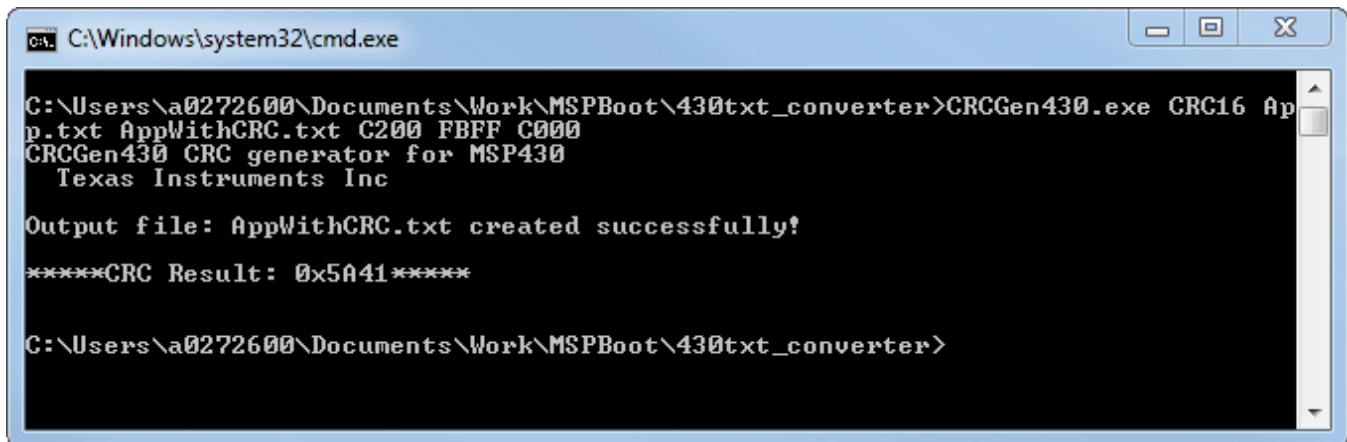
- **CRCGen430:** Calculates the CRC-CCITT or CRC-8 for a memory range using a MSP430 .txt file as input.

Location: MSPBoot\430txt_converter\ CRCGen430.exe

Syntax: CRCGen430.exe CRCType InputFile OutputFile StartAddr EndAddr CRCAddr

Where,

CRCType = CRC16 or CRC8
 InputFile = Input File in .txt format
 OutputFile = Output File in .txt format
 StartAddr = Start address in hexadecimal format
 EndAddr = End address in hexadecimal format
 CRCAddr = Address in hexadecimal format where CRC is stored

Example:


```

C:\Windows\system32\cmd.exe

C:\Users\ao272600\Documents\Work\MSPBoot\430txt_converter>CRCGen430.exe CRC16 AppWithCRC.txt C200 FBFF C000
CRCGen430 CRC generator for MSP430
Texas Instruments Inc

Output file: AppWithCRC.txt created successfully!

*****CRC Result: 0x5A41*****

C:\Users\ao272600\Documents\Work\MSPBoot\430txt_converter>

```

Figure 17. CRCGen430 Example

- **430txt2C:** Perl script that converts an MSP430 .txt file to a C array.

Location: MSPBoot\430txt_converter\ 430txt2C.pl

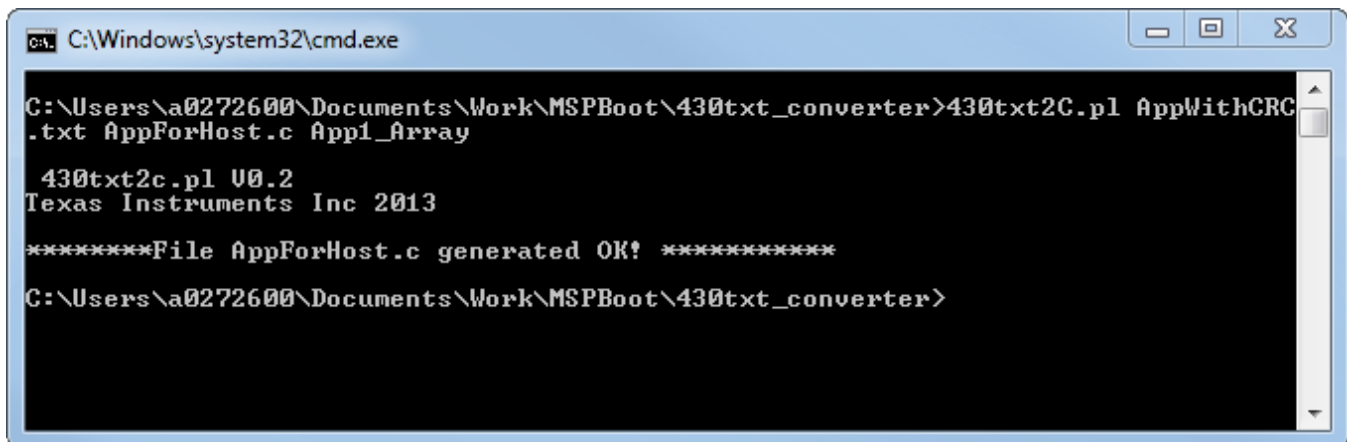
Syntax: 430txt2c.pl src dest struct

Where,

src = Source file in .txt format

dest = Destination file in .c format

struct = Name of Array in C file

Example:


```

C:\Windows\system32\cmd.exe

C:\Users\ao272600\Documents\Work\MSPBoot\430txt_converter>430txt2C.pl AppWithCRC.txt AppForHost.c App1_Array

430txt2c.pl V0.2
Texas Instruments Inc 2013

*****File AppForHost.c generated OK! *****

C:\Users\ao272600\Documents\Work\MSPBoot\430txt_converter>

```

Figure 18. 430txt2C Example

NOTE: A Perl interpreter is required to run this script. Visit <http://www.perl.org/>.

The software package includes several Windows .bat files in this folder:

MSPBoot\430txt_converter

The purpose of these files, in addition to serving as examples, is to automate the process of calculating the CRC for applications, converting to .C, and copying to host projects.

4.2.3 Generating Linker Files

Linker files for CCS and IAR are included for all target configurations and they can be used as a starting point for other devices or custom projects; however, MSPBoot version 1.2 includes a linker generator script that can also help with this process.

- **MSPBootLinkerGen:** Generates application and bootloader linker files for IAR and CCS

Location: MSPBoot\linkerGen\MSPBootLinkerGen.pl

Syntax:

```
MSPBootLinkerGen.pl [-help] -file <filename> -device <dev>
                    [-dual_size <di_size>]
                    [-int_table <int_file>]
                    -params <mem_start> <int_vect> <boot_start>
                    <proxy_size> <sv_size> <RAM_start> <RAM_end>
                    <stack_size> <info_start> <info_end>
```

Where,

-file <filename> = Specifies the prefix of output files. The following files will be generated:

./output/<filename> _Boot.xcl (linker file for Bootloader in IAR)

./output/<filename> _App.xcl (linker file for Application in IAR)

./output/<filename> _Boot.cmd (linker file for Bootloader in CCS)

./output/<./output> _App.cmd (linker file for Application in CCS)

-device <dev> = Specifies the device being used (MSP430G2553)

-dual_size <di_size> = Optional parameter. If defined, enables dual image support with <di_size> specifying the size of each partition. (0x2000 defines a download and Application partitions of 8KB each)

- int_table <int_file> = Optional parameter. If defined, the proxy table is not used and interrupts will be obtained from <int_file>. This is useful for CCS projects using devices like FR57xx which don't need vector redirection.

- params = Defines ten necessary parameters used to defined the memory configuration of the target device.

<mem_start> = Start address of Flash/FRAM (0xC000 for G2553)

<int_vect> = Address of interrupt vector table (0xFFE0 for G2553)

<boot_start> = Start address of Bootloader (0xFC00 for 1KB)

<proxy_size> =Size of the proxy table (48 for 12 vectors using 4B each)

<sv_size> = Size of shared vectors (4 for 2 vectors of 2B each)

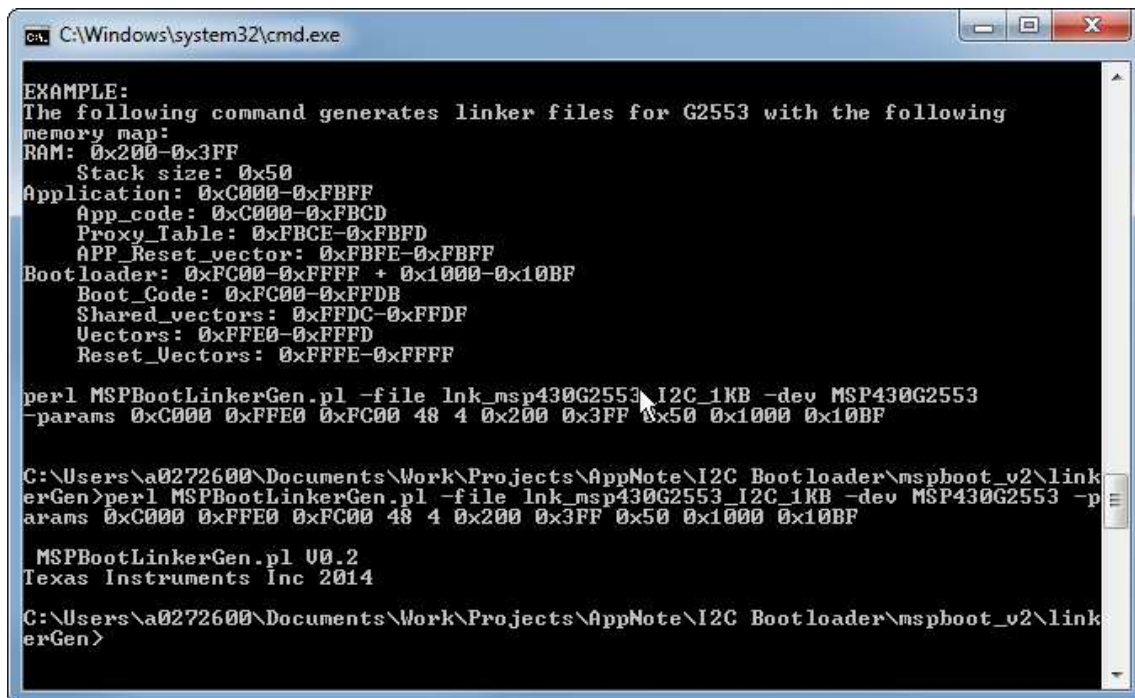
<RAM_start> = Start address of RAM (0x200 for G2553)

<RAM_end> = End address of RAM (0x3FF for G2553)

<stack_size> = Size of the stack (0x50 for G2553)

<info_start> = Start address of info memory used for bootloader (0x1000)

<info_end> = End address of info memory used for bootloader (0x10BF)



```

C:\Windows\system32\cmd.exe

EXAMPLE:
The following command generates linker files for G2553 with the following
memory map:
RAM: 0x200-0x3FF
Stack size: 0x50
Application: 0xC000-0xFBFF
App_code: 0xC000-0xFBCD
Proxy_Table: 0xFBCE-0xFBFD
APP_Reset_vector: 0xFBFE-0xFBFF
Bootloader: 0xFC00-0xFFFF + 0x1000-0x10BF
Boot_Code: 0xFC00-0xFFDB
Shared_vectors: 0xFFDC-0xFFDF
Vectors: 0xFFE0-0xFFFF
Reset_Vectors: 0xFFFE-0xFFFF

perl MSPBootLinkerGen.pl -file lnk_msp430G2553_I2C_1KB -dev MSP430G2553
-params 0xC000 0xFFE0 0xFC00 48 4 0x200 0x3FF 0x50 0x1000 0x10BF

C:\Users\ao272600\Documents\Work\Projects\AppNote\I2C Bootloader\mspbboot_v2\link
erGen>perl MSPBootLinkerGen.pl -file lnk_msp430G2553_I2C_1KB -dev MSP430G2553 -p
arams 0xC000 0xFFE0 0xFC00 48 4 0x200 0x3FF 0x50 0x1000 0x10BF

MSPBootLinkerGen.pl 0.0.2
Texas Instruments Inc 2014

C:\Users\ao272600\Documents\Work\Projects\AppNote\I2C Bootloader\mspbboot_v2\link
erGen>

```

Figure 19. Example Command

NOTE: A Perl interpreter is required in order to run this script. For more information, visit <http://www.perl.org/>.

This script uses templates which are located in the same folder. These templates can be modified as needed but they are required in order to run the script.

The software package includes a Windows .bat file which is used to generate the linker files for all predefined projects and targets configurations. This file can also be used as a base for any further customizations.

5 Demo Using MSP-EXP430F5438 as Host

This software package includes projects and source code for a host device running in MSP-EXP430F5438. This demo uses the LCD and buttons for user interaction, which allows testing protocol commands and sending sample applications to target device. It supports the three MSPBoot protocols and target derivatives.

5.1 Hardware

This demo uses the MSP430F5438A microcontroller running in MSP-EXP430F5438 board:

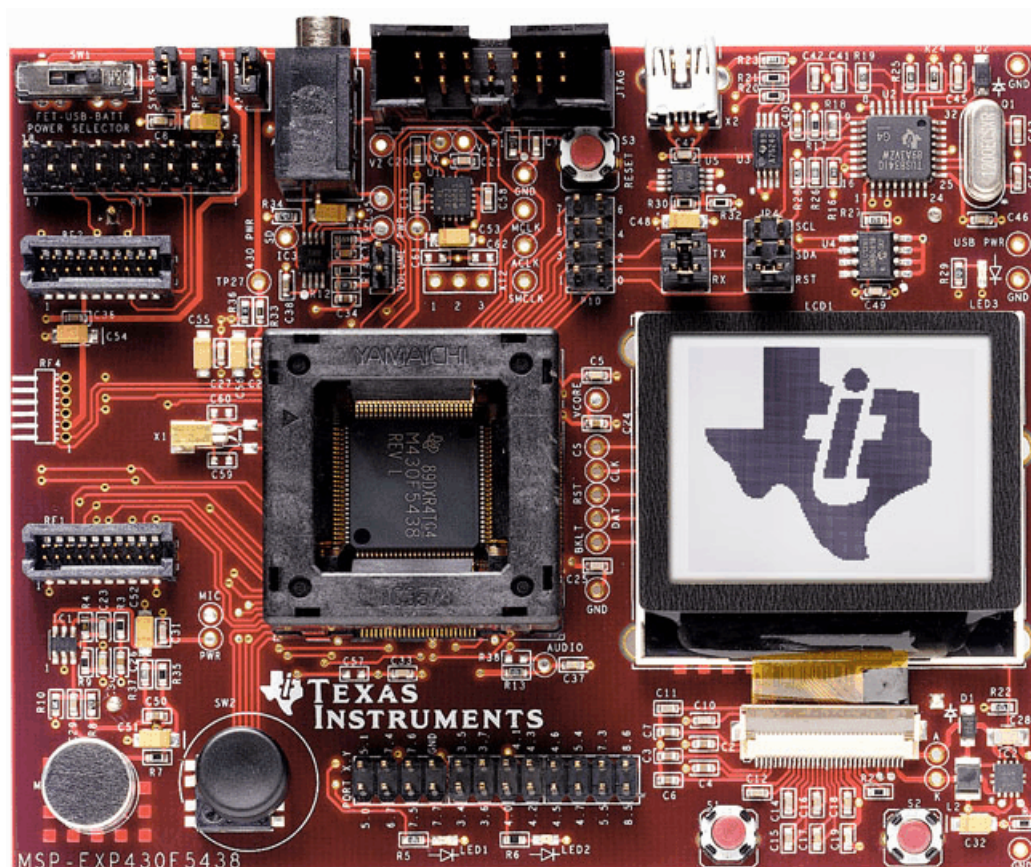


Figure 20. MSP-EXP430F5438

Because the bootloader communicates through I²C, the SDA and SCL lines must be connected in addition to ground. [Table 16](#) shows the pinout when using a target running on MSP-EXP430G2 or MSP-EXP430FR5739.

Table 16. Hardware Connection Between Host and Slave With MSP-EXP430F5438 as Host

Pin	Target		Host
	MSP-EXP430G2	MSP-EXP430FR5739	MSP-EXP430F5438
SCL ⁽¹⁾	P1.6/UCB0SCL ⁽²⁾	P1.7/UCB0SCL	P5.4/UCB1SCL
SDA ⁽¹⁾	P1.7/UCB0SDA	P1.6/UCB0SDA	P3.7/UCB1SDA
GND			

⁽¹⁾ An external pullup is required for the I²C lines (see the I²C specification).

⁽²⁾ Jumper J5, which connects P1.6 to LED2, must be removed in MSP-EXP430G2.

5.2 Building the Host Project

Build the Host project using these steps:

1. Import the project to IAR OR CCS. The project files are located in the following folders:
IAR: MSPBoot\Host\ MSP-EXP430F5438\IAR
CCS: MSPBoot\Host\ MSP-EXP430F5438\CCS
2. Select the target derivative. This can be selected using the different target configurations in IAR or CCS (see [Figure 21](#) and [Figure 22](#)).

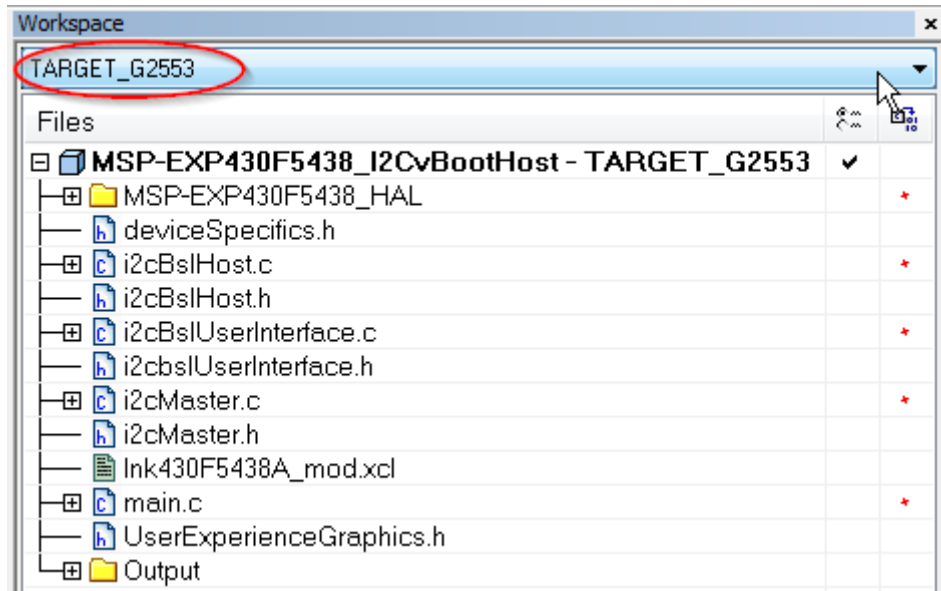


Figure 21. Target Selection for Host Project in IAR

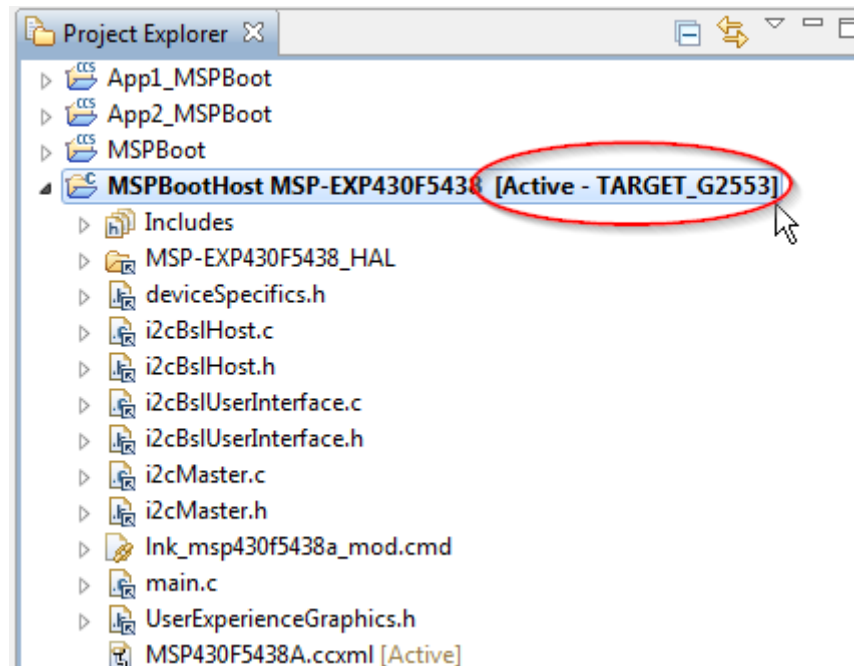


Figure 22. Target Selection for Host Project in CCS

3. Build and download the application.

This project uses application images located in the following folder:

MSPBoot\Host\MSP-EXP430F5438\Src\TargetApps

Pre-built images are included, but target applications can be replaced or updated by following the procedure described in [Section 4.2.1](#) and [Section 4.2.2](#).

5.3 Running the Demo

The MSP-EXP430F5438 LCD and buttons are used to navigate through the commands:

Table 17. Button Navigation in MSP-EXP430F5438

Button	Description
SW2	5-direction joystick used to navigate up and down and to select an option if pressed
S1	Go back
S2	Select current option

The commands shown in [Table 18](#) are available.

Table 18. Host Demo Commands

Menu	Option	Description
Main	1. Simple	Open the Simple Protocol sub-menu
	2. BSL-based	Open the BSL-based protocol sub-menu
	3. SMBus	Open the SMBus-based protocol sub-menu
	4. Duallmg	Open the Dual-Image sub-menu
	5. App Cmds	Open the Application commands sub-menu
Simple	1.1. APP1	Download App 1 using Simple protocol
	1.2. APP2	Download App 2 using Simple protocol
BSL-based	2.1 Version	Request the MSPBoot version
	2.2 APP1	Download App 1 using BSL-based protocol
	2.3 APP2	Download App 1 using BSL-based protocol
	2.4 App Erase	Erase target application area
	2.5 BadChks	Send command with bad CRC (returns error code)
SMBus	3.1 Version	Request the MSPBoot version
	3.2 APP1	Download App 1 using SMBus-based protocol
	3.3 APP2	Download App 1 using SMBus-based protocol
	3.4 App Erase	Erase target Application area
	3.5 Bad Erase	Request target erase with bad PEC (returns error code)
Duallmg	4.1 Version	Request the MSPBoot version
	4.2 APP1	Download App 1 using BSL-based protocol in Dual Image mode
	4.3 APP2	Download App 2 using BSL-based protocol in Dual Image mode
	4.4 App Erase	Erase target virtual application area (Erases "Download" Area)
	4.5 BadChks	Send command with bad CRC (returns error code)
	4.6 Erase XXXX	Erase segment in virtual Application area (Download area)
	4.7 Erase XXXX	Erase segment in physical Download area (returns error)
App Cmds (in App2 when CI PHY-DL is shared)	5.1 LED1	Toggle target LED1
	5.2 LED2	Toggle target LED2
	5.3 LED BOTH	Toggle target LED1 and LED2
	5.4 Force BOOT	Force bootloader mode in the target device

The following steps show how to test the demo using Dual Image mode:

1. Follow the steps in [Section 4.2.1](#) to build and download MSPBoot and to build App1 and App2.
2. Follow the steps in [Section 4.2.2](#) to convert App1 and App2.
 Note: Batch files PrepareIAROutput_F5438.bat and PrepareCCSOutput_F5438.bat show how to calculate CRC, convert to C, and copy the output files.
3. Build and download the host application as described in [Section 5.2](#).
4. Connect boards as described in [Section 5.1](#).
5. Reset and run code on both devices.
6. Enter Target's bootloader mode.
 - (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. Depending on the target configuration, select the appropriate submenu using the host, and test different protocols and images. The following example shows how to test a BSL-based Dual Image target:
 - (a) On the Main menu, select "Duallmg"
 - (b) On the Duallmg menu, select "Version" to request the Boot version of the device.
 The device response is 0xA0 (OK).
 - (c) On the Duallmg menu, select "App Erase" to erase the application area.
 The device response is 0x00 (OK).
 - (d) On the Duallmg menu, select "BadChks" to send a command to the target with an incorrect CRC.
 The device response is 0x52 (Checksum Error, OK)
 - (e) On the Duallmg menu, select "APP1" to send Application 1.
 - (i) LCD shows "APP1 Download OK"
 - (ii) Target executes APP1 blinking LED1 at the beginning
 - (iii) LED1 toggles at a periodic interval using the timer
 - (iv) Go to [Step 6](#) to force bootloader mode
 - (f) On the Duallmg menu, select "APP2" to send Application 2.
 - (i) LCD shows "APP2 Download OK"
 - (ii) Target executes APP2 blinking LED2 at the beginning.
 - (iii) Pressing button S2 toggles LED2
 - (iv) Application commands can be tested:
 - (i) App Cmds menu: LED1 toggles
 - (ii) App Cmds menu: LED2 toggles
 - (iii) App Cmds menu: LED BOTH (LED1 and LED2 toggle)
 - (iv) App Cmds menu: Force BSL
 - (v) Go to [Step 6](#) to force bootloader mode again.

6 Demo Using MSP-EXP430G2 as Host

In addition to the host project for MSP-EXP430F5438, the software package includes a project and source code that uses an MSP-EXP430G2 as the host. This demo shows a more simple implementation than is used with the MSP-EXP430F5438. The MSP-EXP430G2 demo supports only the BSL-based protocol and minimal user interaction, but it can be used to send simple application images to all three target devices and it supports UART and SPI configuration.

6.1 Hardware

This demo uses a MSP430G2553 microcontroller running in an MSP-EXP430G2 board. The connections shown in [Table 19](#) are needed to run the demo.

Table 19. Hardware Connection Between Host and Slave With MSP-EXP430G2 as Host

CI	Pin	Target		Host
		MSP-EXP430G2	MSP-EXP430FR5739	MSP-EXP430G2
I ² C	SCL ⁽¹⁾	P1.6/UCB0SCL ⁽²⁾	P1.7/UCB0SCL	P1.6/UCB0SCL ⁽²⁾
	SDA ⁽¹⁾	P1.7/UCB0SDA	P1.6/UCB0SDA	P3.7/UCB1SDA
UART	RXD ⁽³⁾	P1.2/UCA0TXD	—	P1.1/UCA0RXD
	TXD ⁽³⁾	P1.1/UCA0RXD	—	P1.2/UCA0TXD
SPI	MISO ⁽³⁾	P1.1/UCA0SOMI	—	P1.1/UCA0SIMO
	MOSI ⁽³⁾	P1.2/UCA0SIMO	—	P1.2/UCA0SIMO
	CLK ⁽³⁾	P1.4/UCA0CLK	—	P1.4/UCA0CLK
	SS ⁽³⁾	P1.5	—	P1.5
GND				

⁽¹⁾ An external pullup is required for the I²C lines (see the I²C specification).

⁽²⁾ Jumper J5, which connects P1.6 to LED2, must be removed in MSP-EXP430G2.

⁽³⁾ UART and SPI configuration only enabled with G2553 as target.

6.2 Building the Host Project

Build the Host project using these steps:

1. Import the project to IAR or CCS. The project files are located in the following folders:
IAR: MSPBoot\Host\ MSP-EXP430G2\IAR
CCS: MSPBoot\Host\ MSP-EXP430G2\CCS
2. Select the target derivative. This can be selected using the different target configurations in IAR or CCS (see [Figure 23](#) and [Figure 24](#)).

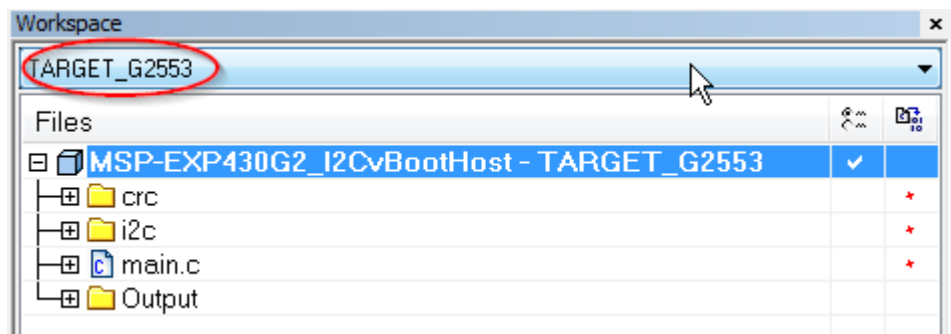


Figure 23. Target Selection for Host Project in IAR

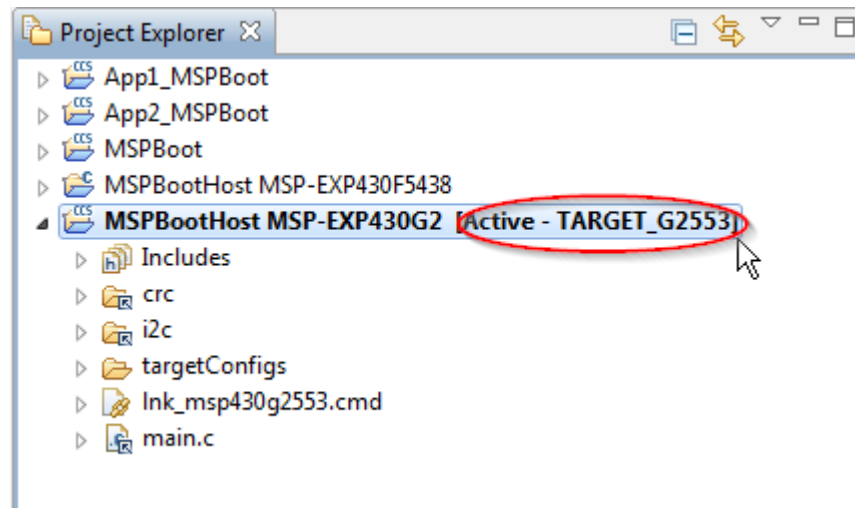


Figure 24. Target Selection for Host Project in CCS

3. Build and Download the application.

This project uses application images located in the following folder:

MSPBoot\Host\MSP-EXP430G2\Src\TargetApps

Pre-built images are included, but target Applications can be replaced or updated by following the procedure described in [Section 4.2.1](#) and [Section 4.2.2](#).

6.3 Running the Demo

The MSP-EXP430G2 Host project sends two different images to the target device and uses a push button for user interaction.

Run the demo using these steps:

1. As described in [Section 4.2.1](#), build and download MSPBoot, and build App1 and App2.
2. Convert App1 and App2 as described in [Section 4.2.2](#).
Note: Batch files PrepareIAROutput_G2.bat, PrepareCCSOutput_G2.bat, PrepareIAROutput_G2_SPI.bat, PrepareCCSOutput_G2_SPI.bat, PrepareIAROutput_G2_UART.bat, and PrepareCCSOutput_G2_UART.bat show how to convert to C and copy the output files. Note that in this host implementation, the microcontroller hold the target image without CRC, so it calculates the CRC value with the assumption that unimplemented locations are 0xFF.
3. Build and Download the host application as described in [Section 6.2](#).
4. Connect boards as described in [Section 6.1](#).
5. Reset and execute code in both devices.
6. To enter the target bootloader mode:
 - (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 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 jump 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) Checks if a slave is present by sending an empty write command. This can cause the following actions:
 - (i) If target device ACKs (that is, if it is running bootloader mode or APP2), the host proceed.
 - (ii) If target device NACKs (that is, if it is running APP1), the host aborts the transfer (see Step 5 to force bootloader mode).
 - (c) Sends the 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.
 - (d) Requests the bootloader version (sends the TX_VERSION command).
 - (i) If the target response is 0xA0-0xAF (expected from BSL protocol), the host continues.
 - (ii) If the target response is any other value, the host aborts the transaction.
 - (e) Erases the target application area (sends the ERASE_APP command)
 - (f) Sends APP1 (uses RX_DATA_BLOCK commands)
 - (g) Programs the CRC of APP1 (uses RX_DATA_BLOCK command)
 - (h) Forces the target application to run (sends JUMP2APP command)
 - (i) Toggle LED1 twice again to indicate successful transfer.
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. The host sends APP2.
10. The target starts running APP2 after completion of transfer.
 - (a) The target device blinks LED2.
 - (b) Press the S2 button on the target board to toggle LED2.
 - (c) Because the Communication Interface 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.

7 References

1. [MSP430x2xx Family User's Guide](#)
2. [MSP430FR57xx Family User's Guide](#)
3. I²C specification 2.1 (<http://www.nxp.com/documents/other/39340011.pdf>)
4. SMBus Specification 2.0 (<http://smbus.org/specs/smbus20.pdf>)
5. [Implementing SMBus Using MSP430 Hardware I2C](#)
6. [MSP430 Programming With the Bootloader \(BSL\)](#)
7. [Creating a Custom Flash-Based Bootloader \(BSL\)](#)

Revision History

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

Changes from March 21, 2014 to February 13, 2017

Page

- Added the paragraph that starts "This software package was built and tested using..." in [Section 4.2, Software](#) 28

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ("TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products (<http://www.ti.com/sc/docs/stdterms.htm>), [evaluation modules](#), and [samples](http://www.ti.com/sc/docs/sampterm.htm) (<http://www.ti.com/sc/docs/sampterm.htm>).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2017, Texas Instruments Incorporated