

# Creating a Custom Flash-Based Bootloader (BSL)

*MSP430 Tools*

## ABSTRACT

MSP430F5xx and MSP430F6xx devices ship with a standard TI BSL. As the TI BSL programmed in Flash memory, the BSL can be reprogrammed and customized. A custom BSL can allow for the creation of custom communication interfaces, start-up sequences, and other possibilities.

This document describes the basics of the BSL memory and describes the TI standard BSL software, so that it can be reused in custom projects. This document also describes a small demonstration BSL that can be used on MSP430G2xx devices. An entry sequence starts the code update and allows the new user code to be sent and stored in flash. A one-byte feedback is provided to indicate status. TA0-based UART communication is used for entry sequence, data, and feedback.

The source code and the firmware image are available in the [Custom MSP430 Bootloader package](#). Note that there is not always a single BSL version programmed for one specific device, and some BSL versions have been programmed for several devices. To obtain complete information which BSL version belong with which device, see Section 5 in the [MSP430™ Flash Device Bootloader \(BSL\) User's Guide](#).

Project collateral and associated source discussed in this application report can be downloaded from the CUSTOM\_BSL430 link at [Bootloader \(BSL\) for MSP low-power microcontrollers](#).

## Contents

1	5xx and 6xx Bootloader Customization .....	3
1.1	BSL Memory Layout.....	3
1.2	Device Start-up Sequence.....	4
1.3	TI-Supplied BSL Software .....	6
1.4	Creation of Custom Peripheral Interface .....	11
1.5	BSL Development and Debug .....	12
2	G2xx Bootloader Creation and Customization .....	13
2.1	Target System Specification .....	13
2.2	BSL Specification.....	13
2.3	Implementation .....	15
2.4	BSL Operation .....	18
3	Frequently Asked Questions (FAQ).....	23

## List of Figures

1	Device Start-up Sequence.....	4
2	Functional Block Diagram, MSP430G2x01 .....	13
3	Demo Setup .....	18
4	Screenshot of Windows® XP Device Manager .....	19
5	Screenshot HTerm COM Port Configuration .....	19
6	View → Register → Calibration_Data (CCS).....	20
7	View → Memory: 0x10FE and 0x10FF (CCS) .....	21
8	Save Device Memory Content to a File in CCS.....	21
9	Select the Project .....	24
10	Setting in Project Properties .....	24
11	Set the PC to the Location of BSL .....	25

12	Run the Debug Session .....	25
13	Debug Setup .....	26
14	Build Project .....	27
15	Set PC to BSL Address .....	27
16	BSL is Executed .....	28

### List of Tables

1	Pin Assignment .....	15
2	Linker Command File .....	16
3	Modify Linker File .....	16
4	Use Custom Linker File .....	17
5	Project Settings .....	17
6	User Application .....	17
7	BSL Demo Setup .....	18

### Trademarks

Code Composer Studio, MSP430, LaunchPad are trademarks of Texas Instruments.  
 Windows is a registered trademark of Microsoft Corporation.  
 All other trademarks are the property of their respective owners.

## 1 5xx and 6xx Bootloader Customization

### 1.1 BSL Memory Layout

The BSL memory is a 2KB section of flash. The specific location of this flash is described in each device-specific data sheet. However, it is typically found between addresses 0x1000 and 0x17FF. Varying sections of this memory can also be protected. This protection is enabled by setting flags in the SYSBSLC register, which is described in the [MSP430F5xx and MSP430x6xx Family User's Guide](#).

#### 1.1.1 Z-Area

When protected, the BSL memory cannot be read or jumped into from a location external to BSL memory. This protection makes the BSL more secure against erase and also prevents erroneous BSL execution. However, if the entire BSL memory space were protected in this way, it would also mean that user application code could not call the BSL in any way, such as an intentional BSL start-up or using certain public BSL functions.

The Z-Area is a special section of memory designed to allow for a protected BSL to be publically accessible in a controlled way. The Z-Area is a section of BSL memory between addresses 0x1000 and 0x100F that can be jumped to and read from external application code. The Z-Area functions as a gateway from which a jump can be performed to any location within the BSL memory. The default TI BSL uses this area for jumps to the start of the BSL and for jumps into BSL public functions.

#### 1.1.2 BSL Reserved Memory Locations

The BSL memory also has specific locations reserved to store defined values to ensure proper BSL start:

##### 0x17FC to 0x17FF

JTAG Key: If all bytes are either 0x00 or 0xFF, then the device has open JTAG access. Any other values prevent JTAG access. See the [MSP430F5xx and MSP430x6xx Family User's Guide](#) for more details about the JTAG Key.

##### 0x17FA

BSL Start Vector. This is the address of the first instruction to be executed when the BSL starts.

##### 0x17F8

Reserved

##### 0x17F6

BSL Unlock Signature 1. This word should be set to 0xC35A to indicate a correctly programmed BSL. If it is written with any other value, the BSL will not start.

##### 0x17F4

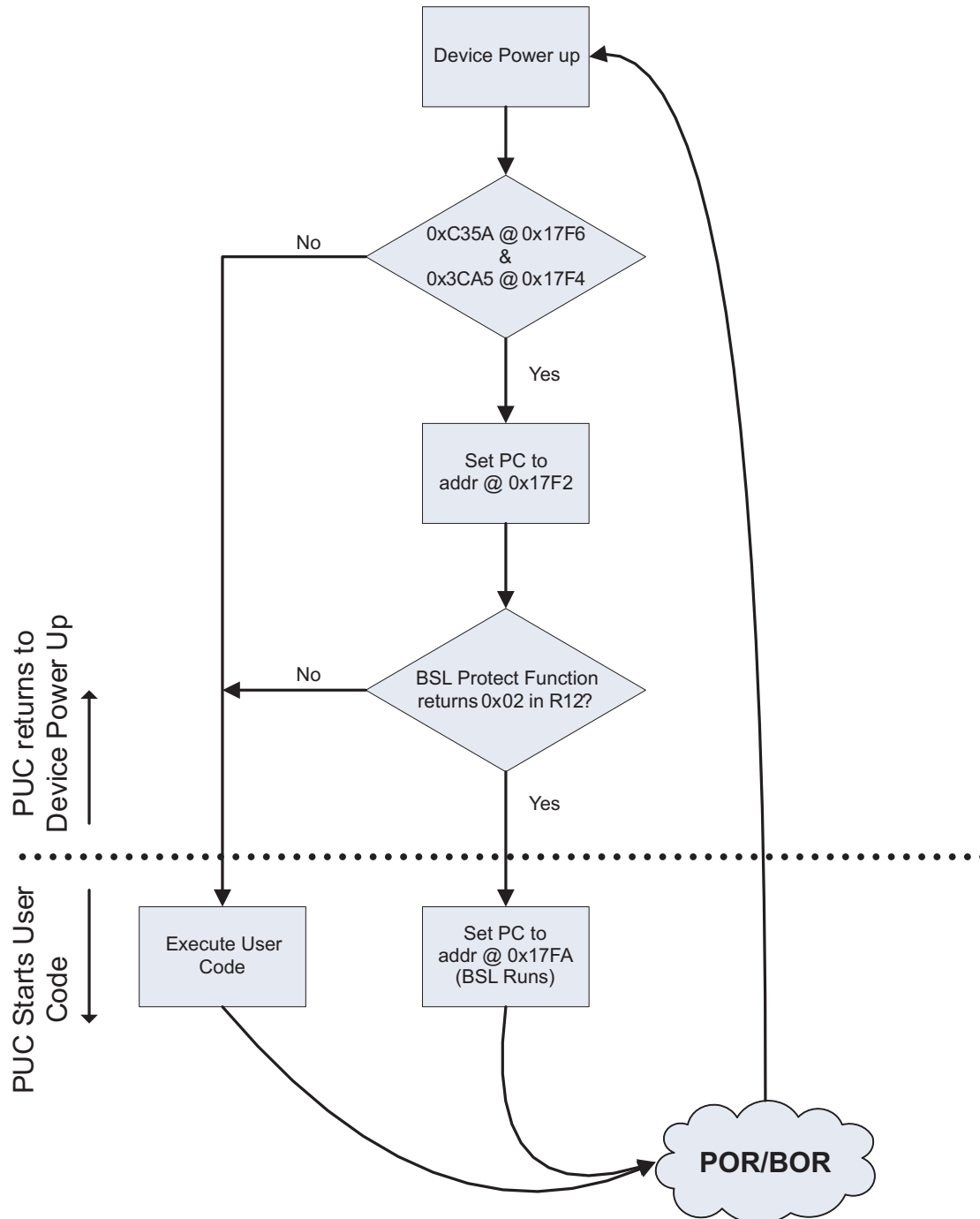
BSL Unlock Signature 2. This word should be set to 0x3CA5 to indicate a correctly programmed BSL. If it is written with any other value, the BSL will not start.

##### 0x17F2

BSL Protect Function Vector. This is the address of the first instruction in the BSL protect function. More details on this function are provided in the following sections.

## 1.2 Device Start-up Sequence

After power up, the device first checks the BSL signature. If the appropriate values are found, the device calls the BSL protect function. The BSL protect function is described in detail in [Section 1.2.1](#), but its main responsibilities are to secure the BSL memory and to indicate if the BSL should be started instead of the user application. The device start-up sequence occurs according to the flow chart in [Figure 1](#).



**Figure 1. Device Start-up Sequence**

### 1.2.1 BSL Protect Function

The BSL Protect function is called after each BOR and before user code is executed. There is no time and functionality limit placed on this code; however, if this function does not return, the device is rendered unresponsive. Excessively long delays in the return functions could lead to problems during debug. At its most basic, the BSL Protect function should perform two essential operations: protect the BSL memory and determine whether the BSL or user code should be executed after exiting the BSL Protect function.

The BSL Protect function is called with the stack pointer set to a default location, which is dependent on the device. Changing the stack pointer or manipulation of the stack pointer data values will most likely lead to a unresponsive device. In this case, nothing, not even reprogramming, will be possible. On some devices, the stack pointer space is very limited, and extensive use of the stack pointer within the BSL Protect function could lead to memory overflows. To ensure proper behavior, the stack access should be limited or the stack pointer should be moved to another location. To make sure that the device returns from the BSL Protect function correctly, the stack pointer must be restored before returning.

#### 1.2.1.1 Protection of BSL Memory

Protection of BSL memory is done by setting the size and protection bits in the SYSBLSC register. Setting the SYSBLSC register is performed regardless of whether or not a BSL invoke is to be requested. For more information on these specific bits, see the [MSP430F5xx and MSP430x6xx Family User's Guide](#).

#### 1.2.1.2 Checking for BSL Invoke

The BSL Protect function also must indicate whether the BSL or application code (if present) should start. This selection is made by setting bits in R12 to these defined values:

##### Bit 0

- 0: Indicates that the JTAG state should be determined based on JTAG Key state.
- 1: Overrides the JTAG Key, keeping JTAG open (used primarily for debugging the BSL).

##### Bit 1

- 0: Indicates the BSL should not be started.
- 1: Indicates the BSL should be started by loading the value in the BSL Start Vector to the PC.

The method for determining whether the BSL should be invoked is entirely dependent on the BSL Protect function. TI-supplied UART BSLs check the SYSBSLIND bit to check for the occurrence of a specific pin-toggle sequence. The TI-supplied USB BSLs, however, have an entirely different start-up criteria based on their application requirements: if USB power is present and the device is blank, the BSL Protect function indicates that the BSL should start, so the blank device may be programmed through USB. However, in a custom BSL, almost any criteria could be used, such as checking the user code against a known CRC value to make sure that only correctly programmed user code begins execution.

## 1.3 TI-Supplied BSL Software

A preprogrammed BSL is supplied with each MSP430x5xx and MSP430x6xx device. Use of this BSL is described in [MSP430™ Flash Device Bootloader \(BSL\) User's Guide](#). Familiarity with the BSL at this level is assumed for the following section. The TI-supplied BSLs were written for and compiled with IAR KickStart. The code is also deployed for Code Composer Studio™ IDE. Standard C-based BSL logic code is maintained the same for IAR and CCS. Differences between the IAR and CCS code depend on pragma and initialization style.

### 1.3.1 Software Overview

The TI-supplied BSL is written in such a way that it is extremely modular. Depending on the level of customization needed, various source files can be reused or replaced.

The three main sections of BSL code are:

#### Peripheral Interface

This section of code has responsibility for receiving and verifying a BSL Core Command. To accomplish this, the code can use any hardware or protocol (wrapper bytes or other options). The actual transmission mechanism and protocol are irrelevant for the rest of the BSL. What is important is that when the BSL is called upon to receive a packet, it does not return until it has correctly received all data sent to it. Because the BSL is used for erasing and programming user memory, the Peripheral Interface cannot use flash-based interrupt vectors. Either event polling or RAM-based interrupt vectors must be used.

#### Command Interpreter

This section of code interprets the supplied BSL Core Command bytes. The code can assume that the Peripheral Interface has successfully received the bytes without error but not necessarily that the bytes are correct for the BSL protocol (for example, if an incorrect command is sent). This code interprets the received Core Command according to the BSL Core Command list (see the [MSP430™ Flash Device Bootloader \(BSL\) User's Guide](#)) and executes received commands by calling the BSL API.

#### BSL API

This section of code provides an abstraction layer between the command interpreter and the memory being written to or read from. It handles all aspects of unlocking memory, writing to it, reading from it, and CRCs. It also, whenever possible, is the section of code where security is handled. This allows the command interpreter to simply make requests and send responses without being concerned with security issues. This model also allows for extremely secure custom BSLs to be made, as it is assumed the BSL API is the least likely section to be replaced in any custom BSL, so any custom BSL benefits from TI-supplied security checks and measures.

### 1.3.2 Software File Details

This section does not describe all functions in detail (for example, parameters and return values). This information is contained in the function comments in the header and source files. The goal of this section is to highlight important responsibilities of individual files that might not be immediately obvious by reading the source.

#### 1.3.2.1 BSL430\_Low\_Level\_Init.s43 (IAR) / BSL430\_Low\_Level\_Init.asm (CCS)

This file handles low-level BSL aspects, such as reserved memory locations, the BSL\_Protect function, and publically available functions in the Z-Area. Usually, the customization required here is limited to:

- Changing the BSL\_Protect function to invoke the BSL on different conditions.
- Writing values that are not 0xFFFF to the JTAG key location for a final BSL image.
- Adding additional functions to the Z-Area to make them executable by user code.

### 1.3.2.2 **BSL\_Device\_File.h**

This file is used to abstract certain device variations and allow all BSL source code files to remain identical between devices. It is where all device-specific information can be found, such as device include file, port redefinitions, clock speeds, and so on. In addition, there are custom BSL definitions as described below. Note that not all definitions are valid for all BSLs and may not appear in this file.

#### **MASS\_ERASE\_DELAY**

The delay value used before sending an ACK on an unlock

#### **INTERRUPT\_VECTOR\_START**

The definition for the start address of the BSL password

#### **INTERRUPT\_VECTOR\_END**

The definition for the end address of the BSL password

#### **SECURE\_RAM\_START**

The start of the RAM that should be overwritten at BSL start, for security. This is usually the lowest RAM value. The RAM between this address and the stack pointer is cleared.

#### **TX\_PORT\_SEL**

This collection of port definitions is used to abstract the TX and RX ports away from specific port pins, so that these values may change between devices without requiring a change to the Peripheral Interface source code.

#### **RAM\_WRITE\_ONLY\_BSL**

When defined, this causes the BSL to compile to a much smaller version, which can only write to RAM in a device and receive the commands required for unlock, set PC, and RX data. It is used in the USB BSL to save space and fit the USB stack in the 2KB BSL memory. A RAM write-only BSL is used to load a complete BSL into RAM and start it for further communication.

#### **RAM\_BASED\_BSL**

When defined, this includes sections of code in the API that are required when the BSL is running out of RAM and not the BSL memory. This is primarily for delays on flash writing and is used only in BSLs running out of RAM such as the USB BSL.

#### **USB\_VID**

The Vendor ID for the USB BSL

#### **USB\_PID**

The Product ID for the USB BSL

#### **SPEED\_x**

This is used to define the speeds of external oscillators that should be tested for in the USB BSL start-up sequence. The values here are the raw values in Hertz (for example, 24000000 for 24 MHz).

#### **SPEED\_x\_PLL**

The corresponding PLL definition from the device header file for the oscillator speed described in the SPEED\_x definition.

### 1.3.2.3 **Ink430FXXXX\_BSL\_AREA.xcl (IAR) / MSP430Fxxxx\_BSL.cmd (CCS)**

This is a custom linker command file that places the project code output into the BSL memory locations. In addition, it has definitions for the reserved BSL memory sections described previously. In general, it should not need modification. In some BSL linker command files, the reset vector output is intentionally placed out of the correct device RESET vector location, as this would cause the device to continually try to jump into the protected BSL memory, which would cause a reset, and thus an infinite reset loop.

### 1.3.3 Known Limitations in CCS CSL Code Example

The CCS compiler is known to generate the BSL images greater than 2KB (the BSL area in the information memory is at address 0x1000 to 0x17FF). The software package MSPBSL\_CustomBSL version 1.2.0 and newer, modifies the linker command file to fit the code in the BSL area. To do this, some BSL functions are commented out on BSL\_Device\_File.h to fit the image into 2KB. To change which functions are included in the build, modify the BSL\_Device\_File.h.

#### 1.3.3.1 Memory Allocation of BSL Code Under Linker Command File

Starting with the new version of MSPBSL\_CustomBSL430 1.2.0, the BSL code is placed under “Flash” section which is located in the Information Memory BSL area.

Memory sections of BSL code

```
...
ZAREA                : origin = 0x1000, length = 0x0010
BSL430_VERSION_VENDOR : origin = 0x1010, length = 0x0001
BSL430_VERSION_CI     : origin = 0x1011, length = 0x0001
BSL430_VERSION_API    : origin = 0x1012, length = 0x0001
BSL430_VERSION_PI     : origin = 0x1013, length = 0x0001
ZAREA_CODE            : origin = 0x1014, length = 0x002E
FLASH                 : origin = 0x1042, length = 0x07AE
BSLSIG                : origin = 0x17F0, length = 0x000C
JTAGLOCKKEY           : origin = 0x17FC, length = 0x0004
...
```

Assignment between build binary and memory sections

```
...
.ZAREA          : {} > ZAREA
.BSL430_VERSION_VENDOR : {} > BSL430_VERSION_VENDOR
.BSL430_VERSION_CI     : {} > BSL430_VERSION_CI
.BSL430_VERSION_API    : {} > BSL430_VERSION_API
.BSL430_VERSION_PI     : {} > BSL430_VERSION_PI
.ZAREA_CODE       : {} > ZAREA_CODE
.BLSIG            : {} > BLSIG
.JTAGLOCKKEY      : {} > JTAGLOCKKEY

.text            : {} >> FLASH          /* CODE                      */
.text:_isr       : {} > FLASH          /* ISR CODE SPACE           */

.cinit           : {} > FLASH          /* INITIALIZATION TABLES   */
.const           : {} > FLASH          /* CONSTANT DATA           */

.cio             : {} > RAM            /* C I/O BUFFER             */

.pinit           : {} > FLASH          /* C++ CONSTRUCTOR TABLES */
.init_array      : {} > FLASH          /* C++ CONSTRUCTOR TABLES */
.mspabi.exidx    : {} > FLASH          /* C++ CONSTRUCTOR TABLES */
.mspabi.extab    : {} > FLASH          /* C++ CONSTRUCTOR TABLES */
```

#### 1.3.3.2 BSL Functions Supported in the Default Setting Project

By default, the BSL functions have to be separated into use cases, so that the build will fit in into 2kB. Under BSL\_Device\_File.h, the use cases are defined. User can replace the required functions under each use cases during compilation.



```
#define PROGRAMMING_FUNCTIONS
//#define MEMORY_READING_FUNCTIONS
//#define OTHER_FUNCTIONS

#ifdef PROGRAMMING_FUNCTIONS
#define SUPPORTS_RX_PASSWORD
#define SUPPORTS_ERASE_SEGMENT
#define SUPPORTS_TOGGLE_INFO
#define SUPPORTS_RX_DATA_BLOCK
#define SUPPORTS_CRC_CHECK
#endif

#ifdef MEMORY_READING_FUNCTIONS
#define SUPPORTS_RX_PASSWORD
#define SUPPORTS_TX_DATA_BLOCK
#define SUPPORTS_CRC_CHECK
#endif

#ifdef OTHER_FUNCTIONS
#define SUPPORTS_RX_DATA_BLOCK_FAST
#define SUPPORTS_TX_BSL_VERSION
#define SUPPORTS_MASS_ERASE
#define SUPPORTS_LOAD_PC
#endif
```

### 1.3.3.3 How to Accomodate Full Function of BSL

To be able to use all BSL functions, some code need to be placed in the other than BSL information memory area. One most important thing to be considered is that the BSL area in the information memory is protected by default, but not with the main memory. The user needs to understand the BSL flow to erase the memory carefully during programming so that the BSL code will not be accidentally erased.

It is recommended to also place the additional section after the interrupt vector area in the main memory which is located from 0xFF80-0xFFFF.

To allocate BSL in main memory, the linker command file need to have second flash section. For example:

Memory sections of BSL code

```
ZAREA          : origin = 0x1000, length = 0x0010
BSL430_VERSION_VENDOR : origin = 0x1010, length = 0x0001
BSL430_VERSION_CI    : origin = 0x1011, length = 0x0001
BSL430_VERSION_API   : origin = 0x1012, length = 0x0001
BSL430_VERSION_PI    : origin = 0x1013, length = 0x0001
ZAREA_CODE          : origin = 0x1014, length = 0x002E
FLASH               : origin = 0x1042, length = 0x07AE
FLASH2              : origin = 0x10000, length = 0x1000
BSLSIG              : origin = 0x17F0, length = 0x000C
JTAGLOCKKEY         : origin = 0x17FC, length = 0x0004
```

Assignment between build binary and memory sections

```

...
.text      : {} >> FLASH | FLASH2      /* CODE                                */
.text:_isr  : {} > FLASH | FLASH2      /* ISR CODE SPACE                      */

.cinit     : {} > FLASH | FLASH2      /* INITIALIZATION TABLES             */
.const     : {} > FLASH | FLASH2      /* CONSTANT DATA                     */

.cio       : {} > RAM                  /* C I/O BUFFER                       */

.pinit     : {} > FLASH | FLASH2      /* C++ CONSTRUCTOR TABLES            */
.init_array : {} > FLASH | FLASH2      /* C++ CONSTRUCTOR TABLES            */
.mspabi.exidx : {} > FLASH | FLASH2    /* C++ CONSTRUCTOR TABLES            */
.mspabi.exstab : {} > FLASH | FLASH2    /* C++ CONSTRUCTOR TABLES            */
...

```

### 1.3.3.4 Using Modified *boot\_hook.h* and *boot.c* (CCS Only)

Using modified boot source code files can help generating smaller BSL code size in the CCS environment. By default, CCS compilation includes the boot initialization that is defined in "ccs\tools\compiler\ti-cgt-msp430\_xx.xx.x.LTS". By using the modified *boot.c* (the header of *boot\_hooks.h* is also required), the default code that is not required for BSL initialization is eliminated. However this step requires further understanding about how the *boot.c* and its content work. It is recommended to still use default settings that CCS provide. Modified *boot.c* and *boot\_hook.h* are provided in the folder "MSPBSL\_CustomBSL430\_1-2- 0/CCS\_BSL\_Source/Custom\_boot".

## 1.4 Creation of Custom Peripheral Interface

The existing BSL software can easily be ported to a different communication interface. This is possible because the Peripheral Interface code that handles communication is very loosely tied to the rest of the BSL software. To implement a custom Peripheral Interface, the following functions in the file BSL430\_PI.h must be defined.

### 1.4.1 PI\_init ()

This function has three primary responsibilities:

- Initialize the communication peripheral so that it is in a state to begin receiving data.
- Initialize the device clock system (usually 8 MHz but can be anything as other code is independent).
- Set the BSL430\_Command\_Interpreter values "BSL430\_ReceiveBuffer" and "BSL430\_SendBuffer" to the location where data packets will be stored. This can be the same buffer in RAM (such as for UART) or different locations used by a peripheral (such as for USB). It should be noted that these pointers need to point to the first byte in the BSL Core Command. So if a Peripheral Interface buffer uses a larger buffer in RAM for wrapper bytes, these pointers should point inside that buffer.

### 1.4.2 PI\_receivePacket()

This is the primary loop function of the BSL. This function receives all bytes for a Core Command and returns a value to the Core Command based on the results. The return value definitions are:

#### DATA\_RECEIVED

The data was successfully received. No checks were performed to verify whether or not the command itself is valid; however, the bytes were correctly received as sent by the host and can be safely processed.

#### RX\_ERROR\_RECOVERABLE

Some error occurred during receiving of a packet. The packet is lost, but the receive function can be called again.

#### RX\_ERROR\_REINIT

Reserved for when an error occurs during receiving of a packet that would require the PI\_init() function be called again before receiving further packets.

#### RX\_ERROR\_FATAL

Reserved for when an error occurs during receiving of a packet that renders further communication impossible. A complete system restart is required.

#### PI\_DATA\_RECEIVED

Indicates that a packet was received and processed by the Peripheral Interface. No action is required other than calling the PI\_receivePacket() function again.

### 1.4.3 PI\_sendData(int bufSize)

This function is called by the Command Interpreter when it has filled the send buffer with a reply. The size of the data within the buffer is passed as a parameter so the Peripheral Interface can send the correct number of bytes.

## 1.5 BSL Development and Debug

### 1.5.1 Development and Testing

Due to the protections in the BSL memory, it can often be difficult to debug and develop BSL code. For "high level" development, such as a new Peripheral Interface, it is easy to develop the BSL as an application that runs out of user code flash.

- Remove the BSL430\_Low\_Level\_Init.s43 from the project build (right click the file in IAR, select "remove from build").
- Use a linker command file from the CONFIG directory that puts the BSL in the "FLASH\_AREA" (Project → Options → Linker → Config → Override default).
- Run the external application (BSL\_Scripter.exe for example) without causing a device reset during the BSL invoke sequence.
- Do not send an incorrect password or trigger a mass erase (remember, IAR builds the RESET vector automatically, so the password includes this).

For development and debugging in the BSL\_Protect function, either the simulator can be used or the BSL memory protection bits can be left open during debugging. In either case, "Run to Main" should be unchecked.

### 1.5.2 Special Notes and Tips

The BSL code uses RAM at the highest and lowest addresses in RAM. For this reason, if a BSL will target multiple devices, its target (in IAR, Project → Options → General Options → Device) should be the device with the smallest amount of RAM. Also SECURE\_RAM\_START should be set to the highest shared address.

IAR version 5.51 was used to build all device BSLs.

### 1.5.3 USB BSL External Oscillator Frequency

The USB BSL uses a routine to measure the speed of the external oscillator that is used in the application. The BSL does this by comparing the speed of the external clock to a known calibrated internal clock. In this way, the default BSL can be used without modification with certain specific external oscillator frequencies. If other frequencies are to be used in an application, the SPEED\_x and corresponding SPEED\_x\_PLL values can be changed. They must be in order from highest to lowest speed. If only one speed will be used, all values must still be defined, but can be defined as the same frequency.

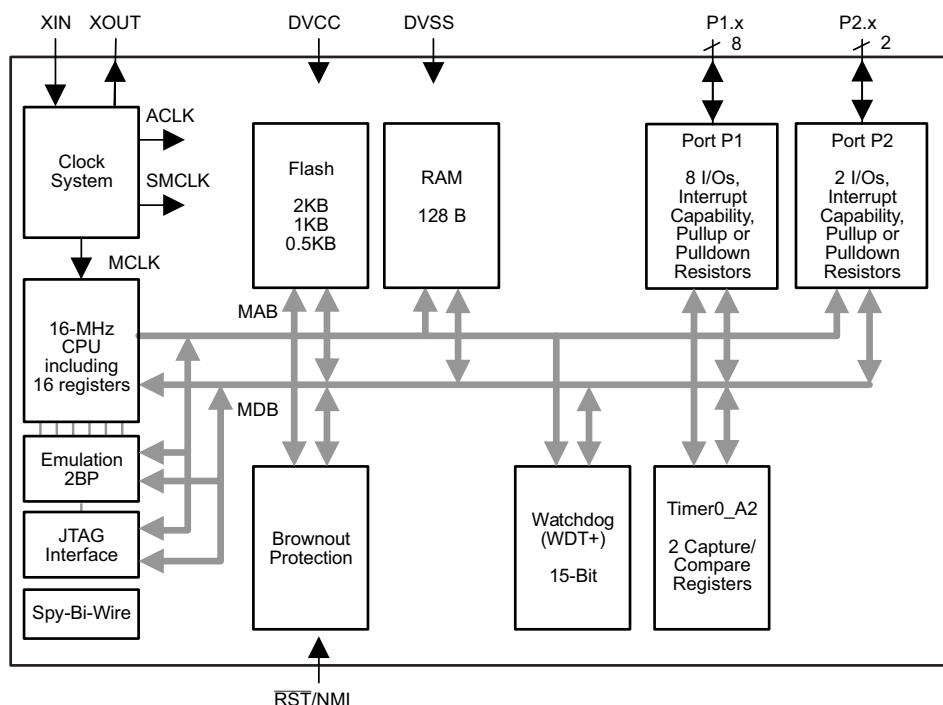
```
//9MHz Example Code
#define SPEED_1          9000000
#define SPEED_1_PLL      USBPLL_SETCLK_9_0
#define SPEED_2          SPEED_1
#define SPEED_2_PLL      SPEED_1_PLL
#define SPEED_3          SPEED_1
#define SPEED_3_PLL      SPEED_1_PLL
#define SPEED_4          SPEED_1
#define SPEED_4_PLL      SPEED_1_PLL
```

Even in the case where only one known frequency is used, as shown in this code example, it is important to keep the measurement loop in the USB Peripheral Interface, as it is also used for a delay to allow for crystal start-up.

## 2 G2xx Bootloader Creation and Customization

### 2.1 Target System Specification

This bootloader is designed for use on a MSP430G2001, which is currently the smallest MSP430™ Value Line device member. However, it can be easily modified to fit other G2xx devices—required modifications are shown in following sections.



**Figure 2. Functional Block Diagram, MSP430G2x01**

For the BSL, the memory size matters:

RAM: 128 bytes

Flash: 512 bytes, one segment on MSP430G2001

Info A: 64 bytes minus 2-byte DCO calibration data for 1 MHz

Info B, C, D: 64 bytes each

### 2.2 BSL Specification

The BSL presented in this document followed one simple requirement: Simplicity. In addition, the available resources on the device greatly influenced the way this BSL works.

**NOTE:** This BSL is not to be confused with other TI BSLs. It is not based on any other MSP430 BSL, neither ROM nor flash.

## 2.2.1 Functionality

With respect to the available code size, only one function is implemented. The flow is described in the following sections. Note that it is possible to run the code only in exactly this order.

### 2.2.1.1 Entry Sequence

Entering the BSL is possible only when the device comes out of a reset event and the BSL\_entry pin is pulled low.

### 2.2.1.2 Synchronization

After the entry sequence, the BSL is awaiting a SYNC character. This is used to make sure the BSL entry was not by accident.

Note that the device loops forever until one byte is received. If it is identical to the SYNC character, the BSL continues its operation; otherwise, it creates a reset event.

SYNC character = 0xBA

### 2.2.1.3 Erasing Previous Flash Content

After the SYNC character is successfully received, the BSL immediately erases the main flash to get ready for new data.

User code resides in main flash but shares its code space with the interrupt vector table. Therefore, the BSL needs to take immediate action to restore the reset vector that points to BSL. This takes approximately 16 ms per flash segment, but the device voltage should be kept constant for at least double the time. See the device-specific data sheet to determine how many flash segments exist on a device.

#### CAUTION

During the time from flash erase until the reset vector has been restored, the system is highly vulnerable to a potential lockout situation in which it is impossible to get access to the device again.

### 2.2.1.4 Receiving and Writing New User Data

After the erase happens, the BSL is ready to accept user data, one byte at a time. There is no handshake in place, so the system expects exactly the right amount of data, which is the total available main flash size minus 2 bytes, starting at the lowest flash address all the way up to, but excluding, the reset vector.

### 2.2.1.5 Data Verification

After all of the data is received, a single-byte XOR checksum is transmitted to device. The BSL compares this checksum against the checksum that it calculates over the flash data that was written.

The BSL returns an ACK character (0xF8) if the checksums match. The BSL returns a NACK (0xFE) if the checksums do not match.

## 2.2.2 Memory Footprint

The most difficult challenge in implementing a bootloader on such a small device is code size. By its nature, a BSL must always remain in the memory to avoid a lockout situation where it is no longer possible to access the device.

To keep the BSL code untouched during BSL user code updates, the BSL needs to reside in a different flash section than user code. As there is only one main flash section available on MSP430G2001 devices, the BSL needs to reside in the only flash part left: information memory with a total useable size of 254 bytes.

### 2.2.3 Peripherals

Some MSP430G2xx devices come without any communication module. Therefore, a simple software-based interface is used. Handshake-less UART is used because of its simple communication scheme. Timer\_A and two GPIO pins are used for an efficient implementation of this software UART with the following settings:

9600 baud, 8 data bits, 1 stop bit, no parity

In addition, one GPIO pin is used to initiate the BSL start sequence in conjunction with a device reset using the RST pin.

**Table 1. Pin Assignment**

Pin	Function	LaunchPad Pin
P1.3	BSL_entry	Switch S2
P1.1	UART RxD	Cross connect! TXD
P1.2	UART TxD	Cross connect! RXD
RESET	RESET	RESET

## 2.3 Implementation

The project is split into two separate parts: The BSL and the main or user application. The user code is available in C or ASM to fit various needs.

### 2.3.1 BSL Assembler Code

All code for the BSL can be found in MSP430G2xxBSL\_CCS(IAR).asm. It consists of:

- Reset interrupt table statement (entry point)
- BSL code
- A small user application example that blinks the LED for demo purposes only

#### 2.3.1.1 Save DCO Calibration Data

Because the BSL code resides in information memory, it is important to save and restore the DCO calibration data that is also stored in InfoA. During prototyping, this can be done manually:

```
; =====
; Note that the user needs to ensure that DCO Cal Data is not
; erased during debugging - Read out from InfoA and hardcode
; two bytes in move commands below.
mov.b  #YOUR_DEVICE_VALUE,&DCOCTL    ; Copy from address 0x010FEh
mov.b  #YOUR_DEVICE_VALUE,&BCSCTL1   ; Copy from address 0x010FFh
; Replace YOUR_DEVICE_VALUE with values gathered from actual
; device. Values look like that: 0b3h and 086h
; (CCS Debug -> Debugger -> Loading options: Load symbols only)
; =====
```

In production, this is easier, because it can be assumed that information memory is empty and no erase is required prior to writing to it. The code should be changed to use the original data. Most production programmers such as the GANG430 also supports the "preserve" flash data feature.

```
; =====
; For Mass production please enable cal data readout directly from
; InfoA. Empty devices should be programmed without erasing flash
; before. If erased first, CAL data is lost. But can be restored
; with FlashPro430 / GangPro430 programmers. www.elprotronic.com
mov.b  &CALDCO_1MHZ,&DCOCTL    ; Set DCO step + modulation
mov.b  &CALBC1_1MHZ,&BCSCTL1   ; Set range
; =====
```

### 2.3.1.2 Linker Command File

Another important file of the BSL is the Linker Command File, which defines the location of code and data. It is slightly different between CCS and IAR but the idea is the same: telling the linker where to put the BSL and the application code.

#### 2.3.1.2.1 Locating the Linker Command File

By default, the standard linker command file that comes with the IDE is selected and used. For this BSL project, some modifications of the file are required.

**Table 2. Linker Command File**

CCS	IAR
CCS automatically copies the default linker command file to the project directory. It can be accessed using the CCS Project Explorer.	IAR selects and uses the default linker command file directly out of the installation path. The typical path is: C:\Program Files\IAR Systems\EWB MSP430 5.20\430\config
<b>Typical file name</b>	
lnk_msp430g2001.cmd	lnk430g2001.xcl
<b>Maintaining the original file</b>	
CCS already copied the file to the project	Copy the original file to the project directory to preserve the default file.
<b>Rename file to clearly identify the customized file</b>	
lnk_msp430g2001_bsl.cmd	lnk430g2001_bsl.xcl

#### 2.3.1.2.2 Modify Linker File

For this BSL project, the following modifications needs to be made to the file:

**Table 3. Modify Linker File**

CCS	IAR
Info memory segments are no longer used for data and therefore can be removed.	
Remove or comment (/* and */) four lines in MEMORY section /* INFOA: origin = 0x10C0, length = 0x0040 */ /* INFOB: origin = 0x1080, length = 0x0040 */ /* INFOC: origin = 0x1040, length = 0x0040 */ /* INFOD: origin = 0x1000, length = 0x0040 */	Remove or comment (//) five lines //-Z(CONST)INFO=1000-10FF //-Z(CONST)INFOA=10C0-10FF //-Z(CONST)INFOB=1080-10BF //-Z(CONST)INFOC=1040-107F //-Z(CONST)INFOD=1000-103F
This no longer used memory is assigned to one new memory block labeled BSL and containing code. The following line is added below the just removed or commented lines.	
BSL : origin = 0x1000, length = 0x00FE	-P(CODE)BSL=1000-10FD
<b>Assign content to memory area</b>	
Remove four lines from SECTIONS part: /* MSP430 INFO FLASH MEMORY SEGMENTS */ /* .infoA : {} > INFOA *//* .infoB : {} > INFOB */ /* .infoC : {} > INFOC */ /* .infoD : {} > INFOD */ Add one line below these lines: bsl : {} > BSL /* BSL CODE */	Already done with above statement



### 2.3.1.2.3 Force the IDE to Use Custom Linker File

For the BSL project, the following modifications needs to be made to the file:

**Table 4. Use Custom Linker File**

CCS	IAR
If the linker command file resides in the project path (which is the default), no action is required.	Project → Options → Linker → Config Override default linker configuration file with the just created file. \$PROJ_DIR\$ can be used if the xcl file is stored in the same location than the asm file: \$PROJ_DIR\$\lnk430g2211_bsl.xcl. Otherwise, a full path can be provided to point to the file.

### 2.3.1.3 Project Settings

The project in the IDE itself is another crucial part of the BSL. It needs to be set up to use the correct device and to allow erase and write access to all information memory locations including InfoA.

**Table 5. Project Settings**

CCS	IAR
Specifying the target device	
Project → Properties → CCS Build → General → Device Variant	Project → Options → General Options → Target → Device
Allow access to information memory	
Debug Properties → Target → MSP430 Properties → Download Options → Erase options: Erase main, information, and protected information memory	Project → Options → Debugger → FET Debugger → Download: Allow erase and write access to locked flash memory, and erase main and information memory

## 2.3.2 User Application

For reference purposes, a blink LED example is provided.

This program, as well as any other user application, can be downloaded and debugged without modifying the BSL.

There is no interaction between the BSL and the user application, except for the program start position. This is because the BSL jumps to this position and assumes that it is code. The user application fails to start correctly if, for example, data is stored at this location. The start position depends on the device and programming language (see [Table 6](#)).

**Table 6. User Application**

CCS	IAR
<b>Assembly</b>	
User Application start right after label	
.text	RSEG CODE
Example code	
mainApp_CCS.asm	mainApp_IAR .asm
<b>C</b>	
User Application starts with main() function. Compiler and linker take care of location, as the actual main start position depends on cinit() function.	
Example Code	
mainApp.c	

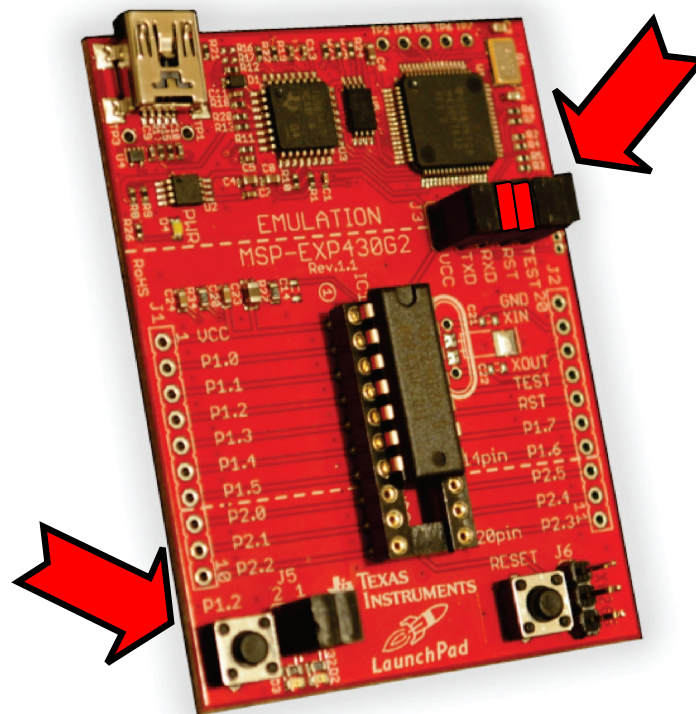
## 2.4 BSL Operation

### 2.4.1 Hardware Setup

For demonstration purposes, it is assumed that the MSP430G2xx is mounted on a LaunchPad™ development kit and the connections shown in [Table 7](#) are made.

**Table 7. BSL Demo Setup**

Pin	Function	LaunchPad Pin
P1.3	BSL_entry	Switch S2
P1.1	UART Rx/D	TXD (Cross connect)
P1.2	UART Tx/D	RXD (Cross connect)
RESET	RESET	RESET
USB	Backchannel UART and power	USB to Host



**Figure 3. Demo Setup**

### 2.4.2 Connection to Host

To run the demo, connect the LaunchPad to the host PC. This power the board, allows downloading of code through Spy-Bi-Wire for the debugging purposes, and (most important for BSL operation) provides a UART interface, as every onboard emulation circuit of the LaunchPad comes with a free Application UART. The Application UART is used to provide a USB-to-UART bridge to the host PC.

**NOTE:** Any other way of providing a UART interface to the device also works, as long as voltage and timing requirements are met.

### 2.4.2.1 Determining COM Port

After the LaunchPad is connected to the host, it needs to be determined which COM port it is assigned to. On Windows® XP systems, this can be determined by using Device Manager. Device Manager can be started by running devmgmt.msc.

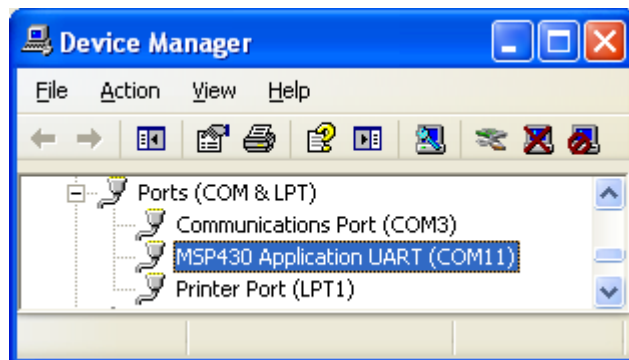


Figure 4. Screenshot of Windows® XP Device Manager

In the section named *Ports (COM & LPT)*, the LaunchPad should show up as MSP430 Application UART. The COM port number is shown to the right of this name. In Figure 4, for example, the port number is COM11.

### 2.4.2.2 Setup of COM Port

As the COM port is a versatile interface, it is important to initialize it to the correct settings:

9600 baud, 8 data bits, 1 stop bit, no parity

The easiest way to do so is by using a terminal program, which is also required for use of this BSL. In this document, a simple yet powerful terminal program called HTerm is used. It can be downloaded from <http://www.der-hammer.info/terminal/hterm.zip> and is free of charge, as of this writing. It is also available for Linux systems.

Select the COM port found in Section 2.4.2.1, set the port values, and connect to the device.

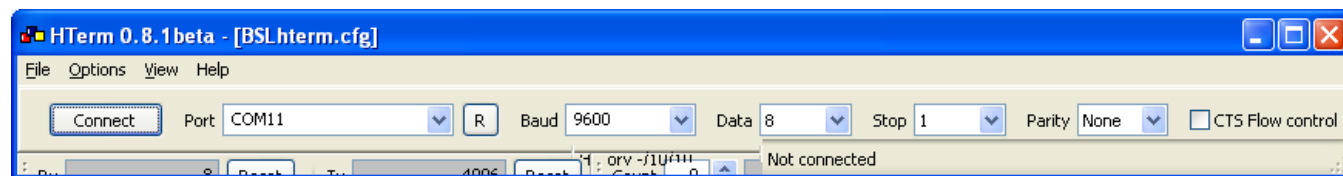


Figure 5. Screenshot HTerm COM Port Configuration

---

**NOTE:** HTerm is one out of many programs that can be used for this purpose. HTerm is only used as an example GUI for this project.

---

### 2.4.3 Operate BSL - Standard Sequence

To flash new user data to the device, the BSL is used in the following sequence.

1. Put the MSP430G2xx into reset state by pressing and holding the reset button on the LaunchPad.
2. Press button S2 (BSL\_entry Pin) on the LaunchPad and release the reset button. The device is now in BSL mode, and S2 can be released.
3. Send the SYNC character from host to device.
4. Wait at least (NumberOfFlashSegments × 16 × 2) ms to allow the device enough time to erase the flash and restore the interrupt vectors.
5. Send the user code byte-wise from host to device.
6. Send the checksum from the host to the device.
7. Read the answer (ACK or NACK) from the device.
8. If the answer is an ACK, the BSL forced a device reset and the device is already in application mode.
9. If the answer is a NACK or there was no response, repeat the procedure until an ACK is received.

In HTerm a "Send File" button can be found to send data that is stored in binary files. The zip archive that is available with this document includes user code (firmware) files to verify the BSL. It also includes a SYNC.bin file to transmit the synchronization character without bothering on number formats.

### 2.4.4 Create New Code to Download Through BSL

#### 2.4.4.1 Create Custom Application

Development of the customer application happens as always by creating specification, coding, debugging, and testing. There is no need to include the BSL in the project at this point.

For convenience, two project demos are available, in C and Assembler, that can be used as a starting point for application development (see [Section 2.3](#) for details).

---

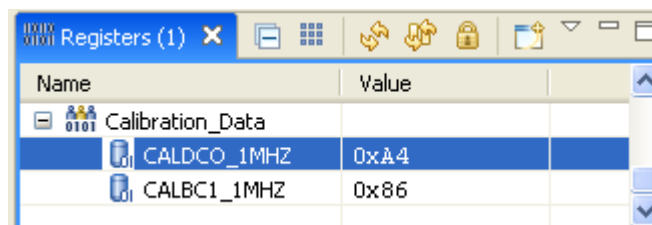
**NOTE:** Information memory cannot be used for user code. It is good practice to uncheck 'erase Info memory' in the debug options, so that this area cannot be used by accident.




---

#### 2.4.4.2 Save Calibration Data

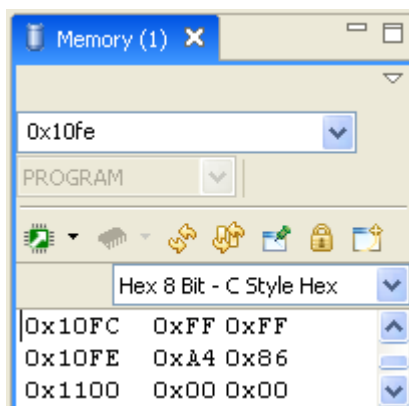
During prototyping, it is important to maintain the 1-MHz DCO calibration data. This is used for stable communication with the host during BSL operation. DCO calibration data is also useful for a defined clock frequency in the user application.

The easiest way to read calibration data is during debugging of a user application that does not overwrite InfoA data. Calibration values can, for example, be retrieved by the methods shown in [Figure 6](#) or [Figure 7](#).



Name	Value
 Calibration_Data	
 CALDCO_1MHZ	0xA4
 CALBC1_1MHZ	0x86

**Figure 6. View → Register → Calibration\_Data (CCS)**



**Figure 7. View → Memory: 0x10FE and 0x10FF (CCS)**

It is important to record these values and restore them in the BSL asm file during debugging. For production, there are other ways of doing so; for example, with the GANG430 production programmer.

#### 2.4.4.3 Make User Application Code a BSL Update File

When the user application is in a state where BSL updates are intended, BSL data can be created.

##### 2.4.4.3.1 Using CCS

The easiest way to get this done is with the help of the IDE. After the application is downloaded to the device, memory can be read back and saved to a file.

In View → Memory a small green IC with an upward arrow allows reading and saving memory (see [Figure 8](#)).



**Figure 8. Save Device Memory Content to a File in CCS**

This file can now be used as input file to the terminal program; for example, HTerm. Note that it does not include the XOR checksum. This is added in the next step.

#### **2.4.4.3.2 Using IAR**

Because IAR can (currently) only save memory locations as TI Text or Intel Hex, after saving the desired memory locations, the output files must then be converted using a conversion utility (such as hex2bin).

#### **2.4.4.4 Obtaining XOR Checksum**

The easiest way to get the correct checksum without any additional program is to use the device in debug mode.

---

**NOTE:** If connected to a debugger, do not use the reset switch on the hardware. Instead use the software reset button of the IDE.

---

##### **2.4.4.4.1 Send User Data**

After a proper entry sequence, all bytes of user code (main flash size minus 2 bytes) are sent from host to device. Those bytes do not include the XOR checksum as this is transmitted in the last byte.

##### **2.4.4.4.2 Read Checksum**

Now the IDE is used to pause the debug session and the already flash checksum that got calculated on the device can be read out. It is located in core register R8, with the symbolic name rCHKSUM.

##### **2.4.4.4.3 Send Acquired Checksum**

Target can now be released again and the value that was extracted from R8 is now be transmitted to the target. This should finish the BSL cycle, and an ACK should be received.

##### **2.4.4.4.4 Verify Data**

To verify if the checksum was correct and the data in the device is correct, the data that was flashed through BSL should be read out (see [Figure 8](#)) and compared against the original flash file.

##### **2.4.4.4.5 Save Checksum**

After comparison is done without errors, the checksum can be appended to the firmware file, allowing an easy download to multiple devices without need to recalculate the checksum for each.

#### **2.4.5 Getting Ready for Production**

For production, it is required to have one valid firmware image to flash to the device.

A simple way to do so is using the text version of the code in the TI-TXT and copy and paste the BSL TI-TXT content into the user application.

Adding user code to the BSL project. This method gives the advantage of being able to debug BSL and user code interaction.

### 3 Frequently Asked Questions (FAQ)

1. **Question: Where can I download the source code and firmware images of the BSL 430?**

Answer: The source code and the firmware images are available from the CUSTOM-BSL430 link at [Bootloader \(BSL\) for MSP low-power microcontrollers](#).

2. **Question: What is the benefit having the Custom BSL programmed in protected memory and overwriting the TI factory BSL rather than just having the Custom BSL in main memory space?**

Answer: When the Custom BSL is programmed in the protected memory, it is secured from erasing and unintended execution. This configuration also supports the invoke sequence, where the bootcode executes the BSL when the invoke sequence is applied. In addition, the custom BSL does not occupy any of the main memory space.

3. **Question: Can the BSL be debugged?**

Answer: Yes. After starting the debug session, set the Program Counter (PC) to point to the BSL entry location (@0x1000).

4. **Question: How to switch from the BSL to an application?**

Answer: The factory BSL (also included in the Custom BSL) provides the SET PC command. This command sets the PC register to the given address to start the application.

5. **Question: What is the default BSL password for empty devices?**

Answer: The devices that come from the factory are erased, and the interrupt vectors are blank (0xFF). This means that the BSL password for initial programming is the default password (all 0xFF).

6. **Question: What does the MASS ERASE BSL command do?**

Answer: The MASS ERASE command erases all banks of the main memory. Information memory segments A to D and the BSL segments A to D are not erased.

7. **Question: If the device is protected using the JTAG Lock, is memory access still possible?**

Answer: Yes, by using BSL when the BSL is unlocked with the correct password.

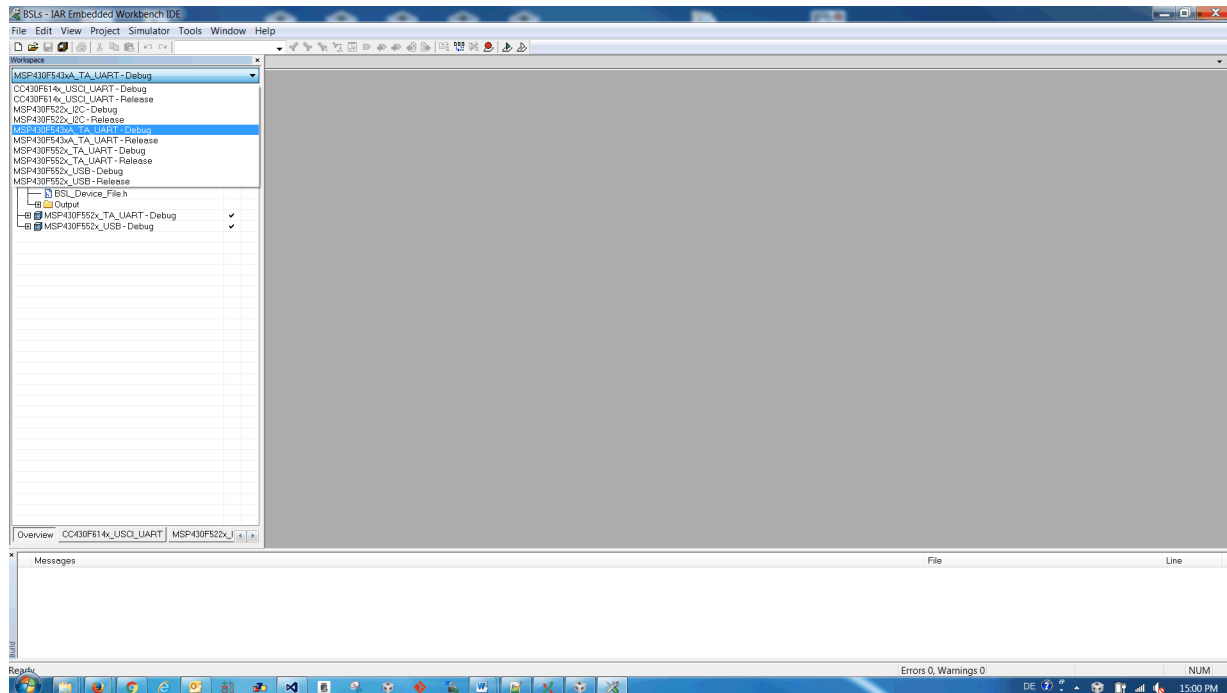
8. **Question: How to recover an overwritten BSL?**

Answer: If the BSL signature is corrupted or the BSL application is not functional, the BSL can still be recovered or reprogrammed using JTAG or SBW, as long as these interfaces are not locked.

## 9. Question: How to debug the BSL example code in IAR?

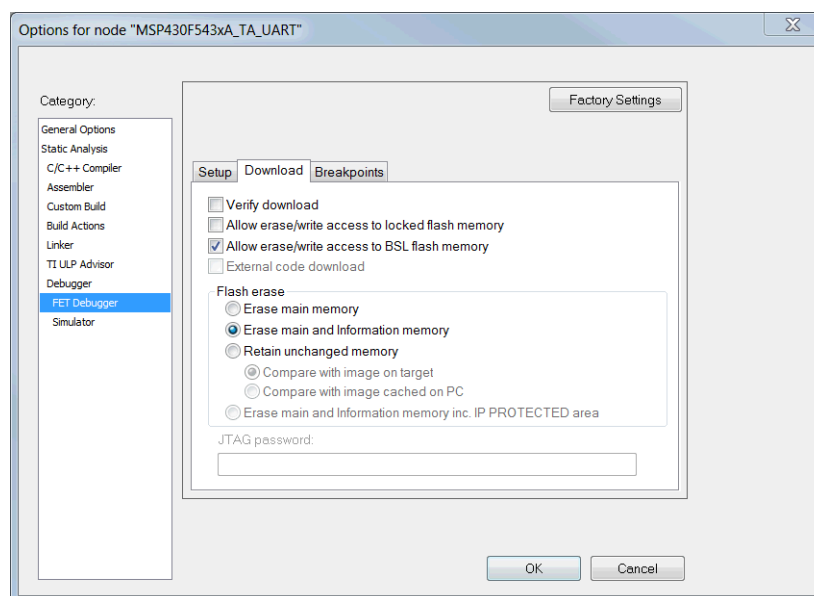
Answer: The IAR project in the Custom BSL430 package was built in IAR 6.30.

- Choose the project to execute. This example demonstrates how to choose the MSP430F543xA\_TA\_UART project.



**Figure 9. Select the Project**

- Check these settings: In the project options, under FET debugger, and the Download tab (see [Figure 10](#)), select the “Allow erase/write access to the BSL flash memory” and “Erase main and information memory”.

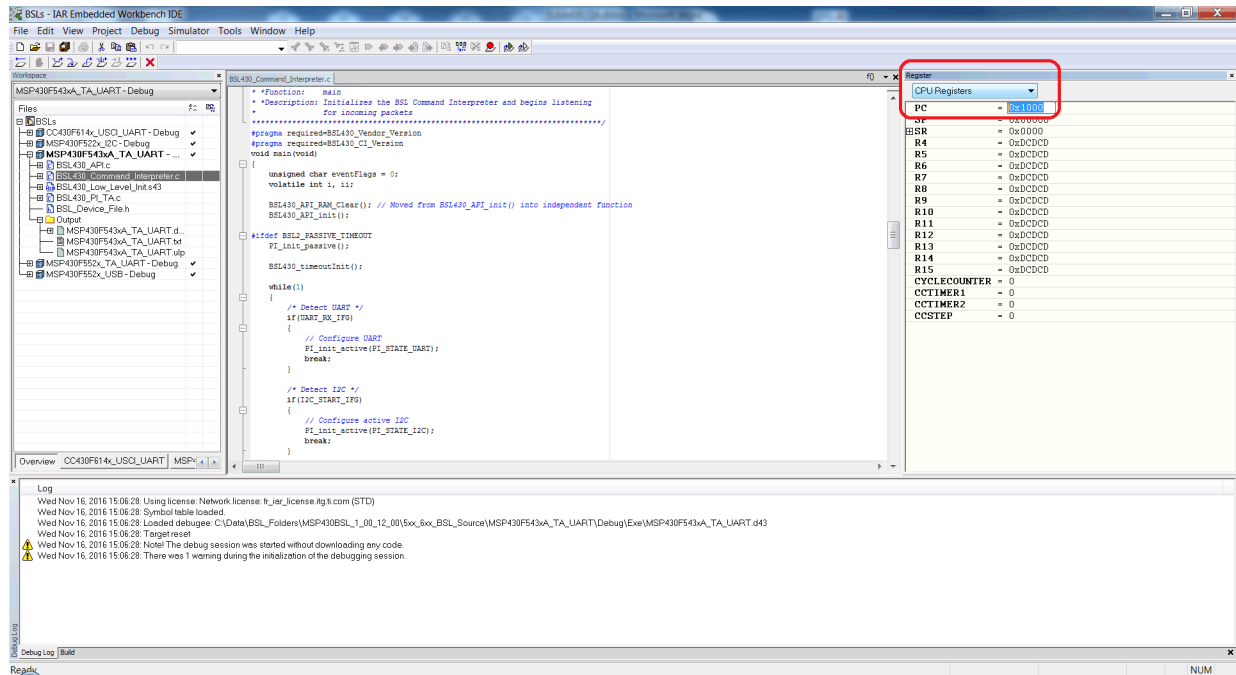


**Figure 10. Setting in Project Properties**

- After compiling the project, start the debug session.

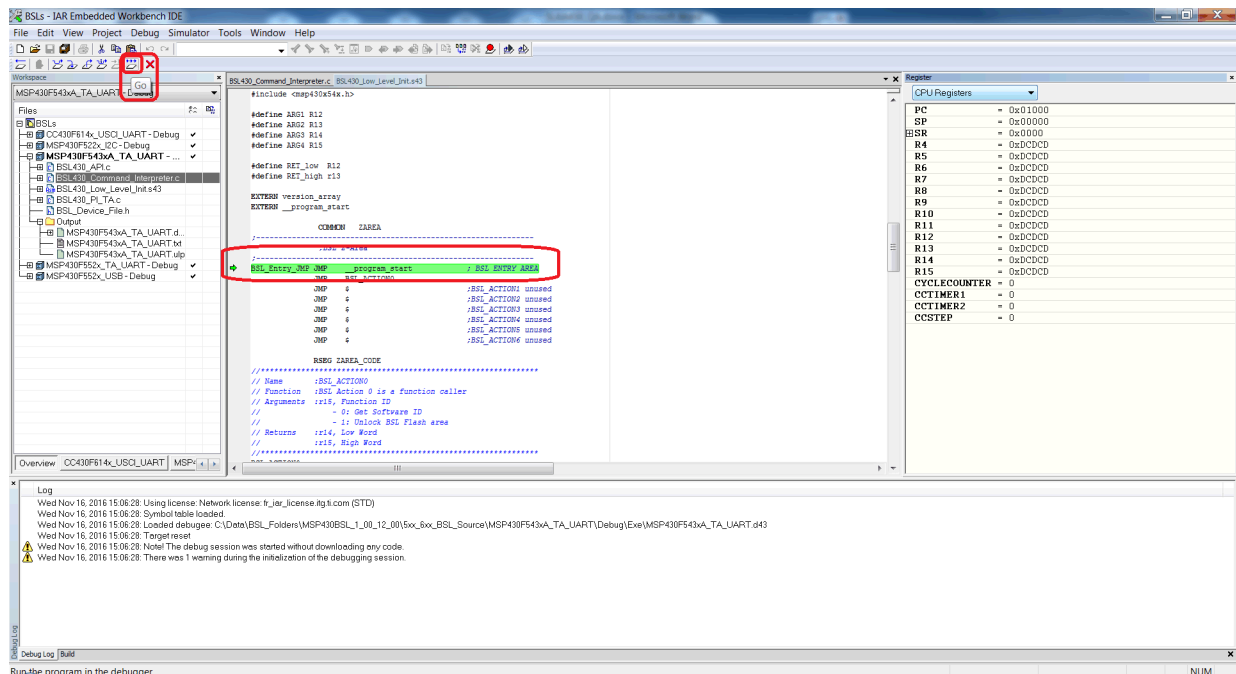


- d. Open the Register View, and manually set the PC to 0x1000, which is the entry location of the BSL (see Figure 11).



**Figure 11. Set the PC to the Location of BSL**

- e. Run the BSL in the debugger. The program starts on the BSL entry function (see Figure 12).

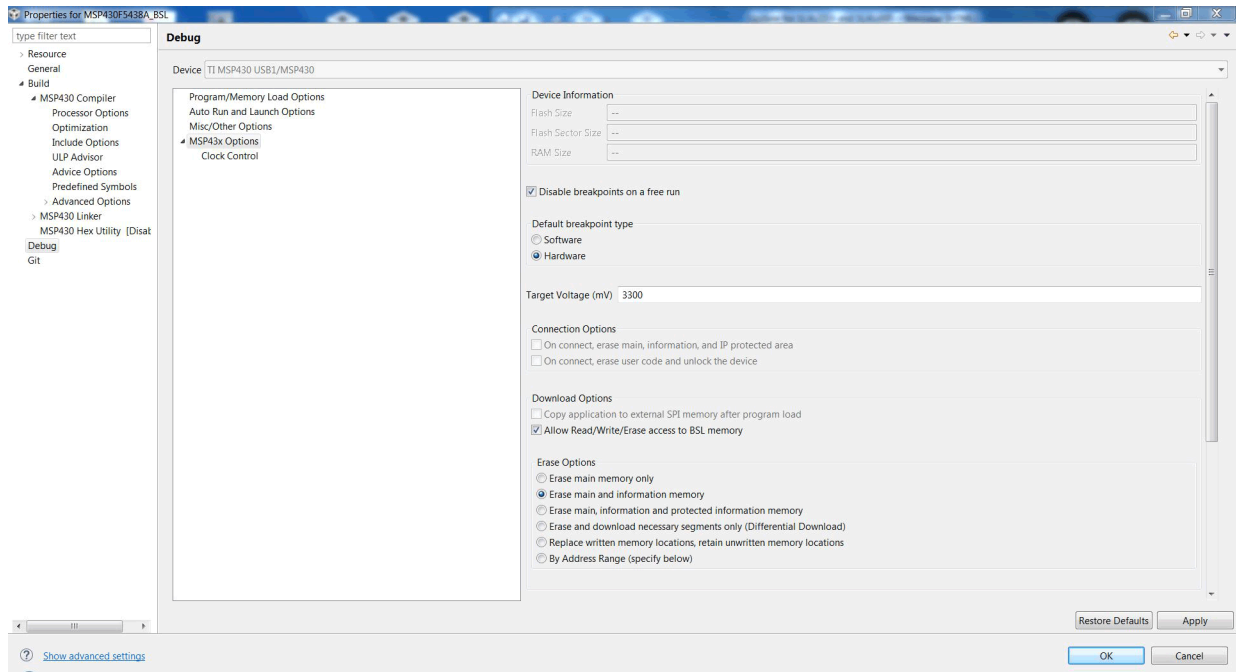


**Figure 12. Run the Debug Session**

## 10. Question: How to debug the BSL example code in CCS?

Answer: MSP430 Custom BSL 1.1.0 provides the CCS example that can be easily imported to CCS.

- After the project is imported, go to the Project Properties, and configure the programming setup (see [Figure 13](#)).

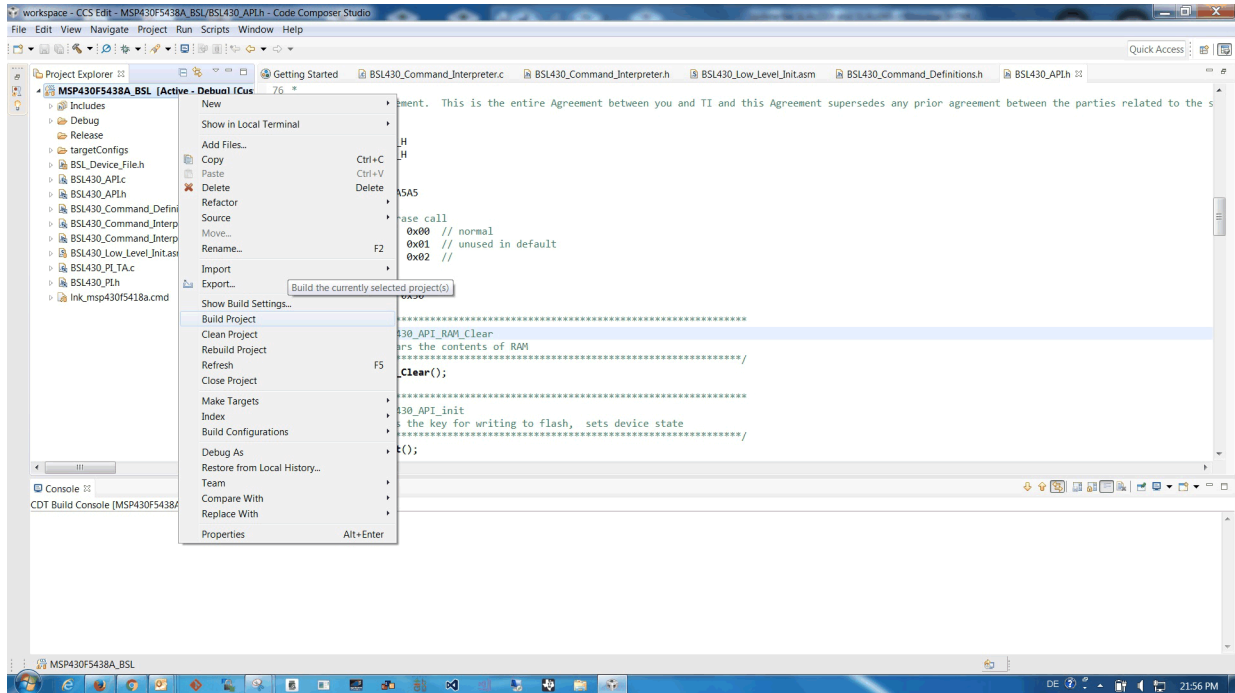


**Figure 13. Debug Setup**

- As [Figure 13](#) shows, select Debug → MSP43x Options.  
Under "Download Options", select "Allow Read/Write/Erase access to BSL memory".  
Under "Erase Options", select "Erase main and information memory".

- c. Build the project (see [Figure 14](#)).

**NOTE:** Due to current compiler restraints in CCS, some Custom BSL combinations will not compile under 2KB memory. If the application must have the BSL completely in its 2KB memory region, use IAR to compile the custom BSL.



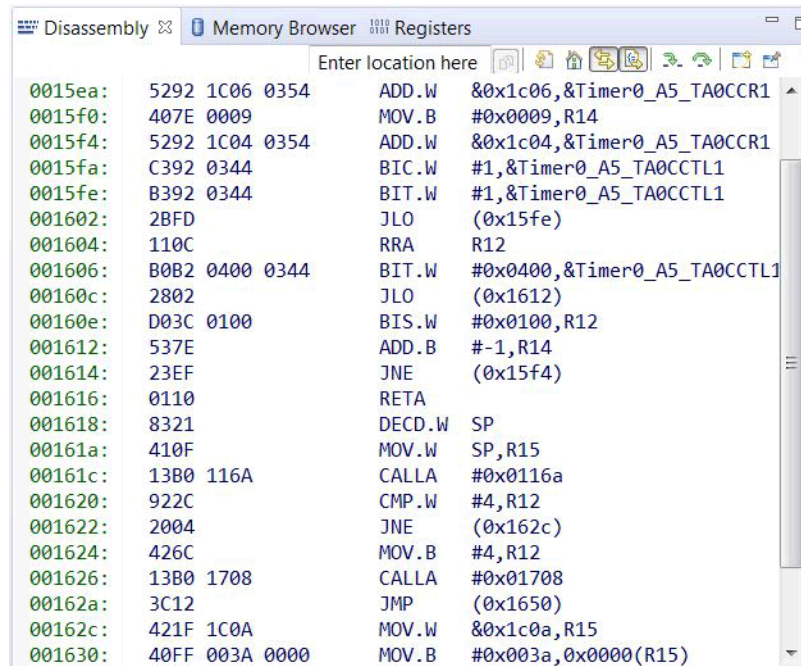
**Figure 14. Build Project**

- d. Run the debug mode, pause the session, and then open the "Registers" view.  
e. Set the PC to 0x1000, which is the entry location of the BSL (see [Figure 15](#)).

Disassembly Memory Browser Registers	
Name	Value
Core Registers	
PC	0x001000
SP	0x005BF0
SR	0x0002
R3	0x000000
R4	0x0FFFFFFF
R5	0x0017F0
R6	0x000008
R7	0x0FFFD
R8	0x00A508
R9	0x000000
R10	0x00FFFF
R11	0x0FFFFB
R12	0x000000
R13	0x00FFFF
R14	0x005BFC
R15	0x000000
ADC12	
Backup_RAM	
Battery_Charger	

**Figure 15. Set PC to BSL Address**

- f. Continue the debug mode.
- g. When the debug mode is paused, the "Disassembly" view shows that the BSL is executed (see [Figure 16](#)).
- h. Add a breakpoint to debug the source code.



The screenshot shows the Disassembly window of a debugger. The window has tabs for Disassembly, Memory Browser, and Registers. The Disassembly tab is active, showing a list of assembly instructions with their addresses, hex values, and mnemonics. The instructions are as follows:

Address	Hex Value	Mnemonic	Comment
0015ea:	5292 1C06 0354	ADD.W	&0x1c06,&Timer0_A5_TA0CCR1
0015f0:	407E 0009	MOV.B	#0x0009,R14
0015f4:	5292 1C04 0354	ADD.W	&0x1c04,&Timer0_A5_TA0CCR1
0015fa:	C392 0344	BIC.W	#1,&Timer0_A5_TA0CCTL1
0015fe:	B392 0344	BIT.W	#1,&Timer0_A5_TA0CCTL1
001602:	2BFD	JLO	(0x15fe)
001604:	110C	RRA	R12
001606:	B0B2 0400 0344	BIT.W	#0x0400,&Timer0_A5_TA0CCTL1
00160c:	2802	JLO	(0x1612)
00160e:	D03C 0100	BIS.W	#0x0100,R12
001612:	537E	ADD.B	#-1,R14
001614:	23EF	JNE	(0x15f4)
001616:	0110	RETA	
001618:	8321	DECD.W	SP
00161a:	410F	MOV.W	SP,R15
00161c:	13B0 116A	CALLA	#0x0116a
001620:	922C	CMP.W	#4,R12
001622:	2004	JNE	(0x162c)
001624:	426C	MOV.B	#4,R12
001626:	13B0 1708	CALLA	#0x01708
00162a:	3C12	JMP	(0x1650)
00162c:	421F 1C0A	MOV.W	&0x1c0a,R15
001630:	40FF 003A 0000	MOV.B	#0x003a,0x0000(R15)

**Figure 16. BSL is Executed**

## Revision History

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

<b>Changes from July 25, 2018 to April 14, 2020</b>	<b>Page</b>
• Added information about CCS in <a href="#">Section 1.3 TI-Supplied BSL Software</a> .....	<a href="#">6</a>
• Changed the title of <a href="#">Section 1.3.2.1</a> .....	<a href="#">6</a>
• Deleted note from <a href="#">Section 1.3.2.2</a> .....	<a href="#">7</a>
• Changed the title of <a href="#">Section 1.3.2.3</a> .....	<a href="#">7</a>
• Added <a href="#">Section 1.3.3</a> .....	<a href="#">8</a>
• Moved Section 1.3.2.4 to <a href="#">Section 1.3.3.4</a> .....	<a href="#">10</a>

## IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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, 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 ([www.ti.com/legal/termsofsale.html](http://www.ti.com/legal/termsofsale.html)) or other applicable terms available either on [ti.com](http://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.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265  
Copyright © 2020, Texas Instruments Incorporated