Application Report

SLAA450B–April 2010–Revised May 2012

Creating a Custom Flash-Based Bootstrap Loader (BSL)

Lane Westlund MSP430 Tools

ABSTRACT

MSP430F5xx and MSP430F6xx devices have the ability to locate their bootstrap loader (BSL) in a protected location of flash memory. Although all devices ship with a standard TI BSL, this can be erased, and a custom made BSL can be programmed in its place. This allows for the creation of using custom communication interfaces, startup sequences, and other possibilities. It is the goal of this document to describe the basics of the BSL memory, as well as describe the TI standard BSL software so it may be reused in custom projects.

This application report also includes 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.

Project collateral and associated source discussed in this application report can be downloaded from http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/CustomBSL/latest/index_FDS.html.

Contents

1 5xx and 6xx Bootstrap Loader Customization 2
2 G2xx Bootstrap Loader Creation and Customization 9

List of Figures

1 Device Startup Sequence 3
2 Functional Block Diagram, MSP430G2x01 9
3 Demo Setup 14
4 Screenshot of Windows® XP Device Manager 15
5 Screenshot HTerm COM Port Configuration 15
6 View -> Register -> Calibration_Data (CCS) 16
7 View -> Memory: 0x10fe and 0x10ff (CCS) 17
8 Save Device Memory Content to a File in CCS 17

List of Tables

1 Pin Assignment 11
2 Linker Command File 12
3 Modify Linker File 12
4 Use Custom Linker File 13
5 Project Settings 13
6 User Application 13
7 BSL Demo Setup 14

Copyright © 2010–2012, Texas Instruments Incorporated

1 5xx and 6xx Bootstrap Loader Customization

1.1 BSL Memory Layout

The BSL memory is a 2-kilobyte 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, described in the *MSP430F5xx and MSP430x6xx Family User's Guide* (SLAU208).

1.1.1 Z-Area

When protected, the BSL memory cannot be read or jumped into from a location external to BSL memory. This is done to make the BSL more secure against erase and to also prevent 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 startup 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. It 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* (SLAU208) for more details about the JTAG Key.

0x17FA

BSL Start Vector. This is the address of the first instruction to be executed on BSL invoke.

0x17F8

Reserved

0x17F6

BSL Unlock Signature 1: This word should be set to 0xC35A to indicate a correctly programmed BSL. If not, BSL will not start.

0x17F4

BSL Unlock Signature 2: This word should be set to 0x3CA5 to indicate a correctly programmed BSL. If not, 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 later in this document.

Copyright © 2010–2012, Texas Instruments Incorporated

1.2 Device Startup Sequence

After power up, the device first checks the BSL signature. If the appropriate values are there, 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 startup sequence occurs according to the flow chart shown in Figure 1.

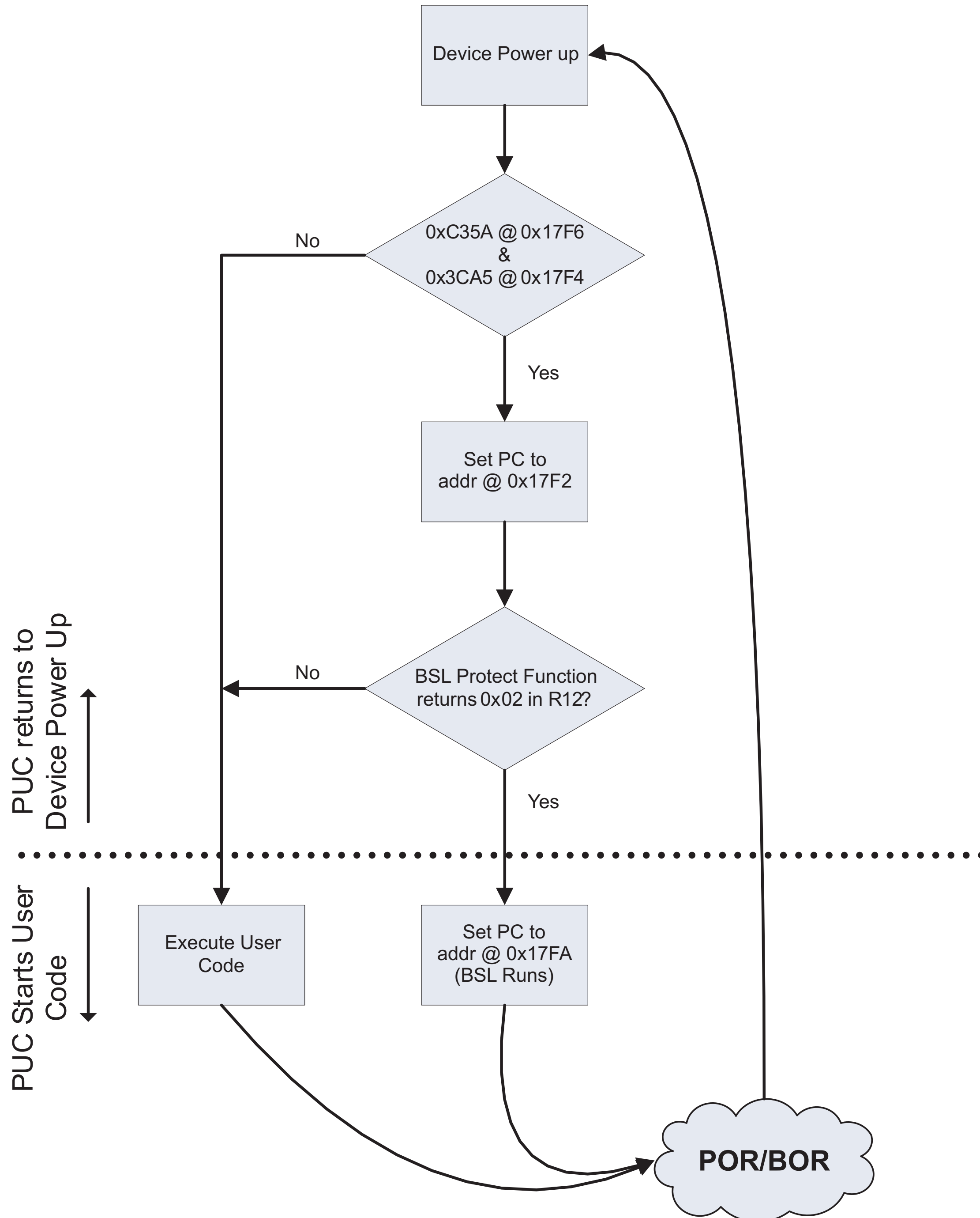

Figure 1. Device Startup Sequence

Copyright © 2010–2012, Texas Instruments Incorporated

1.2.1 BSL Protect Function

The BSL Protect function is a function that 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, it renders the device totally unresponsive. Additionally, 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 functions: protecting the BSL memory and determining whether the BSL or user code should be executed after exiting the BSL Protect function.

1.2.1.1 Protection of BSL Memory

This is done by setting the size and protection bits in the SYSBLSC register. This is performed regardless of whether a BSL invoke is to be requested For more information on these specific bits, see the *MSP430F5xx and MSP430x6xx Family User's Guide* (SLAU208).

1.2.1.2 Checking for BSL Invoke

The BSL_Protect function also must indicate whether the BSL should start or application code should start (if present). It does this 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 JTAG Key, keeping JTAG open (used primarily for debugging 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.

It should be noted that 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 startup criteria based on their application requirements: If USB power is present and the device is blank, the BSL_Protect function indicates the BSL should start, so the blank device may be programmed via USB. However, in a custom BSL, almost any imaginable criteria could be used, such as checking the user code against a known CRC value to ensure only correctly programmed user code begins execution.

Copyright © 2010–2012, Texas Instruments Incorporated

1.3 TI Supplied BSL Software

A pre-programmed BSL is supplied with each MSP430x5xx and MSP430x6xx device. Use of this BSL is described in *MSP430 Programming Via the Bootstrap Loader (BSL)* (SLAU319). 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.

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, it can use any hardware or protocol (wrapper bytes, etc). The actual transmission mechanism and protocol are irrelevant for the rest of the BSL. What is important is that when it is called upon to receive a packet, it does not return until it knows it has correctly received all data sent to it. Since 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. It can assume 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 SLAU319) 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 cover all functions in detail (parameters, return values, etc). This information is contained in the function comments in the header/source files. The goal of this section is to highlight important responsibilities of individual files which might not be immediately obvious by reading the source.

1.3.2.1 BSL430_Low_Level_Init.s43

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 only customization required here will be limited to:

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

Copyright © 2010–2012, Texas Instruments Incorporated

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, etc. In addition, there are custom BSL definitions 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 which should be overwritten at BSL start, for security. This is usually the lowest RAM value. The RAM will be cleared between this address and the stack pointer.

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

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 2-kB 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 which 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 startup 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 lnk430FXXXX_BSL_AREA.xcl

This is a custom linker command file, which places the project code output into the BSL memory locations. Additionally, 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 placed intentionally 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.

Copyright © 2010–2012, Texas Instruments Incorporated

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 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 knows how many bytes to send.

Copyright © 2010–2012, Texas Instruments Incorporated

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.

The following IAR versions were used to build the device BSLs:

MSP430F5438A and CC430F6137: IAR 5.3 KickStart with IAR C/C++ compiler 4.20.1

MSP430F5529: IAR 4.11C KickStart with IAR C/C++ compiler 4.11B

1.5.3 USB BSL External Oscillator Frequency

The USB BSL uses a routine to measure the speed of the external oscillator used in the application. It 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 to 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 illustrated in this code example, where only one known frequency is used, it is important the measurement loop remain in the USB Peripheral Interface as it is also used for a delay to allow for crystal startup.

Copyright © 2010–2012, Texas Instruments Incorporated

2 G2xx Bootstrap Loader Creation and Customization

2.1 Target System Specification

This bootstrap loader 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 as well – required modifications are shown in this document.

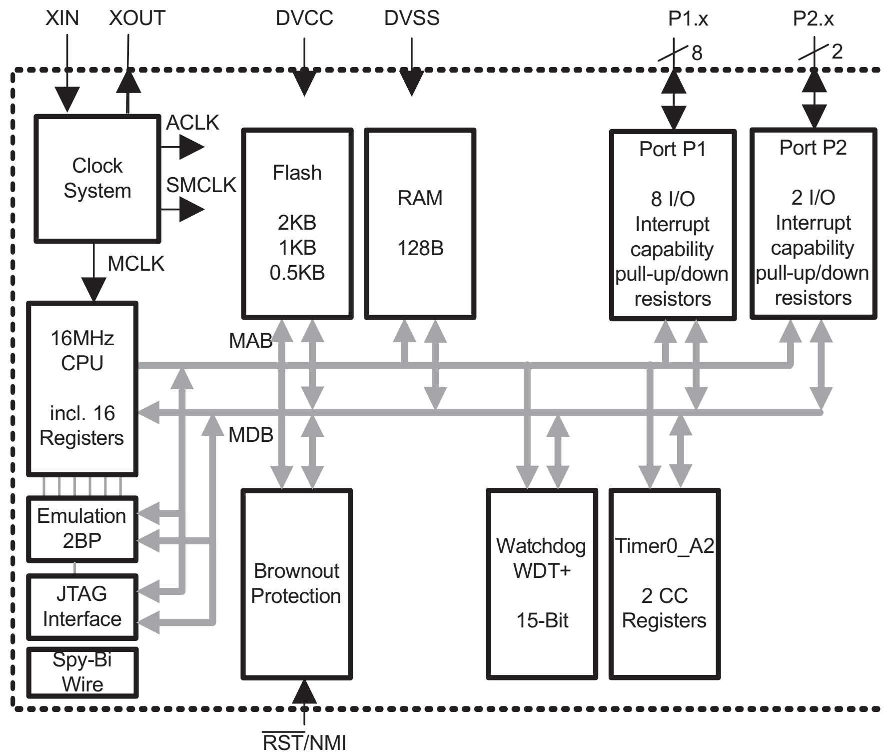

Figure 2. Functional Block Diagram, MSP430G2x01

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. On top of that the available resources on the device greatly influenced the way this BSL is working.

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.

Copyright © 2010–2012, Texas Instruments Incorporated

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 only possible to run the code in exactly this order.

2.2.1.1 Entry Sequence

Entering the BSL is only possible 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. So BSL needs to take immediate action to restore the reset vector pointing 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 total available main flash size minus 2 bytes, starting at 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. It returns a NACK (0xFE) if the checksums do not match.

2.2.2 Memory Footprint

The biggest challenge in implementing a bootstrap loader 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.

Copyright © 2010–2012, Texas Instruments Incorporated

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 via 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 DCO calibration data which 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, as 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
; ================================================================
```


Copyright © 2010–2012, Texas Instruments Incorporated

2.3.1.2 Linker Command File

Another important file of the BSL is the Linker Command File defining 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 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
Typical file name	
lnk_msp430g2001.cmd	lnk430g2001.xcl
Maintaining the original file	
CCS already copied the file to the project	The original file should be copied to project directory to preserve the default file.
Rename file to clearly identify the customized file	
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
Assign content to memory area	
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

Copyright © 2010–2012, Texas Instruments Incorporated

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
Assembly	
User Application start right after label	
.text	RSEG CODE
Example code	
mainApp_CCS.asm	mainApp_IAR .asm
C	
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	

Copyright © 2010–2012, Texas Instruments Incorporated

2.4 BSL Operation

2.4.1 Hardware Setup

For demonstration purposes it is assumed that the MSP430G2xx is mounted on a LaunchPad platform 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 RxD	TXD (Cross connect)
P1.2	UART TxD	RXD (Cross connect)
RESET	RESET	RESET
USB	Backchannel UART and power	USB to Host

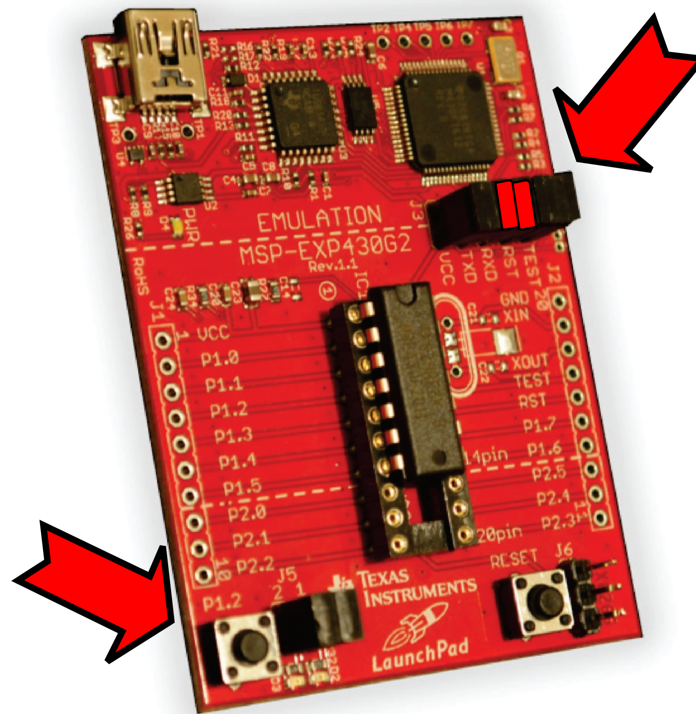

Figure 3. Demo Setup

2.4.2 Connection to Host

To run the demo, it is required to connect the LaunchPad to the Host PC. This power the board, allows downloading of code via 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.

Copyright © 2010–2012, Texas Instruments Incorporated

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 the Device Manager. Device Manager can be started by running devmgmt.msc.

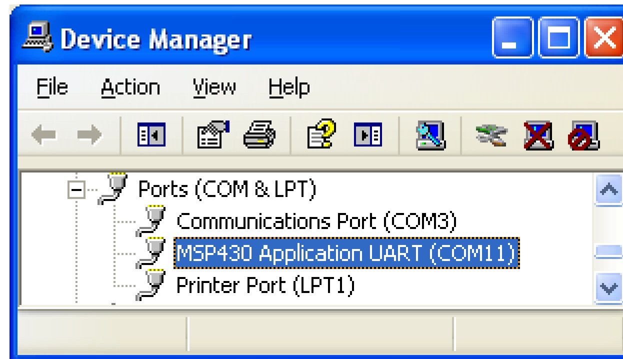

Figure 4. Screenshot of Windows® XP Device Manager

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 the screenshot in Figure 4, for example, it 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.

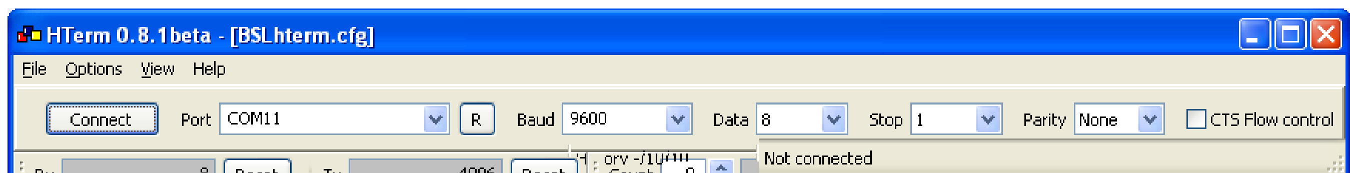

Figure 5. Screenshot HTerm COM Port Configuration

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

Copyright © 2010–2012, Texas Instruments Incorporated

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 via 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.

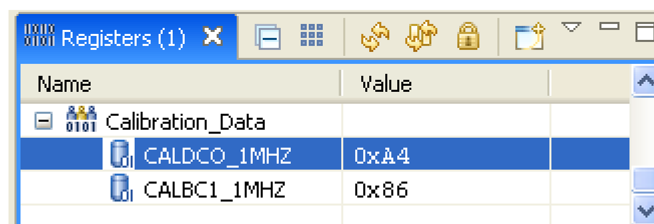

Figure 6. View -> Register -> Calibration_Data (CCS)

Copyright © 2010–2012, Texas Instruments Incorporated

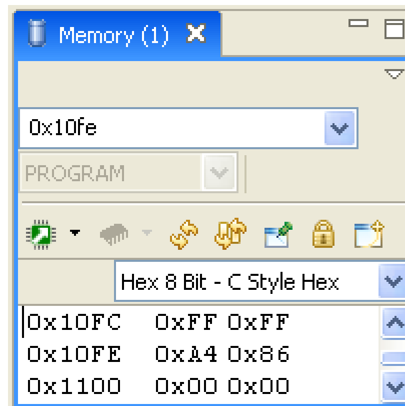

Figure 7. View -> Memory: 0x10fe and 0x10ff (CCS)

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 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

Copyright © 2010–2012, Texas Instruments Incorporated

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 via 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.

Copyright © 2010–2012, Texas Instruments Incorporated

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information 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.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
		TI E2E Community Home Page	e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2012, Texas Instruments Incorporated