Application Report
SLAA450A–April 2010–Revised July 2011

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.

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

List of Figures

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 Family User's Guide* (SLAU208).

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.

1.2 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 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.

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 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.

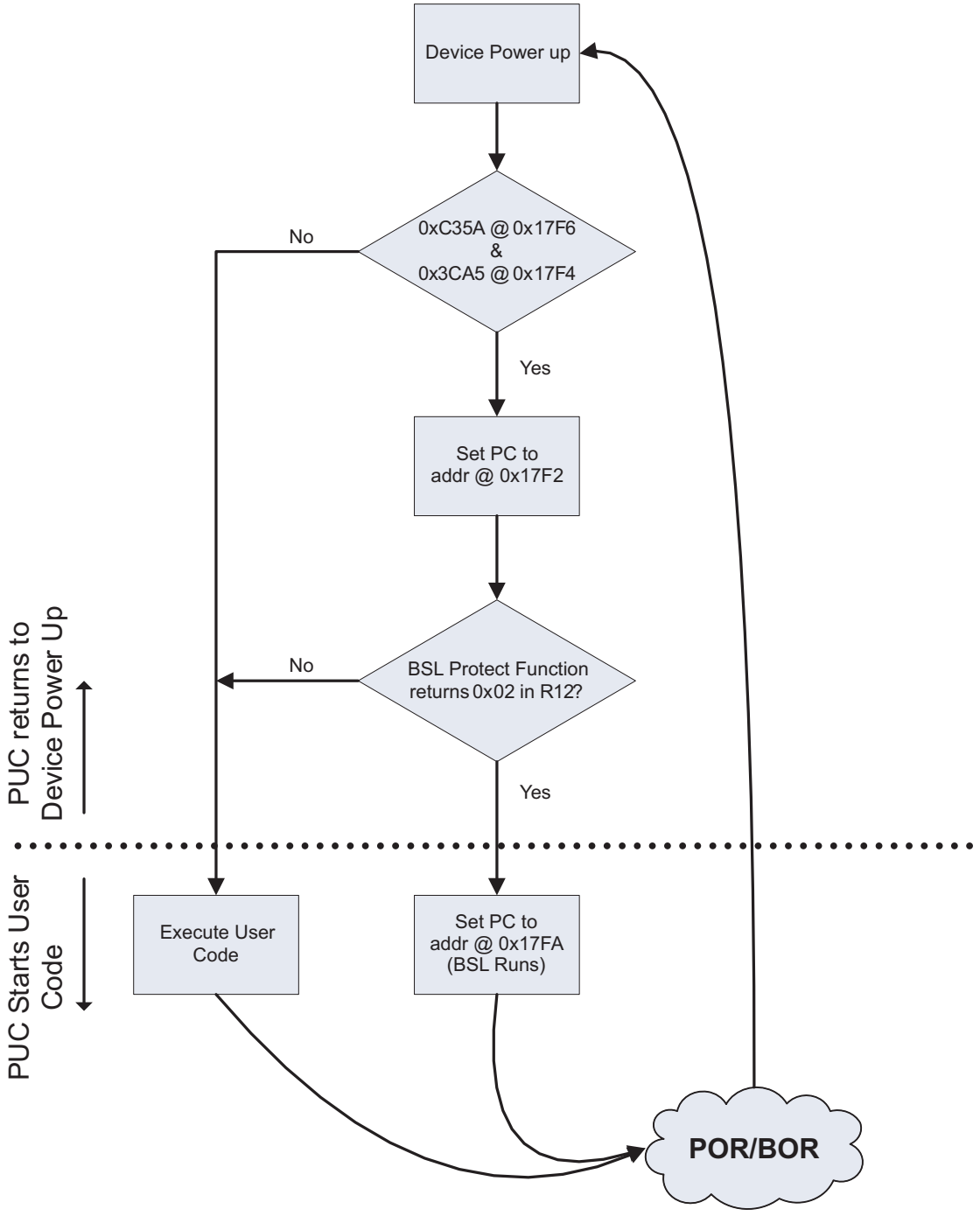

Figure 1. Device Startup Sequence

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.

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 Family User's Guide* (SLAU208).

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.

3 TI Supplied BSL Software

Supplied with each MSP430 5xx/6xx device is a pre-programmed BSL. Usage of this BSL is described in the *Memory Programmers User's Guide* (SLAU265). Familiarity with the BSL at this level is assumed for the following section. Additionally, the TI supplied BSLs were written for and compiled with IAR KickStart.

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 SLAU265) 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.

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.

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.

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.

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.

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.

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.

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.

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.

5 BSL Development and Debug

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.

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

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.

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

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265