TEXAS INSTRUMENTS

Application Report

SLAA096D–September 2005–Revised August 2006

Application of Bootstrap Loader in MSP430 With Flash Hardware and Software Proposal

Volker Rzehak and Stefan Schauer

MSP430

ABSTRACT

The bootstrap loader (sometimes called the bootloader) of MSP430 derivatives with flash memory allows access to their embedded memories during prototyping, production, and in the field. It is possible to download or modify code in flash memory (electrically-erasable and programmable memory) or to store calibration data or other system-relevant data in flash memory or in RAM.

This application report describes simple and low-cost hardware and software solutions to access the bootstrap loader functions of the MSP430 flash devices via the serial port (RS-232) of a PC. The description provided and the C-source code of the software routines allow adaptation of the software to specific requirements.

Contents

1 Introduction 2
2 Bootstrap Loader Basics 2
3 Hardware Description 3
4 Software Description 7
5 References 17
Appendix A Software 18
Appendix B PCB Layout Suggestion 19
Appendix C Demonstration Program Usage 22
Appendix D Errata 24
Appendix E Third-Party Support 25

List of Figures

1 Bootstrap Loader Interface Schematic 3
B-1 Universal BSL Interface PCB Layout—Top 19
B-2 Universal BSL Interface PCB Layout—Bottom 19
B-3 Universal BSL Interface Component Placement 20
B-4 Universal BSL Interface Component Placement 21

List of Tables

1 Bootstrap Loader UART Settings 2
2 Serial-Port Signals and Pin Assignments 4
3 RS-232 Levels 4
4 Pin Assignment of Target Connector 5
5 Universal BSL Interface Parts List 6
6 Bootstrap Loader Access Functions 10
C-1 Command-Line Parameters 22
C-2 Program-Flow Modifiers 22
C-3 Invocation Examples 23

All trademarks are the property of their respective owners.

1 Introduction

The MSP430 derivatives with flash memory (electrically-erasable and programmable memory) allow modification of data and program code in a matter of seconds. Erasure cycles using UV light to clear the EPROM are no longer required. A control unit is required to access and use these features. Part of this control unit is the bootstrap loader, implemented in MSP430 devices with flash memory. This report describes hardware and software solutions to access the loader from a PC for control and simple use of the bootstrap loader.

Note: For additional information, refer to the MSP430 application report, *Features of the MSP430 Bootstrap Loader*, literature number SLAA089.

2 Bootstrap Loader Basics

The bootstrap loader is a program that allows communication with the MSP430 via a serial link, even when the flash memory is completely erased. Do not confuse the bootstrap loader with programs found in some digital signal processors (DSPs) that automatically load program code (and data) from external memory to the internal memory of the DSP. These programs are often referred to as bootstrap loaders as well.

The MSP430 bootstrap loader has protected and unprotected functions. To enable the protected functions, the fully-programmed interrupt-vector table (located in address range 0FFE0h to 0FFFFh) must be sent to the bootstrap loader. The commands and their usage are described in more detail in *Features of the MSP430 Bootstrap Loader* (literature number SLAA089), in the software available along with this application report, and in the device-specific data sheets for the flash-based MSP430 devices. Note that currently it is not possible to blow the JTAG security fuse using the bootstrap loader. If the JTAG fuse is blown, it is still possible to use the bootstrap loader. Access to the code via the bootstrap loader is password protected.

2.1 Access to the Bootstrap Loader

After invoking the bootstrap loader via the RST/NMI and TEST or TCK pins, communication can be established using a standard asynchronous serial protocol. The UART settings are shown in Table 1.

MSP430 port pins are used to transmit and receive data. Usually port pins shared with TA0 are used (e.g., P1.1 for transmit and P2.2 for receive on F1xx derivatives; P1.0 for transmit and P1.1 for receive on F4xx derivatives). Consult the data sheet for the appropriate pinning information.

Table 1. Bootstrap Loader UART Settings

SETTING	VALUE
Baud rate	9600 baud
Data bits	8 (binary)
Parity	Even
Stop bits	1

Note: Timing limitations: Between receiving the ACK character from the MSP430 and the start of the next transmission for the PC side, a delay of 1.2 ms should be ensured. See *Features of the MSP430 Bootstrap Loader*, literature number SLAA089, for more information.

3 Hardware Description

The low-cost hardware presented in this application report (see Figure 1) consists mainly of a low-dropout voltage regulator, some inverters, and operational amplifiers. There are also some resistors, capacitors, and diodes. A complete parts list is provided later in this section.

The functional blocks are described in more detail in the following subsections.

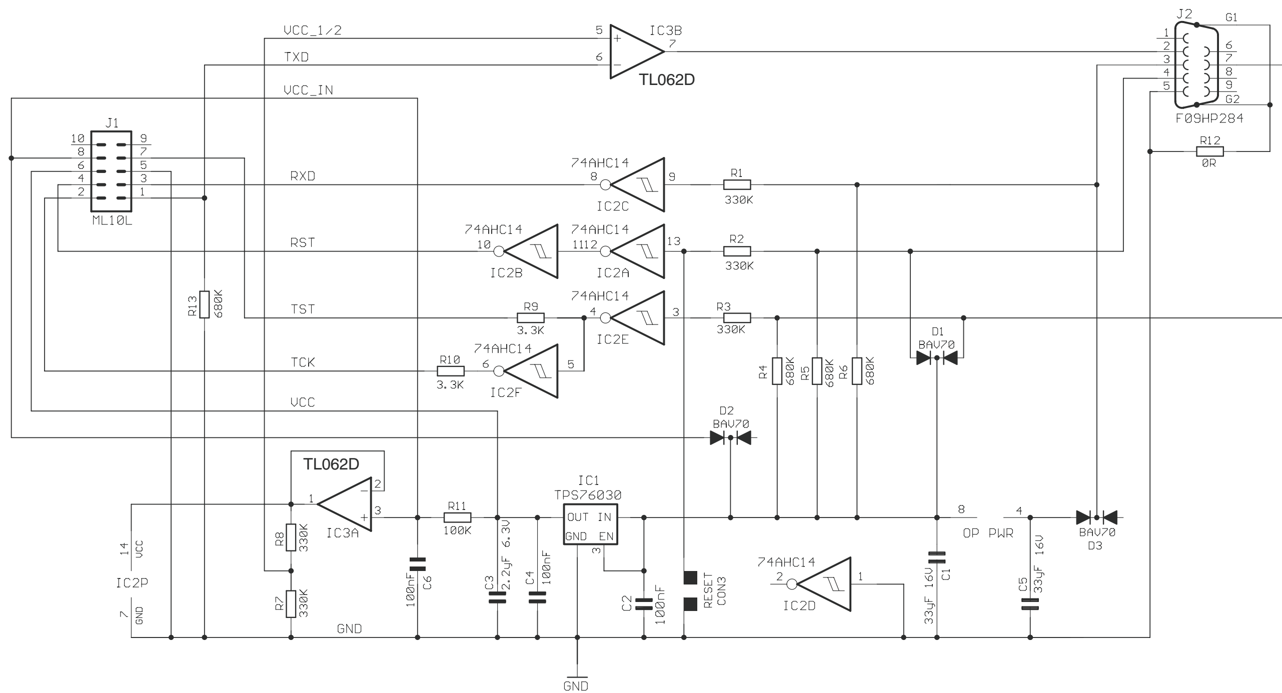

Figure 1. Bootstrap Loader Interface Schematic

3.1 Power Supply

Power for the bootstrap loader hardware can be supplied via the RS-232 interface. RS-232 signals DTR (pin 7 of the serial connector) and RTS (pin 4 of the serial connector) normally deliver a positive voltage to load capacitor C1 and power to the low-dropout voltage regulator IC1 (TI TPS76030 or LP2980-3.0, or equivalent 3-V low-dropout regulator).

Using a fairly big capacitor, it is possible to draw a short-duration current that is higher than the driving serial port can supply. This feature is required to program the flash memory, for example.

It is also possible to connect an external supply voltage to the hardware via pin 8 of the BSL target connector (J1). Diodes are used to prevent reverse-polarity flow.

3.2 Serial Interface

Table 2 shows the signals used to communicate with the bootstrap loader (via connector J2). The names refer to the pin function as seen from the PC. For example, the PC receives data via the RxD pin, whereas the bootstrap loader needs to drive this signal.

Table 2. Serial-Port Signals and Pin Assignments

PIN NAME	FULL NAME (PC)	9-PIN SUB-D	FUNCTION ON BSL INTERFACE
RxD	Receive data	2	Transmit data to PC
TxD	Transmit data	3	Receive data from PC (and negative supply)
DTR	Data terminal ready	4	Reset control (and positive supply)
RTS	Request to send	7	TEST or TCK control (and positive supply)
GND	Ground	5	Ground

3.2.1 Level Shifting

Simple CMOS inverters with Schmitt-trigger characteristics (IC2) are used to transform the RS-232 levels (see Table 3) to CMOS levels.

Table 3. RS-232 Levels

LOGIC LEVEL	RS-232 LEVEL	RS-232 VOLTAGE LEVEL
1	Mark	–3 V to 15 V
0	Space	3 V to 15 V

The inverters are powered via the operational amplifier IC3A. This amplifier permits adjusting the provided logic level to the requirements of the connected target application. A voltage applied to pin 8 of the BSL target connector (VCC_IN) overrides the default 3-V level provided via IC1 and the 100-kΩ series resistor R11. Thus, the output voltage of the operational amplifier is pulled to the applied voltage VCC_IN.

Depending on the overvoltage protection of the device family selected, the excess voltage is either conducted to V_{CC} (as in the TI 74HC14) or to GND (as in the TI 74AHC14). If the protection diode conducts to V_{CC}, the operational amplifier IC3A needs to compensate for the overvoltage. Therefore, the 74AHC14 device, which conducts to ground (GND), is recommended.

To avoid excessive power dissipation and damage of the protection diodes, series resistors (R1, R2, and R3) are used to limit the input current.

An operational amplifier (IC3B) is used to generate RS-232 levels out of CMOS levels. The level at the positive input is set to $V_{CC}/2$ (1.5 V nominal). If the level at the negative input rises above this level, the output is pulled to the negative supply of the operational amplifier (mark). If the level drops below $V_{CC}/2$, the output is pulled to the positive rail (space).

The positive supply of the operational amplifier is the same as the input to the voltage regulator. A separate capacitor (C5) is used to generate the negative-supply voltage. This capacitor is charged via the receiving signal of the bootstrap loader hardware (pin 3 on SUB-D connector J2).

During an asynchronous serial communication, the combination of stop bit and start bit is used to synchronize sender and receiver. After the transmission of a data byte, the stop bit forces the transmission line into a defined state, which is usually a logic 1 or, in RS-232 terms, a mark. This means that the transmission-line voltage is negative when there is no transmission and the capacitor can be charged. Diodes are used to prevent discharge of the capacitor during transmission.

In very rare circumstances, the data sent to the bootstrap loader interface might hold too many zeros, so that the capacitor C5 required for the negative supply is discharged, causing a malfunction of the interface. (A possible work around is to send the respective data in smaller chunks.) But under normal operating conditions, even data containing all zeros does not cause problems.

3.2.2 Control of RST/NMI and TEST or TCK Pins

The two pins used to invoke the bootstrap loader software of the MSP430—RST/NMI and TEST or TCK (for devices without a dedicated TEST pin)—are controlled via the DTR and RTS signals, respectively. These signals also deliver a positive voltage to supply the bootstrap loader hardware.

For devices with a dedicated TEST pin, the levels at RST/NMI and TEST during normal operation are logic 1 and logic 0, respectively. To achieve these levels and to use the corresponding RS-232 signals as power-supply lines, it is necessary to use two inverters (IC1A, IC2B) for the RST/NMI pin and one inverter (IC2E) for the TEST pin.

Devices without the TEST pin require the inverted TEST pin sequence on their TCK pin to invoke the bootstrap loader. Thus, the corresponding signal is inverted (inverter IC2F).

Diodes prevent discharge of capacitor C1 to allow control of the RS-232 lines (RTS and DTR).

3.3 Target Connector

Table 4. Pin Assignment of Target Connector[(1)]

PIN	SIGNAL NAME	DEVICES WITH TEST PIN	PIN ON MSP430F13x OR MSP430F14x	PIN ON MSP430F4xx
1	TXD	P1.1	P1.1	P1.0
2	TCK	Do not connect[(2)]	TCK	TCK
3	RXD	P2.2	P2.2	P1.1
4	RXD	RST/NMI	RST/NMI	RST/NMI
5	GND	GND	GND	GND
6	V_{CC} (3.0 V)	V_{CC}[(3)]	V_{CC}[(3)]	V_{CC}[(3)]
7	TST	TEST	Do not connect	Do not connect
8	VCC_IN	V_{CC}[(3)]	V_{CC}[(3)]	V_{CC}[(3)]
9	Not connected	—	—	—
10	Not connected	—	—	—

(1) For device-specific BSL pin information, refer to the applicable device data sheet.
(2) Signal TCK must not be connected on devices with the TEST pin.
(3) Pin V_{CC} (3.0 V) is a voltage source that can provide a limited current, depending on the serial port driver capability. If an external power supply is used, V_{CC} (3.0 V) must not be connected to the target. In this case, the external supply voltage must be connected to pin VCC_IN. Otherwise, VCC_IN must be unconnected.

3.4 Parts List

Table 5. Universal BSL Interface Parts List

PART	VALUE/PART NUMBER	PACKAGE	COMMENT
C1	33 µF, 16 V	SMD 7243	
C2	100 nF	SMD 0805	
C3	2.2 µF, 6.3 V	SMD 1206	
C4	100 nF	SMD 0805	
C5	33 µF, 16 V	SMD 7243	
C6	100 nF	SMD 0805	
D1	BAV70	SOT23	High-speed double diode
D2	BAV70	SOT23	High-speed double diode
D3	BAV70	SOT23	
IC1	TPS76030	SOT23/5	TI
IC2	74AHC14	SO14	TI
IC3	TL062D	SO8	TI
R1	330 kΩ	SMD 0805	
R2	330 kΩ	SMD 0805	
R3	330 kΩ	SMD 0805	
R4	680 kΩ	SMD 0805	
R5	680 kΩ	SMD 0805	
R6	680 kΩ	SMD 0805	
R7	330 kΩ	SMD 0805	
R8	330 kΩ	SMD 0805	
R9	3.3 kΩ	SMD 0805	
R10	3.3 kΩ	SMD 0805	
R11	100 kΩ	SMD 0805	
R12	0 kΩ	SMD 0805	
R13	680 kΩ	SMD 0805	
J1	pinhd-2x5	2X05	Target connector (see Table D-2)
J2	F09HP284	9-SUB-D female	RS-232 connector
CON3	RESET	SMD0805	Pads to connect an optional reset button

4 Software Description

This section explains the basic sequences required to access the bootstrap loader using the RS-232 interface of a PC. The code presented here is written in C language using 32-bit Windows™ API calls under Microsoft Visual C++™ 5.0 (it may also work seamlessly under version 6.0). The code was originally written for 16-bit Windows and ported to 32-bit Windows (Win32™, Windows 95/98™, and Windows NT™), mainly by replacing the function names. Some of the Win32 features were not used, to maintain portability. A commercial library may be used instead of the Windows API interface.

A detailed description titled *Serial Communications in Win32* is available online at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnfiles/html/msdn_serial.asp. Online documentation on Visual C++ is also available from the Microsoft development network library.

Please consult the Windows software development kit (SDK) documentation for a detailed description of the API functions used.

Complete software listings can be found in Appendix A.

4.1 Global Variables

The following global variables are used throughout the code examples. The definitions of types DCB, COMSTAT, and COMMTIMEOUTS can be found in the Windows SDK documentation.

```
HANDLE       hComPort;    /* COM-port handle              */
DCB          comDCB;      /* COM-port control settings    */
COMSTAT      comState;    /* COM-port status information */
COMMTIMEOUTS orgTimeouts; /* Original COM-port time-out   */
```

4.2 Initialization of the RS-232 Port

To access the serial RS-232 port, the program must request a handle for the port it wants to use (usually either COM1 or COM2). The following code can be used for this request:

```
/* Size of internal WINDOWS-Comm buffer: */
#define QUEUE_SIZE       512
. . .
char* lpszDevice= "COM1" /* For example ... *./
. . .
hComPort= CreateFile(lpszDevice, GENERIC_READ | GENERIC_WRITE,
                     0, 0, OPEN_EXISTING, 0, 0);
if (hComPort == INVALID_HANDLE_VALUE)
{ . . . /* Error! */
}
if (SetupComm(hComPort, QUEUE_SIZE, QUEUE_SIZE) == 0)
{ . . . /* Error! */
}
```

An operation known as overlapped input/output (I/O) can be performed under Win32. This means that the system may immediately return to the caller, even if the I/O operation is not finished, and signal the caller when the operation is complete. Overlapped operation is not a good choice when portability is a concern because most operating systems do not support it. For this reason it is not used in this program, and the corresponding parameters required when calling CreateFile are set to zero.

After receiving a valid handle, the settings of the communication port must be defined and assigned (see Table 1).

Get original settings first:

```
if (!GetCommState(hComPort, &comDCB))
{ . . . /* Error! */
}
```

The settings then can be modified. The most important settings for communication with the bootstrap loader are shown in the following program section:

```
comDCB.BaudRate      = CBR_9600;          /* Startup Baudrate: 9,6kBaud */
comDCB.ByteSize      = 8;
comDCB.Parity        = EVENPARITY;
comDCB.StopBits      = ONESTOPBIT;
comDCB.fBinary       = TRUE;              /* Enable Binary Transmission */
comDCB.fParity       = TRUE;              /* Enable Parity Check */
comDCB.fRtsControl   = RTS_CONTROL_ENABLE; /* For power supply and TEST */
                                          /* pin control */
comDCB.fDtrControl   = DTR_CONTROL_ENABLE; /* For power supply and RST/NMI */
                                          /* pin control */
. . .
```

Finally, the modified settings are assigned to the port:

```
if (!SetCommState(hComPort, &comDCB))
{ . . . /* Error! */
}
```

A Win32 application should always set communication time-outs when using a communication port; otherwise, default settings or left-over values from previous applications are used. This application does not require time-out functionality. Therefore, the time-outs are completely disabled once the original settings are saved so they can be restored at the end of the program:

```
/* Save original time-out values: */
GetCommTimeouts(hComPort, &orgTimeouts);
/* Set Windows time-out values (disable built-in time-outs): */
COMMTIMEOUTS timeouts;
timeouts.ReadIntervalTimeout= 0;
timeouts.ReadTotalTimeoutMultiplier= 0;
timeouts.ReadTotalTimeoutConstant= 0;
timeouts.WriteTotalTimeoutMultiplier= 0;
timeouts.WriteTotalTimeoutConstant= 0;
if (!SetCommTimeouts(hComPort, &timeouts))
{ . . . /* Error! */
}
```

The transmit and receive buffers are cleared to complete the initialization sequence:

```
PurgeComm(hComPort, PURGE_TXCLEAR | PURGE_TXABORT);
PurgeComm(hComPort, PURGE_RXCLEAR | PURGE_RXABORT);
```

A complete example of the initialization routine can be found in *Serial Communication Implementation File* (see Appendix A) in this routine:

```
int comInit(LPCSTR lpszDevice, DWORD aTimeout, int aProlongFactor)
```

4.3 Invoking the Bootstrap Loader

The levels on RST/NMI and TEST must be toggled as shown in section 2.1 to invoke the bootstrap loader.

The following subroutines can be used to control the corresponding RS-232 lines RTS and DTR:

```
void SetRSTpin(BOOL level)
/* Controls RST/NMI pin (0: GND; 1: VCC) */
{
  if (level == TRUE)
    comDCB.fDtrControl = DTR_CONTROL_ENABLE;
  else
    comDCB.fDtrControl = DTR_CONTROL_DISABLE;
  SetCommState(hComPort, &comDCB);
} /* SetRSTpin */
void SetTESTpin(BOOL level)
/* Controls TEST pin (0: VCC; 1: GND) */
{
  if (level == TRUE)
    comDCB.fRtsControl = RTS_CONTROL_ENABLE;
  else
    comDCB.fRtsControl = RTS_CONTROL_DISABLE;
  SetCommState(hComPort, &comDCB);
} /* SetTESTpin */
```

Calling the function *SetRSTpin* with a 0 pulls the RST/NMI pin to ground, whereas calling the function *SetTESTpin* with a 0 applies V_{CC} to the TEST pin. This difference is due to the different number of inverters used at these pins (see Section 3).

The following subroutine allows resetting of the MSP430 on the bootstrap loader hardware using the functions previously shown. First it is necessary to charge capacitor C1 to supply power to the board. Then the RST/NMI and TEST pins can be toggled as required. It is possible to reset the MSP430 and start the user program (*invokeBSL=FALSE*) or to invoke the bootstrap loader with *invokeBSL=TRUE*:

```
void bslReset(BOOL invokeBSL)
{
  /* To charge capacitor on boot loader hardware: */
  SetRSTpin(1);
  SetTESTpin(1);
  delay(250);
  SetRSTpin(0);    /* RST  pin: GND */
  if (invokeBSL)
  {
    SetTESTpin(1); /* TEST pin: GND */
    SetTESTpin(0); /* TEST pin: Vcc */
    SetTESTpin(1); /* TEST pin: GND */
    SetTESTpin(0); /* TEST pin: Vcc */
    SetRSTpin (1); /* RST  pin: Vcc */
    SetTESTpin(1); /* TEST pin: GND */
  }
  else
  {
    SetRSTpin(1);  /* RST  pin: Vcc */
  }
  /* Give MSP430's oscillator time to stabilize: */
  delay(250);
  /* Clear buffers: */
  PurgeComm(hComPort, PURGE_TXCLEAR); PurgeComm(hComPort, PURGE_RXCLEAR);
} /* bslReset */
```

It is possible to gain access to the RS-232 interface of the PC and to invoke the bootstrap loader using the functions presented so far.

We are now ready to access the bootstrap loader.

4.4 Access to the Bootstrap Loader

This section describes the basic routines to access the bootstrap loader. Table 6 gives an overview of the functions presented and indicates the sections within Appendix A where their complete listings can be found. The constant definitions are located in the header listings.

Table 6. Bootstrap Loader Access Functions

SECTIONS	FUNCTIONS
Bootstrap loader communication header file, bootstrap loader communication implementation file	bslReset, bslSync, bslTxRx
Serial-communication header file, serial-communication implementation file	comInit, comTxRx, comDone

4.4.1 Synchronization

The PC and the MSP430 must be synchronized before each frame that calls a function. The following character (0x80) must be sent by the PC to the MSP430 for this purpose:

```
#define BSL_SYNC 0x80
```

Therefore, this value is assigned to a variable and the function *WriteFile* is called with the communication-port handle as the first parameter. The number of bytes to transmit must be specified (1 is used here). The last two parameters are unimportant and can be ignored in this particular case.

```
/* Send synchronization byte: */
ch = BSL_SYNC;
WriteFile(hComPort, &ch, 1, &NrTx, NULL);
```

If the bootstrap loader receives this character correctly, it returns a DATA_ACK (0x90); otherwise, it returns an unknown value because the loader needs this synchronization character to generate its timing (for serial communication, for example). It is also possible that no character is returned, as when no MSP430 with bootstrap loader function is connected. The following subroutine is used to check whether a given number of characters was received within a given time period. It uses the function *ClearCommError* to receive the actual status of the communication port. The field *cbInQue* in the COMSTAT structure holds the number of received bytes:

```
int comWaitForData(int count, DWORD timeout)
{
  DWORD errors;
  int rxCount= 0;
  DWORD startTime= GetTickCount();
  do
  {
    ClearCommError(hComPort, &errors, &comState);
  } while (((rxCount= comState.cbInQue) < count) &&
           (calcTimeout(startTime) <= timeout));
  return(rxCount);
}
```

This subroutine is used in the synchronization function to wait for one character. If characters are received, they can be queried using the function *ReadFile*, shown in the following program section. The synchronization is successful when the character received is a DATA_ACK.

```
/* Wait for 1 byte; time-out: 100ms */
rxCount= comWaitForData(1, 100);
if (rxCount > 0)
{
  ReadFile(hComPort, &ch, 1, &NrRx, NULL);
  if (ch == DATA_ACK)
  { return(ERR_NONE); } /* Sync. successful */
}
```

The routine *int bslSync()* implements the function previously described.

4.4.2 Transmission of Frame

The following function is used to send and receive a frame:

```
int comTxRx(BYTE cmd, BYTE data[], BYTE length)
```

This function implements the functionality of the TI Standard Serial Protocol. Only a subset of the functionality is needed and supported for bootstrap loader communication. This routine uses the Win32 functions previously described.

The function

```
int bslTxRx(BYTE cmd, WORD addr, WORD len, BYTE blkout[], BYTE blkin[])
```

combines the synchronization and transmission of the frame. It also configures the frame to the requirements of the bootstrap loader. One important requirement is that the number of bytes sent to the bootstrap loader (using the command TXBLK, for instance) or requested by the loader (using command RXBLK) must always be even.

4.5 Calling the Bootstrap Loader Functions

It is fairly simple to call the functions of the bootstrap loader using the function *bslTxRx* previously presented. The parameters must be set as required within the function call (see examples in the *Complete Application* section).

The constant definitions corresponding to the available commands are found in the *Bootstrap Loader Communication Header File* (see Appendix A).

4.6 Releasing the RS-232 Port

At the end of a program, the RS-232 port must be released using the function *CloseHandle(hComPort)*. Otherwise, other applications cannot use this port because the access rights to the serial communication ports are usually granted to only one program at a time.

The routine *int comDone( )* provides a more sophisticated method to release the RS-232 port. It waits until all remaining data is transmitted, clears all buffers, and restores the original time-out settings.

Note that closing the serial communication port usually cuts power to an application that is powered via the serial port, because the corresponding control lines are disabled.

4.7 Complete Application

A small application is developed in this section to program a file in TI-TXT format into the MSP430 flash memory. The complete demonstration program code is contained in *Bootstrap Loader Demonstration Program* in Appendix A. The definitions of the variables used are also found there.

First, the communication port has to be opened. The COM-port name can be changed to fit any particular needs, or it can be obtained from the command line.

```
if (comInit("COM1", DEFAULT_TIMEOUT, 4) != 0)
{ . . . /* Error! */
}
```

Then the bootstrap loader is invoked:

```
bslReset(1);
```

In the next step, the flash memory is completely erased using the mass erase command:

```
if ((error= bslTxRx(BSL_MERAS, /* Command: Mass Erase              */
                    0xff00,    /* Any address within flash memory. */
                    0xa506,    /* Required setting for mass erase! */
                    NULL, blkin)) != 0)
{ . . . /* Error! */
}
```

The password to access the protected functions of the bootstrap loader gets reset when the flash memory is erased. All memory cells are now set to 0FFh. The protected functions are enabled by sending the corresponding password.

```
/* Fill blkout with 0xff */
for (I= 0; I < 0x20; I++)
{
  blkout[i]= 0xff;
}
if ((error= bslTxRx(BSL_TXPWORD, /* Command: transmit password  */
                    0xffe0,      /* Address of interrupt vectors */
                    0x0020,      /* Number of bytes             */
                    blkout, blkin)) != 0)
{ . . . /* Error! */
  /* Special case here: 7(ERR_RX_NAK): Password not accepted! */
}
```

Afterwards, the file in TI-TXT format is parsed and the data is programmed into the flash memory and verified.

There is a separate subroutine in the demonstration listing that can be called from the main program to parse the file and to program or verify the flash contents:

```
/* Program: */
/if ((error= programFlash("TEST.TXT", ACTION_PROGRAM)) != 0)
{ . . . /* Error! */
}
/* Verify: */
if ((error= programFlash("TEST.TXT", ACTION_VERIFY)) != 0)
{ . . . /* Error! */
}
```

The routine *programFlash* parses the file with a given name (the file name can be derived from the command line), fills a buffer with the extracted data, and calls another subroutine when the buffer is almost full.

If the data must be programmed, it is sent to the bootstrap loader using the transmit-block command (BSL_TXBLK):

```
error= bslTxRx(BSL_TXBLK, addr, len, blkout, blkin);
```

Data is read from the bootstrap loader (receive block, BSL_RXBLK) and compared against the contents of the transmission buffer for verification:

```
error= bslTxRx(BSL_RXBLK, addr, len, NULL, blkin);
if (error != 0)
```

```
{ . . . /* Cancel! */
}
else
{
  for (I= 0; I < len; I++)
  { /* Compare data in blkout and blkin: */
    if (blkin[i] != blkout[i])
    {
      printf("Verification failed at %x (%x, %x)\n", addr+I, blkin[i],    blkout[i]);
      return(ERR_VERIFY_FAILED); /* Verify failed! */
    }
  }
} /* for (I) */
```

Note that a similar sequence can be used to check the erasure of this range. In this case, the contents of *blkin* are compared against the erasure pattern 0xff.

If a readout functionality is required, it is also possible to write the received data to a file instead of comparing it with given data. (Note that this feature is not included in the provided source code.)

After successful verification, the MSP430 can be reset and the user program can start executing:

```
bslReset(0);
```

The serial communication port must be released at the end of the program:

```
comDone();
```

Now users have all the pieces together to write their own applications to access the MSP430 bootstrap loader and to adapt it to their special needs.

4.8 Error Recovery

There is no error-recovery mechanism implemented within the demonstration program. The program is aborted when an error is detected. In some cases, it might be useful to implement some kind of error-recovery mechanism.

If a data frame transmitted to the MSP430 is rejected with a data-not-acknowledged signal (DATA_NAK), the transmission of the frame can be repeated. But it is possible that wrong data has been programmed in the flash, and a more complex recovery mechanism that includes verification, erasure, and reprogramming might be required.

If a received frame is not correct (wrong checksum, inconsistent lengths), the command previously sent to receive a block from the MSP430 must be repeated.

4.9 Advanced Features

4.9.1 Detecting the Bootstrap Loader Version

Detecting the bootstrap loader version within the currently connected device requires unlocking the protected functions (see previous sections). The address (0x0ffa) where the bootstrap loader version is stored then can be read. The version information consists of two bytes — a main revision number stored in the first byte, and a subrevision number stored in the second byte. After extracting this information from the input buffer, the version can be displayed or used for further processing, depending on the current loader version.

```
/* Read actual bootstrap loader version.                   */
if ((error= bslTxRx(BSL_RXBLK, /* Command: Read/Receive Block */
                    0x0ffa,    /* Start address              */
                    2,         /* No. of bytes to read       */
                    NULL, blkin)) == ERR_NONE)
{
  BYTE bslVerLo;
  BYTE bslVerHi;
  memcpy(&bslVerHi, &blkin[0], 1);
  memcpy(&bslVerLo, &blkin[1], 1);
  bslVer= (bslVerHi << 8) | bslVerLo;
  printf("Current bootstrap loader version: %x.%x\n", bslVerHi, bslVerLo);
}
else
{ . . . /* Error Handling */ . . . }
```

4.9.2 Executing Code

The bootstrap loader command LOADPC allows the execution of previously-programmed code. This command loads a given address in the program counter and starts execution at this address. The executed code may be located in any type of memory (such as RAM or flash memory).

For example, this feature can be used to load a calibration routine into RAM, run the calibration, return to the bootstrap loader, and read back the calibration data. Or it can be used to program and execute another loader with additional features. The available memory (RAM or flash), the resources required by the bootstrap loader, their initialization (for example, P1SEL, P2SEL, and CCTL0 must be cleared), and especially the stack usage must be carefully considered. Note that the initialization of the stack pointer differs from version to version of the bootstrap loader.

The loader start-address vector stored at address (0x0c00) can be used to return from a loaded routine to the bootstrap loader:

```
              br     &00C00h      ; Return to bootstrap loader
```

The demonstration program shows how to program an updated bootstrap loader into RAM and then execute it. As a prerequisite, the protected functions of the loader must be unlocked by sending the appropriate password.

After obtaining access to the protected bootstrap loader commands, a subroutine within the loader code must be called to prepare the position of the stack pointer, if the loader version is 1.10 or below. These versions of the loader have a so-called dynamic stack pointer initialization, and there is no assurance that loading data to RAM will not interfere with the actual stack. The stack pointer is initialized to point to the fixed loader stack frame by loading the PC with the address of the appropriate routine within the bootstrap loader. Calling this function locks the protected commands, and the password must be resent.

```
if (bslVer <= 0x0110)           /* BSL Version 1.10 or below? */
{
  if ((error= bslTxRx(BSL_LOADPC,  /* Command: load PC         */
                      0x0C22,      /* Address to load into PC */
                      0,           /* No additional data!     */
                      NULL, blkin)) != ERR_NONE)
  { . . . /* Error! */
  }
  /* Resend password to regain access to protected functions. */
  if ((error= txPasswd(passwdFile)) != ERR_NONE)
```

```
  { . . . /* Error! */
    /* Special case here: 7(ERR_RX_NAK): Password not accepted! */
  }
}
```

It is now possible to program any data into RAM using the standard bootstrap loader methods. In the demonstration program, *routineTIText* is used to program and verify another loader with the name of the corresponding TI-TXT file given in the variable *newBSLFile*:

```
printf("Load new BSL \"%s\" into RAM...\n", newBSLFile);
if ((error= programTIText(newBSLFile, /* File to program */
                          ACTION_PROGRAM)) != ERR_NONE)
{ . . . /* Error! */ }
printf("Verify new BSL \"%s\"...\n", newBSLFile);
if ((error= programTIText(newBSLFile, /* File to verify */
                          ACTION_VERIFY)) != ERR_NONE)
{ . . . /* Error! */ }
```

The programmed code now can be executed. The new loader has a start-up vector located at address 0x0300. After reading this address, the contents of the start-up vector (variable *startaddr* in the following code snippet) are used to load the program counter:

```
/* Read startvector of bootstrap loader: */
if ((error= bslTxRx(BSL_RXBLK, 0x0300, 2, NULL, blkin)) == ERR_NONE)
{
  WORD startaddr;
  memcpy(&startaddr,  &blkin[0], 2);
  printf("Starting new BSL at %x...\n", startaddr);
  error= bslTxRx(BSL_LOADPC, /* Command: Load PC        */
                 startaddr,  /* Address to load into PC */
                 0,          /* No additional data!     */
                 NULL, blkin);
}
/* . . . Error Handling . . . */
```

The loaded program is executed now. Since it is just another loader with the same communication protocol as the original one, it is possible to continue programming the flash memory as if it was the original loader.

The protected functions of the loader are locked, since it was started with the start address pointing to its initialization routine. The password must be sent again to unlock these commands. Execution then can proceed as normal.

Within the demonstration program, the command-line parameter *-b* controls if a new loader is used. For example, to use the loader contained in the TI-TXT file BL_130V.TXT, the following command line may be used:

```
BSLDEMO -bBL_130V.TXT +epr TEST.TXT
```

4.10 Patch for First Version of Bootstrap Loader

The first versions (1.10 and below) of the bootstrap loader require a small patch to program the flash (Bug Ids: BSL2, BSL3, and BSL4). The patch is described in this section, and its handling is included in the program BSLDEMO.C (see *Bootstrap Loader Demonstration Program* in Appendix A). The TI-TXT file *patch.txt*, included in Appendix A, is also required. The patch handling within the demonstration program can be switched off (for future versions of the bootstrap loader) by deleting, or *commenting out*, the following line:

```
#define WORKAROUND
```

The parts of the code required for the workaround are surrounded by preprocessor commands:

```
#ifdef WORKAROUND
   . . .
#endif
```

After obtaining access to the protected bootstrap loader commands, the position of the stack pointer must be prepared for the patch, as described in Section 4.9.2. Afterwards, the patch can be written into RAM. The text file *patch.txt* holding the patch can be found in Appendix A. Downloading is performed by the flash-programming function that parses a text file:

```
programFlash("PATCH.TXT", ACTION_PROGRAM | ACTION_VERIFY);
```

In this case, the masks ACTION_PROGRAM and ACTION_VERIFY are used together to program and verify the patch in one single pass. This means that the file *patch.txt* is read only once. Note that the patch must be located in the same directory as the executable program. To use the patch for programming, only the PC needs to be loaded with the start address of the patch (0x0220) before sending a frame. In the demonstration program, the invocation of the patch is done in the separate *preparePatch* function (see *Bootstrap Loader Demonstration Program* in Appendix A).

There is another bug that can affect memory cells (either RAM or peripheral-module registers) if the transmitted frame has a certain checksum; unfortunately, a general workaround can not be provided. Transmitting data to, or receiving data from the MSP430 using the patch prevents this error. In all other cases, the only help that can be provided is a warning when this situation occurs (however, it is unlikely that the error occurs if reading and writing are performed using the patch). The warning is generated within the *comRxTx* function contained in file SSP.C (see *Serial Communication Implementation File* in Appendix A).

```
#define BSL_CRITICAL_ADDR 0x0A00
{
WORD accessAddr= (0x0212 + (checksum^0xffff)) & 0xfffe;
  if (BSLMemAccessWarning && (accessAddr < BSL_CRITICAL_ADDR))
  {
    printf("WARNING: This command might change data
           "at address %x or %x!\n",
           accessAddr, accessAddr + 1);
  }
}
```

The global variable *BSLMemAccessWarning* allows warning message turnon or turnoff. Since the patch fixes this bug, the warning is turned off within the *preparePatch* function, and turned back on in the *postPatch* function (see *Bootstrap Loader Demonstration Program* listing in Appendix A).

The complete sequence for receiving data from the MSP430 becomes:

```
error= preparePatch();
if (error != ERR_NONE) return(error);
error= bslTxRx(BSL_RXBLK, addr, len, NULL, blkin);
postPatch();
```

Similarly, the transmission of data to the MSP430 is handled as:

```
error= preparePatch();
if (error != ERR_NONE) return(error);
error= bslTxRx(BSL_TXBLK, addr, len, blkout, blkin);
postPatch();
```

The demonstration program solves an additional problem that applies only to devices with flash memory sizes greater than 4K bytes. For these memories, the built-in mass-erase time may be too short to erase it completely. This problem can be fixed simply by repeating the mass-erase command several times. This workaround is activated by defining the default number of mass-erase cycles:

```
#define ADD_MERASE_CYCLES  20
```

The number of mass-erase cycles can be changed to any value using the demonstration program's command-line parameter *-m*. For example, *-m1* can be used to have only one mass-erase cycle with MSP430F11x(1), which is sufficient for these devices.

5 References

1. *MSP430F11x Mixed Signal Microcontroller* data sheet, literature number SLAS256
2. *MSP430F11x1 Mixed Signal Controller* data sheet, literature number SLAS241
3. *Features of the MSP430 Bootstrap Loader* application report, literature number SLAA089
4. Denver, Allen. *Serial Communication in Win32*, Microsoft Developer Network (MSDN) Library
5. *Microsoft Win32* Software Development Kit (SDK) documentation

Appendix A Software

A.1 Building the Demonstration Program

To build the demonstration program, it is necessary to compile the files bslcomm.c and bsldemo.c and to link the resulting object files. It is not necessary to compile the file ssp.c separately because it is directly included in the file bslcomm.c. For a different approach, copy and paste the contents of ssp.c into bslcomm.c at the position of the following line, and remove this line:

```
#include "ssp.c"
```

For example, if using Visual C++, create a new project and select the *Win32 Console Application* template. The project created must be empty. Include the files bsldemo.c, bslcomm.c, and bslcomm.h to build the project. As previously stated, the files ssp.c and ssp.h automatically are included. The building process will fail if the ssp.c file is included in the project.

A.2 TI-TXT File for Bootstrap Loader Patch (patch.txt)

```
@0220
31 40 1A 02 09 43 B0 12 2A 0E B0 12 BA 0D 55 42
0B 02 75 90 12 00 1F 24 B0 12 BA 02 55 42 0B 02
75 90 16 00 16 24 75 90 14 00 11 24 B0 12 84 0E
06 3C B0 12 94 0E 03 3C 21 53 B0 12 8C 0E B2 40
10 A5 2C 01 B2 40 00 A5 28 01 30 40 42 0C 30 40
76 0D 30 40 AC 0C 16 42 0E 02 17 42 10 02 E2 B2
08 02 14 24 B0 12 10 0F 36 90 00 10 06 28 B2 40
00 A5 2C 01 B2 40 40 A5 28 01 D6 42 06 02 00 00
16 53 17 83 EF 23 B0 12 BA 02 D3 3F B0 12 10 0F
17 83 FC 23 B0 12 BA 02 D0 3F 18 42 12 02 B0 12
10 0F D2 42 06 02 12 02 B0 12 10 0F D2 42 06 02
13 02 38 E3 18 92 12 02 BF 23 E2 B3 08 02 BC 23
30 41
q
```

Appendix B PCB Layout Suggestion

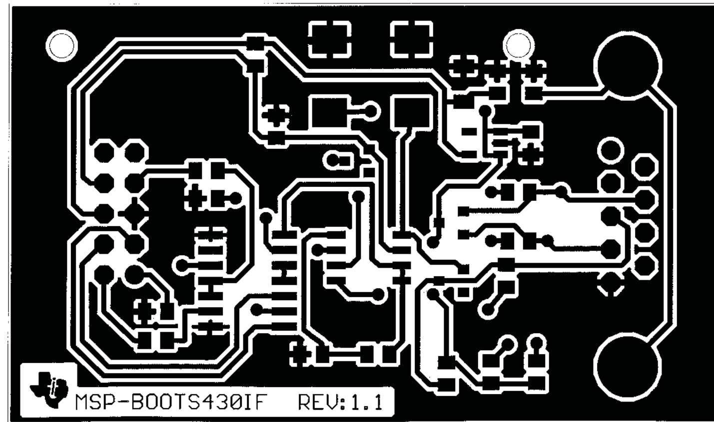

Figure B-1. Universal BSL Interface PCB Layout—Top

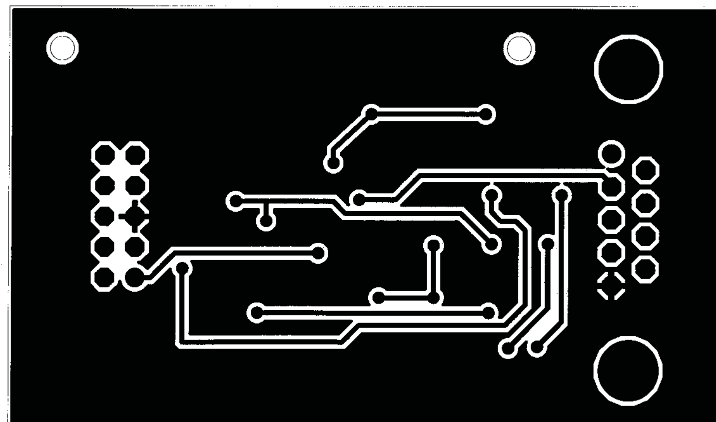

Figure B-2. Universal BSL Interface PCB Layout—Bottom

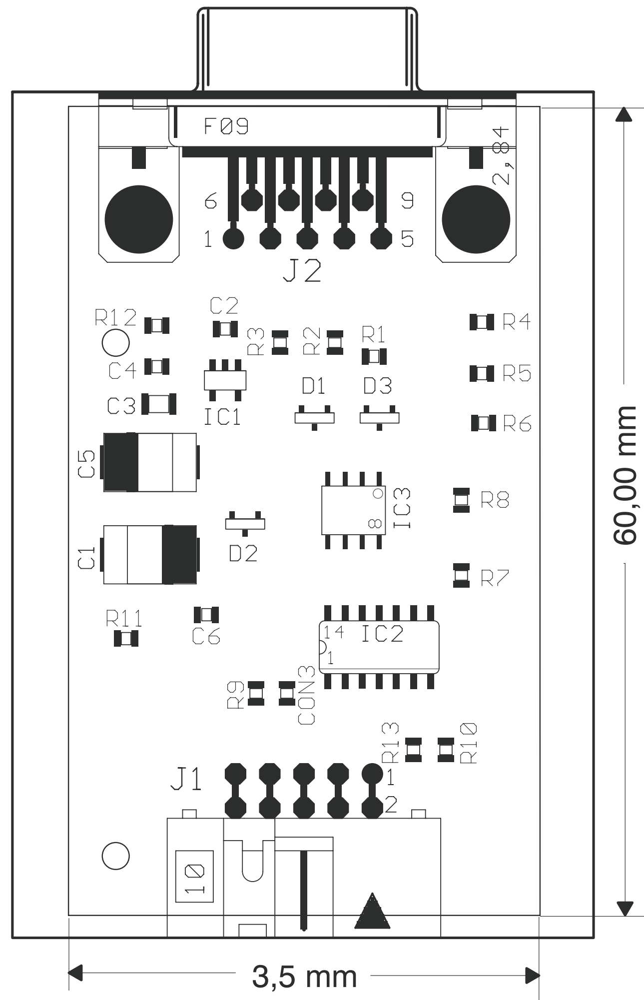

Figure B-3. Universal BSL Interface Component Placement

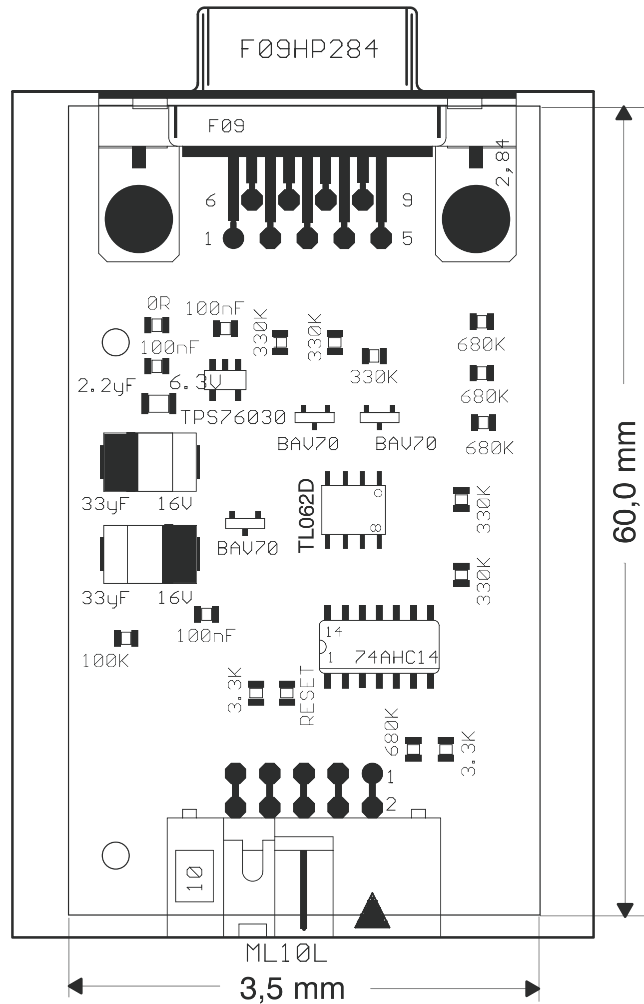

Figure B-4. Universal BSL Interface Component Placement

Appendix C Demonstration Program Usage

The bootstrap loader demonstration program has a simple command-line interface. The only parameter required is the name of the TI-TXT file to program. All other parameters are optional and must be entered before of the TI-TXT file name. The TI-TXT file name must be the last parameter.

Table C-1 shows the command-line parameters available.

Table C-1. Command-Line Parameters

PARAMETER	DESCRIPTION	EXAMPLE
-h	Shows help screen	-h
-c{COM-port name}	Specifies the COM port to be used (default: COM1)	-cCOM2
-p{TI-TXT file name}	Specifies a TI-TXT file containing the actual password to access the bootstrap loader	-pint_vect.txt
-w	The program will wait after successful programming and reset for the <ENTER> key. The application can run powered via the serial port.	-w
-1	Do erase-check, programming, and verification in one pass (the TI-TXT file is read and parsed only once). This option is discouraged with the first version(s) of the bootstrap loader that require the work-around patch.	-1
-a{TI-TXT file name}	Specifies a TI-TXT file containing a valid patch for the bootstrap loader	-apatch.txt
-b{TI-TXT file name}	Specifies a TI-TXT file containing a valid loadable bootstrap loader that is loaded before any further programming takes place and that replaces the original one for programming	-bbl_130v.txt
-f{num}	Specifies the maximum number of data bytes within one transmitted frame	-f240
-e{startnum}	Erase segment specified by {startnum}	-e 0xF000
-m{num}	Specifies the number of mass-erase cycles	-m20
-s{num}	Changes the baud rate (num=0:9600; 1:19200; 2:38400)	-s2
-r{startnum} {lenmum} {file}	Read memory from startnum for a size of lenmum and write data to file in TI-TXT format (values are in HEX-format)	-r0xF000 0x1000 out.txt
-x	Enables MSP430X extended memory support	-x

In addition to these command-line parameters, a parameter exists that allows control of the program flow. For example, it is possible to just verify the contents of the flash memory. This parameter is introduced by the + character. The + character is followed by the specification of the steps to be taken (and only these steps are taken). Table C-2 shows the modifiers available. The steps are taken in the order of the modifiers within Table C-2.

Table C-2. Program-Flow Modifiers[1]

MODIFIER	DESCRIPTION
a	Restore INFOA after mass erase (only with mass erase)
e	Erase complete flash (mass erase)
c	Check erasure
p	Program given file
v	Verify against given file
r	Reset device
w	Wait for <ENTER> at the end
u	User called – Skip entry sequence

(1) If the modifier e is omitted, the flash is not erased completely and, thus, it is required to provide a password file using the –p parameter (see fourth invocation example in Table C-3).

Table C-3 shows examples of the demonstration program invocation.

Table C-3. Invocation Examples[(1)]

EXAMPLE	DESCRIPTION
bsldemo -h	Shows help screen
bsldemo test.txt	Erase flash, check erasure, program and verify file test.txt, exit
bsldemo -1 -w -cCOM2 test.txt	Same as above, but the hardware is connected to COM2; erase-check, program, and verify are done in one pass through file test.txt; the program waits for <ENTER> at the end.
bsldemo +vrw -pint_vect.txt test.txt	Use data within file int_vect.txt as password, verify against file test.txt (no erasure or programming), reset MSP430, wait for <ENTER> at the end.
bsldemo +rw -pint_vect.txt test.txt	Reset MSP430 and wait for <ENTER> at the end. Password and file name are also required.
bsldemo -bbl_130v.txt +epr test.txt	Load new bootstrap loader bl_130v.txt into RAM and program file test.txt using the new loader (the verification step is omitted because the loader bl_130v.txt does the verification internally during programming.) Note: There must be enough memory to load the new loader.

(1) If the downloaded bootstrap loader Bl_130v.txt is used or the connected device has a bootstrap loader version 1.40 and higher, the verification step can be omitted because these loaders perform the verification during programming (see last invocation example in Table C-3).

Appendix D Errata

This appendix summarizes errata in former revisions of the *Application of Bootstrap Loader in MSP430* application report.

Errata SLAA096A:

- Appendix D: Universal Bootstrap Loader Interface Board: Operational amplifier IC2 must be replaced with TL062D or equivalent type.

Appendix E Third-Party Support

Please visit the MSP430 third-party web page at www.ti.com/msp430 for updated information on available MSP430 BSL tools.

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 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.

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.

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

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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

Copyright © 2006, Texas Instruments Incorporated