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

Volker Rzehak   *MSP/ALP Design*

ABSTRACT

The bootstrap loader (sometimes called the bootloader) of the 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 note 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 personal computer (PC). The description provided and the C-source code of the software routines allow adaptation of the software to specific requirements.

Contents

List of Figures

List of Tables

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.

2 Bootstrap Loader Basics

This section reviews the basic principals and use of the bootstrap loader. Please see the documentation and data sheets for the flash-based MSP430 derivatives and the application note *Features of the MSP430 Bootstrap Loader* for more details.

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 (DSP) 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, too.

2.1 Invoking the Bootstrap Loader

The MSP430 bootstrap loader does not start automatically; a special sequence is required on the RST/NMI and TEST or TCK pins. TCK is used on devices with no dedicated TEST pin.

The start-up sequence for devices with a TEST pin is shown in Figures 1 and 2.

The user program, with its reset vector located at memory address 0FFFEh, starts when the TEST pin is pulled low during a low-to-high transition of RST/NMI (see Figure 1). Figure 2 shows the sequence required on RST/NMI and TEST to start the bootstrap loader.

Devices without a TEST pin just require the inverted TEST pin sequence on their TCK pin.

See the documentation mentioned in the Bootstrap Loader Basics section for more information on the start-up sequences. The data sheets of the particular MSP430 versions also describe the required sequences.
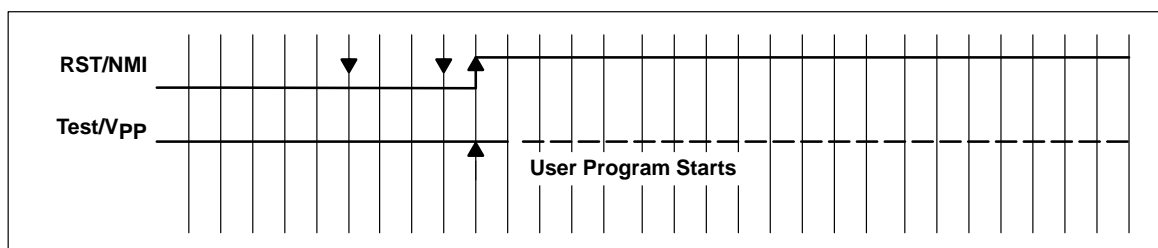

Figure 1. RST/NMI and TEST Sequence to Start User Program
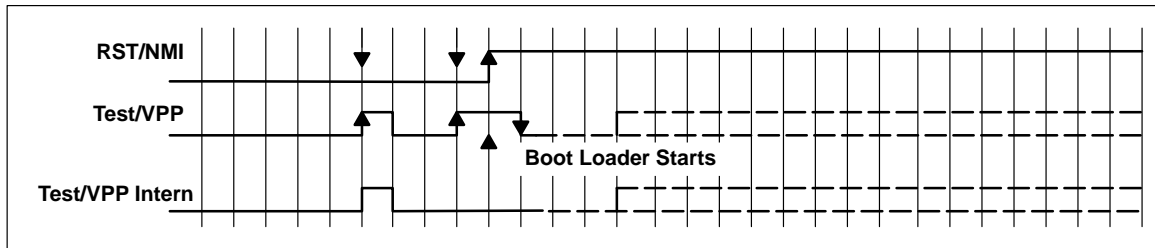
Figure 2. RST/NMI and TEST Sequence to Start Bootstrap Loader

2.2 Access to 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 to get the appropriate pinning information.

Table 1. Bootstrap Loader UART Settings

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

The protocol used to communicate with the bootstrap loader is derived from a more complex protocol; this adds some overhead to the user program.

First the PC must send a synchronization byte. If the bootstrap loader receives this character correctly, it returns an acknowledge byte.

After successful synchronization, the PC sends a frame containing a command and its data. The frame (see Figure 3) can be divided into a header section, a data section, and the check-sum bytes. The data section contains at least four bytes of data: usually a start address and a length, but some commands may ignore the contents or interpret it differently. In the case of word data (two bytes per word), the byte order is always low-byte/high-byte.

| Hdr | Cmd | L1 | L2 | A (lo) | A (hi) | L (lo) | L (hi) | Data | Data | . . . | Data | Data | | |

Header Section   Data Section   Check Sum

Hdr:	0×80 (Always)	
Cmd:	Code For Requested Function	
L1, L2:	Number of Bytes in Data Section	

A:	Address (Start)
L:	Length
(lo):	Low Byte
(hi):	High Byte

Figure 3. Frame Sent to Bootstrap Loader

After processing the frame, the bootstrap loader either returns an acknowledge signal, a negative acknowledge (if the frame was not valid), or a command-failed signal (if the command is not allowed).

2.3 Bootstrap Loader Functions

The 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.
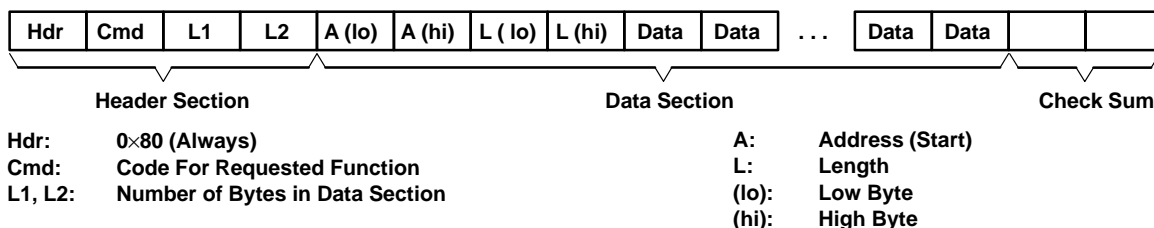
This section gives a brief overview of the commands available. The commands and their usage are described in more detail in the *Features of the MSP430 Bootstrap Loader in the MSP430F1121* document, literature number SLAA089, and in the software section of this application note.

Note that it is not currently 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 pass-word protected.

2.3.1 Unprotected Functions

- Set password, enables protected functions

- Mass erase, completely erases the flash memory. Afterwards the password to access protected functions is 16 times 0FFFFh.

2.3.2 Protected Functions

- Transmit block, writes data into MSP430's memory

- Receive block, reads data out of MSP430's memory

- Erase segment

- Load program counter, starts execution

Table 2. Bootstrap Loader Functions Overview

FUNCTION	CMD	L1=L2	ADDRESS	LENGTH	DATA
Set password	10h	24h	xx	xx	Password
Mass erase	18h	04h	Flash	0A506h	–
Transmit byte	12h	n+4	Start	n	Data (n byte)
Receive byte	14h	04h	Start	n	–
Erase	16h	04h	Segment	0A502h	–
Load PC	1Ah	04h	Start	xx	–

3 Hardware Description

The low-cost hardware presented in this application note (Figure 4) 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.
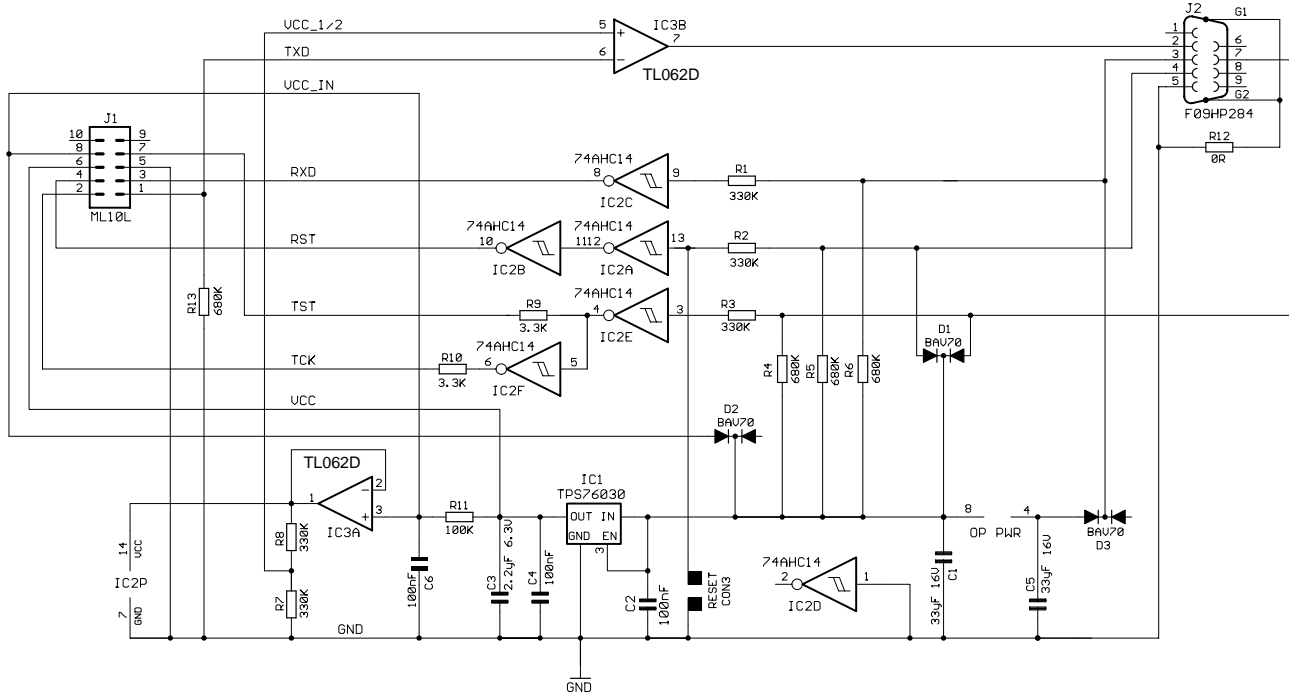
Figure 4. Bootstrap Loader Interface Schematic

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 (Texas Instruments 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 3 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 3. 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 4) to CMOS levels.

Table 4. RS-232 Levels

LOGIC LEVEL	RS-232 LEVEL	RS-232 VOLTAGE LEVEL
1	Mark	–3 V – 15 V
0	Space	3 V – 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) will override 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 Vcc (as in the TI 74HC14) or to GND (as in the TI 74AHC14). If the protection diode conducts to Vcc, 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 Vcc/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 Vcc/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 will not cause any 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 deliver a positive voltage to supply the bootstrap loader hardware, too.

For devices with 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 simply 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 5. Pin Assignment of Target Connector

PIN	SIGNAL NAME	PIN ON MSP430F11x(1)	PIN ON MSP430F14x OR MSP430F13x	PIN ON MSP430F4xx
1	TXD	P1.1	P1.1	P1.0
2	TCK	Do not connect (see Note 1)	TCK	TCK
3	RXD	P2.2	P2.2	P1.1
4	RST	RST/NMI	RST/NMI	RST/NMI
5	GND	GND	GND	GND
6	V_{CC} (3.0 V)	V_{CC} (see Note 2)	V_{CC} (see Note 2)	V_{CC} (see Note 2)
7	TST	Test	Do not connect	Do not connect
8	VCC_IN	V_{CC} (see Note 2)	V_{CC} (see Note 2)	V_{CC} (see Note 2)
9	Not connected	—	—	—
10	Not connected	—	—	—

NOTES: 1. Signal TCK must not be connected on MSP430F11x(1) devices.
   2. Pin VCC (3.0 V) is a voltage source that can provide a limited current, depending on the serial port driver's capability. If an external power supply is used, VCC (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, pin VCC_IN must be unconnected.

3.4 Parts List

Table 6. 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 Ω	SMD 0805	
R13	680 kΩ	SMD 0805	
J1	pinhd-2×5	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/techart/msdn_serial.htm. 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.

Microsoft, Windows, Win32, Windows NT, Windows CE, and Visual C++ are trademarks of Microsoft Corporation.

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

Get original settings first:

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

Then they 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 under *Serial Communication Implementation File* (see Appendix A) in 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 Vcc 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 7 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 7. 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 already 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* presented before. The parameters must be set as required within the function call (see the 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 can not 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 which 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's 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* simply 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 simply 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 needs to be repeated.

4.9   Advanced Features

4.9.1   Detecting Bootstrap Loader's Version

Detecting the bootstrap loader version within the device currently connected requires unlocking the protected functions (see previous sections). The address (0x0ffa) where the bootstrap loader version is stored can then 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's 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's 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's 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 Text 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 can now 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 can then 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 Text 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 text 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 turn-on or turn-off. 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).

So 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 follows:

```
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. Graf, Franz. *Features of the MSP430 Bootstrap Loader in the MSP430F1121*, 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   Listings

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. If you prefer 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 you are 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 your project. As stated before, the files ssp.c and ssp.h are included automatically. The building process will fail if you include the ssp.c file in your project.

A.2   Bootstrap Loader Communication Header File—bslcomm.h

```
/*************************************************************
 *
 * Copyright (C) 1999-2000 Texas Instruments, Inc.
 * Author: Volker Rzehak
 *
 *-----------------------------------------------------------
 * All software and related documentation is provided "AS IS" and
 * without warranty or support of any kind and Texas Instruments
 * expressly disclaims all other warranties, express or implied,
 * including, but not limited to, the implied warranties of
 * merchantability and fitness for a particular purpose. Under no
 * circumstances shall Texas Instruments be liable for any
 * incidental, special or consequential damages that result from
 * the use or inability to use the software or related
 * documentation, even if Texas Instruments has been advised of
 * the liability.
 *
 * Unless otherwise stated, software written and copyrighted by
 * Texas Instruments is distributed as "freeware". You may use
 * and modify this software without any charge or restriction.
 * You may distribute to others, as long as the original author
 * is acknowledged.
 *
 *************************************************************
 *
 * Project: MSP430 Bootstrap Loader Demonstration Program
 *
 * File:    BSLCOMM.H
 *
 * History:
 *    Version 1.00 (05/2000)
 *    Version 1.11 (09/2000)
 *      - Added definition of BSL_CRITICAL_ADDR.
 *
 *************************************************************/
#ifndef BSLComm__H
```

```
#define BSLComm__H
#include "ssp.h"
/* Transmit password to boot loader: */
#define BSL_TXPWORD 0x10
/* Transmit block    to boot loader: */
#define BSL_TXBLK    0x12
/* Receive  block  from boot loader: */
#define BSL_RXBLK    0x14
/* Erase one segment:                */
#define BSL_ERASE    0x16
/* Erase complete FLASH memory:      */
#define BSL_MERAS    0x18
/* Load PC and start execution:      */
#define BSL_LOADPC  0x1A
/* Bootstrap loader synchronization error: */
#define ERR_BSL_SYNC       99
/* Upper limit of address range that might be modified by
 * "BSL checksum bug".
 */
#define BSL_CRITICAL_ADDR 0x0A00
#ifdef __cplusplus
extern "C" {
#endif
extern int BSLMemAccessWarning;
/*-------------------------------------------------------------*/
void bslReset(BOOL invokeBSL);
/* Applies BSL entry sequence on RST/NMI and TEST/VPP pins
 * Parameters: invokeBSL = TRUE:  complete sequence
 *             invokeBSL = FALSE: only RST/NMI pin accessed
 */
/*-------------------------------------------------------------*/
int bslSync();
/* Transmits Synchronization character and expects to
 * receive Acknowledge character
 * Return == 0: OK
 * Return == 1: Sync. failed.
 */
/*-------------------------------------------------------------*/
int bslTxRx(BYTE cmd, WORD addr, WORD len,
            BYTE blkout[], BYTE blkin[]);
/* Transmits a command (cmd) with its parameters:
 * start-address (addr), length (len) and additional
 * data (blkout) to boot loader.
 * Parameters return by boot loader are passed via blkin.
 * Return == 0: OK
 * Return != 0: Error!
 */
#ifdef __cplusplus
}
#endif
#endif
/* EOF */
```

A.3 Bootstrap Loader Communication Implementation File—bslcomm.c

```
/***************************************************************
 *
 * Copyright (C) 1999-2000 Texas Instruments, Inc.
 * Author: Volker Rzehak
 *
 *-------------------------------------------------------------
 * All software and related documentation is provided "AS IS" and
 * without warranty or support of any kind and Texas Instruments
 * expressly disclaims all other warranties, express or implied,
 * including, but not limited to, the implied warranties of
 * merchantability and fitness for a particular purpose. Under no
 * circumstances shall Texas Instruments be liable for any
 * incidental, special or consequential damages that result from
 * the use or inability to use the software or related
 * documentation, even if Texas Instruments has been advised of
 * the liability.
 *
 * Unless otherwise stated, software written and copyrighted by
 * Texas Instruments is distributed as "freeware". You may use
 * and modify this software without any charge or restriction.
 * You may distribute to others, as long as the original author
 * is acknowledged.
 *
 ***************************************************************
 *
 * Project: MSP430 Bootstrap Loader Demonstration Program
 *
 * File:    BSLCOMM.C
 *
 * History:
 *    Version 1.00 (05/2000)
 *    Version 1.11 (09/2000)
 *       - Added handling of frames with odd starting address
 *         (This is required for bootstrap loaders with word
 *         programming algorithm! > BSL-Version >= 1.30)
 *       - Usage of BSL_CRITICAL_ADDR for warnings about memory
 *         accesses due to "BSL checksum bug".
 *
 ***************************************************************/
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include "bslcomm.h"
#include "ssp.c"
#define BSL_SYNC 0x80
/* 1: Warning, if access to memory below 0x1000 is possible.
 *    This can happen due to an error in the first version(s) of
 *    the bootstrap loader code in combination with specific
 *    checksum values.
 * 0: No Warning.
 */
```

```
int BSLMemAccessWarning= 0; /* Default: no warning. */
/*------------------------------------------------------------*/
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 */
/*------------------------------------------------------------*/
void bslReset(BOOL invokeBSL)
/* Applies BSL entry sequence on RST/NMI and TEST/VPP pins
 * Parameters: invokeBSL = TRUE:  complete sequence
 *             invokeBSL = FALSE: only RST/NMI pin accessed
 *
 * RST is inverted twice on boot loader hardware
 * TEST is inverted (only once)
 * Need positive voltage on DTR, RTS for power-supply of hardware
 */
{
  /* 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 */
/*----------------------------------------------------------------*/
int bslSync()
/* Transmits Synchronization character and expects to
 * receive Acknowledge character
 * Return == 0: OK
 * Return == 1: Sync. failed.
 */
{
  BYTE  ch;
  int rxCount, loopcnt;
  const BYTE cLoopOut = 3; /* Max. trials to get synchronization */
  DWORD NrTx;
  DWORD NrRx;

  for (loopcnt=0; loopcnt < cLoopOut; loopcnt++)
  {
    PurgeComm(hComPort, PURGE_RXCLEAR); /* Clear receiving queue */
    /* Send synchronization byte: */
    ch = BSL_SYNC;
    WriteFile(hComPort, &ch, 1, &NrTx, NULL);
    /* Wait for 1 byte; Timeout: 100ms */
    rxCount= comWaitForData(1, 100);
    if (rxCount > 0)
    {
      ReadFile(hComPort, &ch, 1, &NrRx, NULL);
      if (ch == DATA_ACK)
      { return(ERR_NONE); } /* Sync. successful */
    }
  } /* for (loopcount) */

  return(ERR_BSL_SYNC); /* Sync. failed */
} /* bslSync */
/*----------------------------------------------------------------*/
int bslTxRx(BYTE cmd, WORD addr, WORD len,
            BYTE* blkout, BYTE* blkin)
/* Transmits a command (cmd) with its parameters:
 * start-address (addr), length (len) and additional
 * data (blkout) to boot loader.
 * Parameters return by boot loader are passed via blkin.
 * Return == 0: OK
 * Return != 0: Error!
 */
{
    BYTE dataOut[MAX_FRAME_SIZE];
    int error;
    WORD length= 4;
//    /* Make sure that len is even, when sending data to BSL: */
//    if ((cmd == BSL_TXBLK) && ((len % 2) != 0))
//    { /* Inc. len and fill blkout with 0xFF
//       * => even number of bytes to send!
//       */
//      blkout[(len++)]= 0xFF;
```

```
//      }
        if (cmd == BSL_TXBLK)
        {
          /* Align to even start address */
          if ((addr % 2) != 0)
          {
            /* Decrement address and                    */
            addr--;
            /* fill first byte of blkout with 0xFF */
            memmove(&blkout[1], &blkout[0], len);
            blkout[0]= 0xFF;
            len++;
          }
          /* Make sure that len is even */
          if ((len % 2) != 0)
          {
            /* Inc. len and fill last byte of blkout with 0xFF */
            blkout[(len++)]= 0xFF;
          }
        }
//      /* Make sure that len is even, if receiving data from BSL: */
//      if ((cmd == BSL_RXBLK) && ((len % 2) != 0))
//      {
//        len++;
//      }
        if (cmd == BSL_RXBLK)
        {
          /* Align to even start address */
          if ((addr % 2) != 0)
          {
            /* Decrement address but        */
            addr--;
            /* request an additional byte. */
            len++;
          }
          /* Make sure that len is even */
          if ((len % 2) != 0)
          {
            len++;
          }
        }
        if ((cmd == BSL_TXBLK) || (cmd == BSL_TXPWORD))
        {
          length = len + 4;
        }
        /* Add necessary information data to frame: */
        dataOut[0] =  (BYTE)( addr        & 0x00ff);
        dataOut[1] =  (BYTE)((addr >> 8)  & 0x00ff);
        dataOut[2] =  (BYTE)( len         & 0x00ff);
        dataOut[3] =  (BYTE)((len  >> 8)  & 0x00ff);

        if (blkout != NULL)
        { /* Copy data out of blkout into frame: */
```

TEXAS
INSTRUMENTS

```
            memcpy(&dataOut[4], blkout, len);
        }

        if (bslSync() != ERR_NONE)
        {
            return(ERR_BSL_SYNC);
        }
        /* Send frame: */
        error = comTxRx(cmd, dataOut, (BYTE)length);
        if (blkin != NULL)
        { /* Copy received data out of frame buffer into blkin: */
            memcpy(blkin, &rxFrame[4], rxFrame[2]);
        }
        return (error);
}
/* EOF */
```

A.4   Serial Communication Header File—ssp.h

```
/****************************************************************
*
* Copyright (C) 1999-2000 Texas Instruments, Inc.
* Author: Volker Rzehak
*
*---------------------------------------------------------------
* All software and related documentation is provided "AS IS" and
* without warranty or support of any kind and Texas Instruments
* expressly disclaims all other warranties, express or implied,
* including, but not limited to, the implied warranties of
* merchantability and fitness for a particular purpose. Under no
* circumstances shall Texas Instruments be liable for any
* incidental, special or consequential damages that result from
* the use or inability to use the software or related
* documentation, even if Texas Instruments has been advised of
* the liability.
*
* Unless otherwise stated, software written and copyrighted by
* Texas Instruments is distributed as "freeware". You may use
* and modify this software without any charge or restriction.
* You may distribute to others, as long as the original author
* is acknowledged.
*
****************************************************************/
#ifndef SSP__H
#define SSP__H
#include <windows.h>
#define MODE_SSP 0
#define MODE_BSL 1
/* Error Codes:
 */
/* No Error:               */
#define ERR_NONE            0
/* Unspecific error:       */
#define ERR_COM             1
```

```
/* OpenComm failed:             */
#define ERR_OPEN_COMM          2
/* SetCommState failed:         */
#define ERR_SET_COMM_STATE     3
/* Synchronisation failed:      */
#define ERR_SYNC_FAILED        4
/* Unspecific error concerning transmission of command: */
#define ERR_SEND_COMMAND       5
/* Timeout while receiving header": */
#define ERR_RX_HDR_TIMEOUT     6
/* NAK received:                */
#define ERR_RX_NAK             7
/* Command did not send ACK: indicates that it didn't complete correctly: */
#define ERR_CMD_NOT_COMPLETED  8
/* Command failed, is not defined or is not allowed: */
#define ERR_CMD_FAILED         9
/* CloseComm failed:            */
#define ERR_CLOSE_COMM         10
/* Header Definitions: */
#define CMD_FAILED        0x70
#define DATA_FRAME        0x80
#define DATA_ACK          0x90
#define DATA_NAK          0xA0
#define QUERY_POLL        0xB0
#define QUERY_RESPONSE    0x50
#define OPEN_CONNECTION   0xC0
#define ACK_CONNECTION    0x40
#define DEFAULT_TIMEOUT 300
#define DEFAULT_PROLONG  10
#define MAX_FRAME_SIZE 256
#define MAX_DATA_BYTES 250
#define MAX_DATA_WORDS 125
#ifdef __cplusplus
extern "C" {
#endif
/*---------------------------------------------------------------
 * Support Subroutines:
 *---------------------------------------------------------------
 */
/*------------------------------------------------------------*/
extern DWORD calcTimeout(DWORD startTime);
/* Calculates the difference between startTime and the actual
 * windows time (in milliseconds).
 */
/*------------------------------------------------------------*/
extern void delay(DWORD time);
/* Delays the execution by a given time in ms.
 */
/*------------------------------------------------------------*/
extern int comWaitForData(int count, DWORD timeout);
/* Waits until a given number (count) of bytes was received or a
 * given time (timeout) has passed.
 */
```

```
extern void comTxHeader(const BYTE txHeader);
/*----------------------------------------------------------------
 * Communication Subroutines:
 *----------------------------------------------------------------
 */
extern int comGetLastError();
/* Returns the error code generated by the last function call to
 * a SERCOMM-Function.  If this function returned without errors,
 * comGetLastError will return zero (errNoError) as well.
 */
/*--------------------------------------------------------------*/
#ifdef __cplusplus
extern int comInit(LPCSTR lpszDevice    = ”COM1”,
                   DWORD  aTimeout      = DEFAULT_TIMEOUT,
                   int    aProlongFactor= DEFAULT_PROLONG);
#else
extern int comInit(LPCSTR lpszDevice,
                   DWORD aTimeout, int aProlongFactor);
#endif
/* Tries to open the serial port given in ’lpszDevice’ and
 * initializes the port and global variables.
 * The timeout and the number of allowed errors is multiplied by
 * ’aProlongFactor’ after transmission of a command to give
 * plenty of time to the micro controller to finish the command.
 * Returns zero if the function is successful.
 */
/*--------------------------------------------------------------*/
extern int comDone();
/* Closes the used serial port.
 * This function must be called at the end of a program,
 * otherwise the serial port might not be released and can not be
 * used in other programs.
 * Returns zero if the function is successful.
 */
#ifdef __cplusplus
}
#endif
#endif
/* EOF */
```

A.5 Serial Communication Implementation File—ssp.c

```
/****************************************************************
*
* Copyright (C) 1999-2000 Texas Instruments, Inc.
* Author: Volker Rzehak
*
*----------------------------------------------------------------
* All software and related documentation is provided ”AS IS” and
* without warranty or support of any kind and Texas Instruments
* expressly disclaims all other warranties, express or implied,
* including, but not limited to, the implied warranties of
* merchantability and fitness for a particular purpose. Under no
* circumstances shall Texas Instruments be liable for any
```

```
* incidental, special or consequential damages that result from
* the use or inability to use the software or related
* documentation, even if Texas Instruments has been advised of
* the liability.
*
* Unless otherwise stated, software written and copyrighted by
* Texas Instruments is distributed as "freeware". You may use
* and modify this software without any charge or restriction.
* You may distribute to others, as long as the original author
* is acknowledged.
*
****************************************************************/
#include <string.h>
#include <stdio.h>
#include <windows.h>
#include "ssp.h"
/* Global Constants: */
/* Size of internal WINDOWS-Comm-Buffer: */
#define QUEUE_SIZE       512
#define MAX_FRAME_COUNT   16
#define MAX_ERR_COUNT      5
/* Global Variables: */
const unsigned short protocolMode= MODE_BSL;
HANDLE       hComPort;    /* COM-Port Handle             */
DCB          comDCB;      /* COM-Port Control-Settings   */
COMSTAT      comState;    /* COM-Port Status-Information */
COMMTIMEOUTS orgTimeouts; /* Original COM-Port Time-out  */
/* Time in milliseconds until a timeout occurs: */
DWORD timeout       = DEFAULT_TIMEOUT;
/* Factor by which the timeout after sending a frame is prolonged: */
int prolongFactor= DEFAULT_PROLONG;
/* Variable to save the latest error (used by comGetLastError): */
int lastError;
BYTE seqNo, reqNo, txPtr, rxPtr;
BYTE rxFrame[MAX_FRAME_SIZE];
DWORD nakDelay; /* Delay before DATA_NAK will be send */
/****************************************************************/
DWORD calcTimeout(DWORD startTime) /* exported! */
/* Calculates the difference between startTime and the actual
 * windows time (in milliseconds).
 */
{
  return((DWORD)(GetTickCount() - startTime));
}
/*--------------------------------------------------------------*/
void delay(DWORD time) /* exported! */
/* Delays the execution by a given time in ms.
 */
{
#ifndef WIN32
  DWORD startTime= GetTickCount();
  while (calcTimeout(startTime) < time);
#else
```

```
    Sleep(time);
#endif
}
/*------------------------------------------------------------*/
WORD calcChecksum(BYTE data[], WORD length)
/* Calculates a checksum of "data".
 */
{
  WORD* i_data;
  WORD checksum= 0;
  BYTE i= 0;
  i_data= (WORD*)data;
  for (i= 0; i < length/2; i++)
  {
    checksum^= i_data[i];    /* xor-ing   */
  }
  return(checksum ^ 0xffff); /* inverting */
}
/*------------------------------------------------------------*/
int comWaitForData(int count, DWORD timeout) /* exported! */
/* Waits until a given number (count) of bytes was received or a
 * given time (timeout) has passed.
 */
{
  DWORD errors;
  int rxCount= 0;
  DWORD startTime= GetTickCount();
  do
  {
    ClearCommError(hComPort, &errors, &comState);
  } while (((rxCount= comState.cbInQue) < count) &&
           (calcTimeout(startTime) <= timeout));
  return(rxCount);
}
/*------------------------------------------------------------*/
int comRxHeader(BYTE *rxHeader, BYTE *rxNum,
                DWORD timeout)
{
  BYTE Hdr;
  DWORD dwRead;
  if (comWaitForData(1, timeout) >= 1)
  {
    ReadFile(hComPort, &Hdr, 1, &dwRead, NULL);
    *rxHeader= Hdr & 0xf0;
    *rxNum   = Hdr & 0x0f;
    if (protocolMode == MODE_BSL)
    { reqNo= 0;
      seqNo= 0;
      *rxNum= 0;
    }
    return(ERR_NONE);
  }
  else
```

```
    {
      *rxHeader= 0;
      *rxNum= 0;
      return(lastError= ERR_RX_HDR_TIMEOUT);
    }
  }
/*---------------------------------------------------------------*/
void comTxHeader(const BYTE txHeader)
{
  DWORD dwWrite;
  BYTE Hdr= txHeader;
  WriteFile(hComPort, &Hdr, 1, &dwWrite, NULL);
}
/*************************************************************/
int comGetLastError()
/* Returns the error code generated by the last function call to
 * a SERCOMM-Function.  If this function returned without errors,
 * comGetLastError will return zero (errNoError) as well.
 */
{ return(lastError); }
/*************************************************************/
int comInit(LPCSTR lpszDevice, DWORD aTimeout, int aProlongFactor)
/* Tries to open the serial port given in 'lpszDevice' and
 * initializes the port and global variables.
 * The timeout and the number of allowed errors is multiplied by
 * 'aProlongFactor' after transmission of a command to give
 * plenty of time to the micro controller to finish the command.
 * Returns zero if the function is successful.
 */
{
  COMMTIMEOUTS timeouts;
  DWORD dwCommEvents;
  /* Init. global variables: */
  seqNo= 0;
  reqNo= 0;
  rxPtr= 0;
  txPtr= 0;
  timeout= aTimeout;
  prolongFactor= aProlongFactor;

  hComPort= CreateFile(lpszDevice, GENERIC_READ | GENERIC_WRITE,
                       0, 0, OPEN_EXISTING, 0, 0);
  /* In this application the serial port is used in
   * nonoverlapped mode!
   */
  if (hComPort == INVALID_HANDLE_VALUE)
  {
    hComPort= 0;
    return (lastError= ERR_OPEN_COMM); /* Error! */
  }
  if (SetupComm(hComPort, QUEUE_SIZE, QUEUE_SIZE) == 0)
  {
    CloseHandle(hComPort);
```

```
    hComPort= 0;
  return (lastError= ERR_OPEN_COMM); /* Error! */
}
/* Save original timeout values: */
GetCommTimeouts(hComPort, &orgTimeouts);
/* Set Windows timeout values (disable build-in timeouts): */
timeouts.ReadIntervalTimeout= 0;
timeouts.ReadTotalTimeoutMultiplier= 0;
timeouts.ReadTotalTimeoutConstant= 0;
timeouts.WriteTotalTimeoutMultiplier= 0;
timeouts.WriteTotalTimeoutConstant= 0;
if (!SetCommTimeouts(hComPort, &timeouts))
{
  CloseHandle(hComPort);
    hComPort= 0;
  return (lastError= ERR_OPEN_COMM); /* Error! */
}
dwCommEvents= EV_RXCHAR | EV_TXEMPTY | EV_RXFLAG | EV_ERR;
SetCommMask(hComPort, dwCommEvents);
/* Get state and modify it: */
if (!GetCommState(hComPort, &comDCB))
{
  CloseHandle(hComPort);
    hComPort= 0;
  return (lastError= ERR_OPEN_COMM); /* Error! */
}
  comDCB.BaudRate    = CBR_9600; /* Startup-Baudrate: 9,6kBaud */
  comDCB.ByteSize    = 8;
  nakDelay= (DWORD)((11*MAX_FRAME_SIZE)/9.6);
comDCB.Parity      = EVENPARITY;
comDCB.StopBits    = ONESTOPBIT;
comDCB.fBinary     = TRUE; /* Enable Binary Transmission */
comDCB.fParity     = TRUE; /* Enable Parity Check       */
comDCB.ErrorChar   = (char)0xff;
/* Char. w/ Parity-Err are replaced with 0xff
 *(if fErrorChar is set to TRUE)
 */
comDCB.fRtsControl = RTS_CONTROL_ENABLE; /* For power supply */
comDCB.fDtrControl = DTR_CONTROL_ENABLE; /* For power supply */

comDCB.fOutxCtsFlow= FALSE;         comDCB.fOutxDsrFlow= FALSE;
comDCB.fOutX       = FALSE;         comDCB.fInX        = FALSE;
comDCB.fNull       = FALSE;
  comDCB.fErrorChar   = FALSE;
/* Assign new state: */
if (!SetCommState(hComPort, &comDCB))
{
  CloseHandle(hComPort);
    hComPort= 0;
  return(lastError= ERR_SET_COMM_STATE); /* Error! */
}
/* Clear buffers: */
PurgeComm(hComPort, PURGE_TXCLEAR | PURGE_TXABORT);
```

```
    PurgeComm(hComPort, PURGE_RXCLEAR | PURGE_RXABORT);
    return(lastError= 0);
} /* comInit */
/***********************************************************/
int comDone()
/* Closes the used serial port.
 * This function must be called at the end of a program,
 * otherwise the serial port might not be released and can not be
 * used in other programs.
 * Returns zero if the function is successful.
 */
{
  DWORD errors;
  DWORD startTime= GetTickCount();
  /* Wait until data is transmitted, but not too long... (Timeout-Time) */
  do
  {
    ClearCommError(hComPort, &errors, &comState);
  } while ((comState.cbOutQue > 0) &&
           (calcTimeout(startTime) < timeout));
  /* Clear buffers: */
  PurgeComm(hComPort, PURGE_TXCLEAR | PURGE_TXABORT);
  PurgeComm(hComPort, PURGE_RXCLEAR | PURGE_RXABORT);
  /* Restore original timeout values: */
  SetCommTimeouts(hComPort, &orgTimeouts);
  /* Close COM-Port: */
  if (!CloseHandle(hComPort))
    return(lastError= ERR_CLOSE_COMM); /* Error! */
  else
    return(lastError= ERR_NONE);
} /* comDone */
/***********************************************************/
/*---------------------------------------------------------*/
int comRxFrame(BYTE *rxHeader, BYTE *rxNum)
{
  DWORD dwRead;
  WORD checksum;
  BYTE* rxLength;
  WORD rxLengthCRC;
  rxFrame[0]= DATA_FRAME | *rxNum;
  if (comWaitForData(3, timeout) >= 3)
  {
    ReadFile(hComPort, &rxFrame[1], 3, &dwRead, NULL);

    if ((rxFrame[1] == 0) && (rxFrame[2] == rxFrame[3]))
    {
      rxLength= &rxFrame[2];       /* Pointer to rxFrame[2]   */
      rxLengthCRC= *rxLength + 2;  /* Add CRC-Bytes to length */

      if (comWaitForData(rxLengthCRC, timeout) >= rxLengthCRC)
      {
        ReadFile(hComPort, &rxFrame[4], rxLengthCRC, &dwRead, NULL);
        /* Check received frame: */
```

```
          checksum= calcChecksum(rxFrame, (WORD)(*rxLength+4));
                    /* rxLength+4: Length with header but w/o CRC */
          if ((rxFrame[*rxLength+4] == (BYTE)checksum) &&
              (rxFrame[*rxLength+5] == (BYTE)(checksum >> 8)))
          {
            return(ERR_NONE);
            /* Frame received correctly (=> send next frame) */
          } /* if (Checksum correct?)        */
        } /* if (Data: no timeout?)          */
      } /* if (Add. header info. correct?)   */
    } /* if (Add. header info.: no timeout?) */
    return(ERR_COM); /* Frame has errors! */
  }  /* comRxFrame */
  /*-------------------------------------------------------------*/
  int comTxRx(BYTE cmd, BYTE dataOut[], BYTE length)
  /* Sends the command cmd with the data given in dataOut to the
   * microcontroller and expects either an acknowledge or a frame
   * with result from the microcontroller.  The results are stored
   * in dataIn (if not a NULL pointer is passed).
   * In this routine all the necessary protocol stuff is handled.
   * Returns zero if the function was successful.
   */
  {
    DWORD dwWrite;
    DWORD errors;
    BYTE txFrame[MAX_FRAME_SIZE];
    WORD checksum= 0;
    int k= 0;
    int errCtr= 0;
    int resendCtr= 0;
    BYTE rxHeader= 0;
    BYTE rxNum= 0;
    int resentFrame= 0;
    int pollCtr= 0;
    /* Transmitting part -----------------------------------------*/
    /* Prepare data for transmit */
    if ((length % 2) != 0)
    { /* Fill with one byte to have even number of bytes to send */
      if (protocolMode == MODE_BSL)
        dataOut[length++]= 0xFF; // fill with 0xFF
      else
        dataOut[length++]= 0;    // fill with zero
    }
    txFrame[0]= DATA_FRAME | seqNo;
    txFrame[1]= cmd;
    txFrame[2]= length;
    txFrame[3]= length;
    reqNo= (seqNo + 1) % MAX_FRAME_COUNT;
    memcpy(&txFrame[4], dataOut, length);
    checksum= calcChecksum(txFrame, (WORD)(length+4));
    txFrame[length+4]= (BYTE)(checksum);
    txFrame[length+5]= (BYTE)(checksum >> 8);
    {
```

```
      WORD accessAddr= (0x0212 + (checksum^0xffff)) & 0xfffe;
                          /* 0x0212: Address of wCHKSUM */
      if (BSLMemAccessWarning && (accessAddr < BSL_CRITICAL_ADDR))
      {
        printf("WARNING: This command might change data "
               "at address %x or %x!\n",
               accessAddr, accessAddr + 1);
      }
    }
    /* Transmit data: */
    k= 0;
    /* Clear receiving queue: */
    PurgeComm(hComPort, PURGE_RXCLEAR | PURGE_RXABORT);
    do
    {
      WriteFile(hComPort, &txFrame[k++], 1, &dwWrite, NULL);

      ClearCommError(hComPort, &errors, &comState);
    } while ((k < length + 6) && (comState.cbInQue == 0));
    /* Check after each transmitted character,
     * if microcontroller did send a character (probably a NAK!).
     */
    /* Receiving part ------------------------------------------*/
    rxFrame[2]= 0;
    rxFrame[3]= 0; /* Set lengths of received data to 0! */
    do
    {
      lastError= 0; /* Clear last error */
      if (comRxHeader(&rxHeader, &rxNum, timeout*prolongFactor) == 0)
          /* prolong timeout to allow execution of sent command */
      { /* => Header received */
        do
        {
          resentFrame= 0;
          switch (rxHeader)
          { case DATA_ACK:
                if (rxNum == reqNo)
                { seqNo= reqNo;
                  return(lastError= ERR_NONE);
                  /* Acknowledge received correctly => next frame */
                }
            break; /* case DATA_ACK */
            case DATA_NAK:
                return(lastError= ERR_RX_NAK);
            break; /* case DATA_NAK */
            case DATA_FRAME:
              if (rxNum == reqNo)
                if (comRxFrame(&rxHeader, &rxNum) == 0)
                  return(lastError= ERR_NONE);
            break; /* case DATA_FRAME */
            case CMD_FAILED:
              /* Frame ok, but command failed. */
              return(lastError= ERR_CMD_FAILED);
```

```
              break; /* case CMD_FAILED */
              default:
                  ;
          } /* switch */

              errCtr= MAX_ERR_COUNT;
         } while ((resentFrame == 0) && (errCtr < MAX_ERR_COUNT));
      } /* if (comRxHeader) */
      else
      { /* => Timeout while receiving header */
          errCtr= MAX_ERR_COUNT;
      } /* else (comRxHeader) */
    } while (errCtr < MAX_ERR_COUNT);
    if (lastError == ERR_CMD_NOT_COMPLETED)
    { /* Accept QUERY_RESPONSE as real ACK and correct Seq.-No.: */
      seqNo= reqNo;
    }
    if (lastError == ERR_NONE)
      return(lastError= ERR_COM);
    else
      return(lastError);
} /* comTxRx */
/* EOF */
```

A.6 Bootstrap Loader Demonstration Program—bsldemo.c

```
/***************************************************************
*
* Copyright (C) 1999-2000 Texas Instruments, Inc.
* Author: Volker Rzehak
*
*-----------------------------------------------------------------
* All software and related documentation is provided "AS IS" and
* without warranty or support of any kind and Texas Instruments
* expressly disclaims all other warranties, express or implied,
* including, but not limited to, the implied warranties of
* merchantability and fitness for a particular purpose. Under no
* circumstances shall Texas Instruments be liable for any
* incidental, special or consequential damages that result from
* the use or inability to use the software or related
* documentation, even if Texas Instruments has been advised of
* the liability.
*
* Unless otherwise stated, software written and copyrighted by
* Texas Instruments is distributed as "freeware". You may use
* and modify this software without any charge or restriction.
* You may distribute to others, as long as the original author
* is acknowledged.
*
***************************************************************
*
* Project: MSP430 Bootstrap Loader Demonstration Program
*
* File:    BSLDEMO.C
```

```
*
* Description:
*    This is the main program of the bootstrap loader
*    demonstration.
*    The main function holds the general sequence to access the
*    bootstrap loader and program/verify a file.
*    The parsing of the TI TXT file is done in a separate
*    function.
*
*    A couple of parameters can be passed to the program to
*    control its functions.  For a detailed description see the
*    appendix of the corresponding application note.
*
* History:
*    Version 1.00 (05/2000)
*    Version 1.10 (08/2000)
*       - Help screen added.
*       - Additional mass erase cycles added
*         (Required for larger flash memories)
*         Defined with: #define ADD_MERASE_CYCLES 20
*       - Possibility to load a completely new BSL into RAM
*         (supposing there is enough RAM) – Mainly a test feature!
*       - A new workaround method to cope with the checksum bug
*         established. Because this workaround is incompatible with
*         the former one the required TI TXT file is renamed to
*         "PATCH.TXT".
*    Version 1.11 (09/2000)
*       - Added handling of frames with odd starting address
*         to BSLCOMM.C. (This is required for loaders with word
*         programming algorithm! > BSL–Version >= 1.30)
*       - Changed default number of data bytes within one frame
*         to 240 bytes (old: 64). Speeds up programming.
*       - Always read BSL version number (even if new one is loaded
*         into RAM.
*       - Fixed setting of warning flag in conjunction with loading
*         a new BSL into RAM.
*       - Added a byte counter to the programTIText function.
*       - Number of mass erase cycles can be changed via command
*         line option (–m)
*    Version 1.12 (09/2000)
*       - Minor fixes and cosmetics.
*
*****************************************************************/
#include <string.h>
#include <stdio.h>
#include <windows.h>
#include "bslcomm.h"
/*-------------------------------------------------------------
 * Defines:
 *-------------------------------------------------------------
 */
/* This definition includes code to load a new BSL into RAM:
 * NOTE: Can only be used with devices with sufficient RAM!
```

```
 * The program flow is changed slightly compared to a version
 * without "NEW_BSL" defined.
 * The definition also defines the filename of the TI-TXT file
 * with the new BSL code.
 */
#define NEW_BSL
/* The "WORKAROUND" definition includes code for a workaround
 * required by the first version(s) of the bootstrap loader.
 */
#define WORKAROUND
/* If "DEBUG" is defined, all checked and programmed blocks are
 * logged on the screen.
 */
//#define DEBUG
/* Additional mass erase cycles required for (some) F149 devices.
 * If ADD_MERASE_CYCLES is not defined only one mass erase
 * cycle is executed.
 * Remove #define for fixed F149 or F11xx devices.
 */
#define ADD_MERASE_CYCLES  20
/* Error: verification failed:        */
#define ERR_VERIFY_FAILED      98
/* Error: erase check failed:         */
#define ERR_ERASE_CHECK_FAILED 97
/* Error: unable to open input file: */
#define ERR_FILE_OPEN          96
/* Mask: program data:    */
#define ACTION_PROGRAM 0x01
/* Mask: verify data:     */
#define ACTION_VERIFY  0x02
/* Mask: erase check:     */
#define ACTION_ERASE_CHECK 0x04
/* Mask: transmit password:  */
/* Note: Should not be used in conjunction with any other action! */
#define ACTION_PASSWD      0x08
/*----------------------------------------------------------------
 * Global Variables:
 *----------------------------------------------------------------
 */
char *programName=    "MSP430 Bootstrap Loader Demonstration Program";
char *programVersion= "Version 1.12";
/* Max. bytes sent within one frame if parsing a TI TXT file.
 * ( >= 16 and == n*16 and <= MAX_DATA_BYTES!)
 */
int maxData= 240;
/* Buffers used to store data transmitted to and received from BSL: */
BYTE blkin [MAX_DATA_BYTES]; /* Receive buffer  */
BYTE blkout[MAX_DATA_BYTES]; /* Transmit buffer */
#ifdef WORKAROUND
char *patchFile    = "PATCH.TXT";
#endif /* WORKAROUND */
BOOL patchRequired = FALSE;
BOOL patchLoaded   = FALSE;
```

```
WORD bslVer= 0;
char *newBSLFile= NULL;
struct toDoList
{
  unsigned MassErase : 1;
  unsigned EraseCheck: 1;
  unsigned Program   : 1;
  unsigned Verify    : 1;
  unsigned Reset     : 1;
  unsigned Wait      : 1; /* Wait for <Enter> at end of program */
                          /* (0: no; 1: yes):                   */
  unsigned OnePass   : 1; /* Do EraseCheck, Program and Verify  */
                          /* in one pass (TI TXT file is read   */
                          /* only once)                         */
} toDo;
void *errData= NULL;
int byteCtr= 0;
/*-----------------------------------------------------------
 * Functions:
 *-----------------------------------------------------------
 */
int preparePatch()
{
  int error= ERR_NONE;
#ifdef WORKAROUND
  if (patchLoaded)
  {
    /* Load PC with 0x0220.
     * This will invoke the patched bootstrap loader subroutines.
     */
    error= bslTxRx(BSL_LOADPC,  /* Command: Load PC         */
                   0x0220,      /* Address to load into PC */
                   0,           /* No additional data!     */
                   NULL, blkin);
    if (error != ERR_NONE) return(error);
    BSLMemAccessWarning= 0; /* Error is removed within workaround code */
  }
#endif /* WORKAROUND */
  return(error);
}
void postPatch()
{
#ifdef WORKAROUND
  if (patchLoaded)
  {
    BSLMemAccessWarning= 1; /* Turn warning back on. */
  }
#endif /* WORKAROUND */
}
int verifyBlk(WORD addr, WORD len, unsigned action)
{
  int i= 0;
  int error= ERR_NONE;
```

```
    if ((action & (ACTION_VERIFY | ACTION_ERASE_CHECK)) != 0)
    {
#ifdef DEBUG
    printf("Check starting at %x, %i bytes... ", addr, len);
#endif /* DEBUG */
    error= preparePatch();
    if (error != ERR_NONE) return(error);
    error= bslTxRx(BSL_RXBLK, addr, len, NULL, blkin);
    postPatch();
#ifdef DEBUG
    printf("Error: %i\n", error);
#endif /* DEBUG */
    if (error != ERR_NONE)
    {
      return(error); /* Cancel, if read error */
    }
    else
    {
      for (i= 0; i < len; i++)
      {
        if ((action & ACTION_VERIFY) != 0)
        {
          /* 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! */
          }
          continue;
        }
        if ((action & ACTION_ERASE_CHECK) != 0)
        {
          /* Compare data in blkin with erase pattern: */
          if (blkin[i] != 0xff)
          {
            printf("Erase Check failed at %x (%x)\n", addr+i, blkin[i]);
            return(ERR_ERASE_CHECK_FAILED); /* Erase Check failed! */
          }
          continue;
        } /* if ACTION_ERASE_CHECK */
      } /* for (i) */
    } /* else */
  } /* if ACTION_VERIFY | ACTION_ERASE_CHECK */
  return(error);
}
int programBlk(WORD addr, WORD len, unsigned action)
{
  int i= 0;
  int error= ERR_NONE;
  if ((action & ACTION_PASSWD) != 0)
  {
    return(bslTxRx(BSL_TXPWORD, /* Command: Transmit Password  */
```

```
                     addr,          /* Address of interrupt vectors */
                     len,           /* Number of bytes             */
                     blkout, blkin));
  } /* if ACTION_PASSWD */
  /* Check, if specified range is erased: */
  error= verifyBlk(addr, len, action & ACTION_ERASE_CHECK);
  if (error != ERR_NONE)
  {
    return(error);
  }

  if ((action & ACTION_PROGRAM) != 0)
  {
#ifdef DEBUG
    printf("Program starting at %x, %i bytes... ", addr, len);
#endif /* DEBUG */
    error= preparePatch();
    if (error != ERR_NONE) return(error);
    /* Program block: */
    error= bslTxRx(BSL_TXBLK, addr, len, blkout, blkin);
    postPatch();
#ifdef DEBUG
    printf("Error: %i\n", error);
#endif /* DEBUG */
    if (error != ERR_NONE)
    {
      return(error); /* Cancel, if error (ACTION_VERIFY is skipped!) */
    }
  } /* if ACTION_PROGRAM */
  /* Verify block: */
  error= verifyBlk(addr, len, action & ACTION_VERIFY);
  if (error != ERR_NONE)
  {
    return(error);
  }

  return(error);
} /* programBlk */
int programTIText (char *filename, unsigned action)
{
  int next= 1;
  int error= ERR_NONE;
  int linelen= 0;
  int linepos= 0;
  WORD dataframelen=0;
  WORD currentAddr;
  char strdata[128];
  FILE* infile;
  byteCtr= 0;
  if ((infile = fopen(filename, "rb")) == 0)
  {
    errData= filename;
    return(ERR_FILE_OPEN);
```

```
    }
    /* Convert data for MSP430, TXT-File is parsed line by line: */
    for (next= 1; next>=1; )
    {
      /* Read one line: */
      if ((fgets(strdata, 127, infile) == 0) ||
          /* if End Of File                      or */
          (strdata[0] == 'q'))
          /* if q (last character in file)        */
      {    /* => send frame and quit              */
        if (dataframelen > 0) /* Data in frame? */
        {
          error= programBlk(currentAddr, dataframelen, action);
          byteCtr+= dataframelen; /* Byte Counter */
          dataframelen=0;
        }
        next=0;  /* Quit! */
        continue;
      }
      linelen= strlen(strdata);
      if (strdata[0] == '@')
         /* if @ => new address => send frame and set new addr. */
      {
        if (dataframelen > 0)
        {
          error= programBlk(currentAddr, dataframelen, action);
          byteCtr+= dataframelen; /* Byte Counter */
          dataframelen=0;
        }
        sscanf(&strdata[1], "%lx\n", &currentAddr);
        continue;
      }
      /* Transfer data in line into blkout: */
      for(linepos= 0;
          linepos < linelen-3; linepos+= 3, dataframelen++)
      {
        sscanf(&strdata[linepos], "%3x", &blkout[dataframelen]);
        /* (Max 16 bytes per line!) */
      }
      if (dataframelen > maxData-16)
         /* if frame is getting full => send frame */
      {
        error= programBlk(currentAddr, dataframelen, action);
        byteCtr+= dataframelen; /* Byte Counter */
        currentAddr+= dataframelen;
        dataframelen=0;
      }
      if (error != ERR_NONE)
      {
        next=0; /* Cancel loop, if any error */
      }
    }
    fclose(infile);
```

```
    return(error);
} /* programTIText */
int txPasswd(char* passwdFile)
{
  int i;
  if (passwdFile == NULL)
  {
    /* Send "standard" password to get access to protected functions. */
    printf("Transmit Password...\n");
    /* Fill blkout with 0xff
     * (Flash is completely erased, the contents of all Flash cells is 0xff)
     */
    for (i= 0; i < 0x20; i++)
    {
      blkout[i]= 0xff;
    }
    return(bslTxRx(BSL_TXPWORD, /* Command: Transmit Password  */
                   0xffe0,       /* Address of interupt vectors */
                   0x0020,       /* Number of bytes             */
                   blkout, blkin));
  }
  else
  {
    /* Send TI TXT file holding interrupt vector data as password: */
    printf("Transmit password file \"%s\"...\n", passwdFile);
    return(programTIText(passwdFile, ACTION_PASSWD));
  }
} /* txPasswd */
int signOff(int error, BOOL passwd)
{
  if (toDo.Reset)
  {
    bslReset(0); /* Reset MSP430 and start user program. */
  }
  switch (error)
  {
    case ERR_NONE:
      printf("Programming completed!\n");
      break;
    case ERR_BSL_SYNC:
      printf("ERROR: Synchronization failed!\n");
      printf("Device with boot loader connected?\n");
      break;
    case ERR_VERIFY_FAILED:
      printf("ERROR: Verification failed!\n");
      break;
    case ERR_ERASE_CHECK_FAILED:
      printf("ERROR: Erase check failed!\n");
      break;
    case ERR_FILE_OPEN:
      printf("ERROR: Unable to open input file \"%s\"!\n", (char*)errData);
      break;
    default:
```

```
        if ((passwd) && (error == ERR_RX_NAK))
           /* If last command == transmit password && Error: */
         printf("ERROR: Password not accepted!\n");
       else
         printf("ERROR: Communication Error!\n");
   } /* switch */
   if (toDo.Wait)
   {
     printf("Press <ENTER> ...\n"); getchar();
   }
   comDone();   /* Release serial communication port.   */
   /* After having released the serial port,
    * the target is no longer supplied via this port!
    */
   if (error == ERR_NONE)
     return(0);
   else
     return(1);
} /* signOff */
void showHelp()
{
   char *help[]=
   {
     "BSLDEMO [-h] [-c{port}] [-p{file}] [-w] [-1] [-m{num}] [+ecpvrw] {file}",
     "",
     "The last parameter is required: file name of TI-TXT file to be
      programmed.",
     "",
     "Options:",
     "   -h        Shows this help screen.",
     "   -c{port}  Specifies the communication port to be used (e.g. -cCOM2).",
     "   -p{file}  Specifies a TI-TXT file with the interrupt vectors that are",
     "             used as password (e.g. -pINT_VECT.TXT).",
     "   -w        Waits for <ENTER> before closing serial port.",
     "   -1        Programming and verification is done in one pass through the
                   file.",
/*
     "   -a{file}  Filename of workaround patch (e.g. -aWAROUND.TXT).",
     "   -b{file}  Filename of complete loader to be loaded into RAM (e.g. -
         bBSL.TXT).",
     "   -f{num}   Max. number of data bytes within one transmitted frame (e.g.
         -f240).",
*/
#ifdef ADD_MERASE_CYCLES
     "   -m{num}   Number of mass erase cycles (e.g. -m20).",
#endif /* ADD_MERASE_CYCLES */
     "",
     "Program Flow Specifiers [+ecpvrw]",
     "   e         Mass Erase",
     "   c         Erase Check by file {file}",
     "   p         Program file {file}",
     "   v         Verify by file {file}",
     "   r         Reset connected MSP430. Starts application.",
```

```
       "   w          Wait for <ENTER> before closing serial port.",
       "  Only the specified actions are executed!",
       "Default Program Flow Specifiers (if not explicitly given): +ecpvr",
       "\0" /* Marks end of help! */
    };
    int i= 0;
    while (help[i] != "\0") printf("%s\n", help[i++]);
}
/*---------------------------------------------------------------
 * Main:
 *---------------------------------------------------------------
 */
int main(int argc, char *argv[])
{
    int error= ERR_NONE;
    int i, j;
    char comPortName[10]= "COM1"; /* Default setting. */
    char *filename= NULL;
    char *passwdFile= NULL;
#ifdef ADD_MERASE_CYCLES
    int meraseCycles= ADD_MERASE_CYCLES;
#else
    const int meraseCycles= 1;
#endif /* ADD_MERASE_CYCLES */
#ifdef NEW_BSL
    newBSLFile= NULL;
#endif /* NEW_BSL */
    /* Default: all actions turned on: */
    toDo.MassErase = 1;
    toDo.EraseCheck= 1;
    toDo.Program   = 1;
    toDo.Verify    = 1;
    toDo.Reset     = 1;
    toDo.Wait      = 0; /* Do not wait for <Enter> at the end! */
    toDo.OnePass   = 0; /* Do erase check, program and verify  */
                        /* sequential!                         */
#ifdef WORKAROUND
    /* Show memory access warning, if working with bootstrap
     * loader version(s) requiring the workaround patch.
     * Turn warning on by default until we can determine the
     * actual version of the bootstrap loader.
     */
    BSLMemAccessWarning= 1;
#endif /* WORKAROUND */
    printf("%s (%s)\n", programName, programVersion);
    /*---------------------------------------------------------------
     * Parse Command Line Parameters ...
     *---------------------------------------------------------------
     */
    if (argc > 1)
    {
      for (i= 1; i < (argc - 1); i++)
      {
```

```
        switch (argv[i][0])
        {
          case '-':
            switch (argv[i][1])
            {
              case 'h': case 'H':
                showHelp(); /* Show help screen and */
                return(1);  /* exit program.        */
              break;
              case 'c': case 'C':
                memcpy(comPortName, &argv[i][2], strlen(argv[i])-2);
              break;
              case 'p': case 'P':
                passwdFile= &argv[i][2];
              break;
              case 'w': case 'W':
                toDo.Wait= 1; /* Do wait for <Enter> at the end! */
              break;
              case '1':
                toDo.OnePass= 1;
              break;
              case 'f': case 'F':
                if (argv[i][2] != 0)
                {
                 sscanf(&argv[i][2], "%i", &maxData);
                 /* Make sure that conditions for maxData are met:
                  * ( >= 16 and == n*16 and <= MAX_DATA_BYTES!)
                  */
                 maxData= (maxData > MAX_DATA_BYTES) ? MAX_DATA_BYTES : maxData;
                 maxData= (maxData <           16) ?          16 : maxData;
                 maxData= maxData - (maxData % 16);
                 printf("Max. number of data bytes within one frame set to
                  %i.\n",
                          maxData);
                }
              break;
#ifdef ADD_MERASE_CYCLES
              case 'm': case 'M':
                if (argv[i][2] != 0)
                {
                  sscanf(&argv[i][2], "%i", &meraseCycles);
                  meraseCycles= (meraseCycles < 1) ? 1 : meraseCycles;
                  printf("Number of mass erase cycles set to %i.\n", merase
                   Cycles);
                }
              break;
#endif /* ADD_MERASE_CYCLES */
#ifdef WORKAROUND
              case 'a': case 'A':
                patchFile= &argv[i][2];
              break;
#endif /* WORKAROUND */
#ifdef NEW_BSL
```

```
            case 'b': case 'B':
              newBSLFile= &argv[i][2];
            break;
#endif /* NEW_BSL */
            default:
              printf("ERROR: Illegal command line parameter!\n");
          } /* switch argv[i][1] */
        break; /* '-' */

        case '+':
          /* Turn all actions off: */
          toDo.MassErase = 0;
          toDo.EraseCheck= 0;
          toDo.Program   = 0;
          toDo.Verify    = 0;
          toDo.Reset     = 0;
          toDo.Wait      = 0;
          /* Turn only specified actions back on:                 */
          for (j= 1; j < (int)(strlen(argv[i])); j++)
          {
            switch (argv[i][j])
            {
            case 'e': case 'E':
              /* Erase Flash                              */
              toDo.MassErase = 1;
              break;
            case 'c': case 'C':
              /* Erase Check (by file)                    */
              toDo.EraseCheck= 1;
              break;
            case 'p': case 'P':
              /* Program file                             */
              toDo.Program   = 1;
              break;
            case 'v': case 'V':
              /* Verify file                          */
              toDo.Verify    = 1;
              break;
            case 'r': case 'R':
              /* Reset MSP430 before waiting for <Enter>     */
              toDo.Reset     = 1;
              break;
            case 'w': case 'W':
              /* Wait for <Enter> before closing serial port */
              toDo.Wait      = 1;
              break;
            default:
              printf("ERROR: Illegal action specified!\n");
            } /* switch */
          } /* for (j) */
        break; /* '+' */

        default:
```

```
          printf("ERROR: Illegal command line parameter!\n");
        } /* switch argv[i][0] */
      } /* for (i) */
      if (stricmp("-h", argv[i]) == 0)
      {
        showHelp(); /* Show help screen and */
        return(1);  /* exit program.       */
      }
      else
      {
        filename= argv[i];
      }
    }
    else
    {
      printf("ERROR: Filename required!\n");
      printf("Use -h to get help!\n");
      return(1);
    }
    /*-------------------------------------------------------
     * Communication with Bootstrap Loader ...
     *-------------------------------------------------------
     */
    /* Open COMx port (Change COM-port name to your needs!): */
    if (comInit(comPortName, DEFAULT_TIMEOUT, 4) != 0)
    {
      printf("ERROR: Opening COM-Port failed!\n");
      return(1);
    }

    bslReset(1); /* Invoke the boot loader. */

#ifdef NEW_BSL
    if ((newBSLFile == NULL) || (passwdFile == NULL))
    {
      /* If a password file is specified the "new" bootstrap loader can be
loaded
       * (if also specified) before the mass erase is performed. Than the mass
       * erase can be done using the "new" BSL. Otherwise the mass erase is done
       * now!
       */
#endif /* NEW_BSL */
    if (toDo.MassErase)
    {
      int i;
      /* Erase the flash memory completely (with mass erase command): */
      printf("Mass Erase...\n");
      for (i= 0; i < meraseCycles; i++)
      {
        if (i == 1)
        {
          printf("Additional Mass Erase Cycles...\n");
        }
```

```
          if ((error= bslTxRx(BSL_MERAS, /* Command: Mass Erase         */
                               0xff00,    /* Any address within flash memory. */
                               0xa506,    /* Required setting for mass erase! */
                               NULL, blkin)) != ERR_NONE)
          {
            return(signOff(error, FALSE));
          }
        }
        passwdFile= NULL; /* No password file required! */
    }
#ifdef NEW_BSL
  } /* if ((newBSLFile == NULL) || (passwdFile == NULL)) */
#endif /* NEW_BSL */
  /* Transmit password to get access to protected BSL functions. */
  if ((error= txPasswd(passwdFile)) != ERR_NONE)
  {
    return(signOff(error, TRUE)); /* Password was transmitted! */
  }
  /* 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);
    printf("Current bootstrap loader version: %x.%x\n", bslVerHi, bslVerLo);

    bslVer= (bslVerHi << 8) | bslVerLo;
    if (bslVer <= 0x0110)
    {
#ifdef WORKAROUND
#ifdef NEW_BSL
      if (newBSLFile == NULL)
      {
#endif /* NEW_BSL */
        printf("Patch for flash programming required!\n");
        patchRequired= TRUE;
#ifdef NEW_BSL
      }
#endif /* NEW_BSL */
#endif /* WORKAROUND */
      BSLMemAccessWarning= 1;
    }
    else
    {
      BSLMemAccessWarning= 0; /* Fixed in newer versions of BSL. */
    }
  }
  if (patchRequired || ((newBSLFile != NULL) && (bslVer <= 0x0110)))
  {
```

```
      /* Execute function within bootstrap loader
       * to prepare stack pointer for the following patch.
       * This function will lock the protected functions again.
       */
      printf("Load PC with 0x0C22...\n");
      if ((error= bslTxRx(BSL_LOADPC, /* Command: Load PC        */
                          0x0C22,     /* Address to load into PC */
                          0,          /* No additional data!     */
                          NULL, blkin)) != ERR_NONE)
      {
        return(signOff(error, FALSE));
      }
      /* Re-send password to re-gain access to protected functions. */
      if ((error= txPasswd(passwdFile)) != ERR_NONE)
      {
        return(signOff(error, TRUE)); /* Password was transmitted! */
      }
    }
#ifdef NEW_BSL
  if (newBSLFile != NULL)
  {
    printf("Load new BSL \"%s\" into RAM...\n", newBSLFile);
    if ((error= programTIText(newBSLFile, /* File to program */
                              ACTION_PROGRAM)) != ERR_NONE)
    {
      return(signOff(error, FALSE));
    }
    printf("Verify new BSL \"%s\"...\n", newBSLFile);
    if ((error= programTIText(newBSLFile, /* File to verify */
                              ACTION_VERIFY)) != ERR_NONE)
    {
      return(signOff(error, FALSE));
    }

    /* Read start vector 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);
    }
    if (error != ERR_NONE)
    {
      return(signOff(error, FALSE));
    }
    /* BSL-Bugs should be fixed within "new" BSL: */
    BSLMemAccessWarning= 0;
    patchRequired= FALSE;
```

```
      patchLoaded  = FALSE;


      /* Re-send password to re-gain access to protected functions. */
      if ((error= txPasswd(passwdFile)) != ERR_NONE)
      {
        return(signOff(error, TRUE)); /* Password was transmitted! */
      }
  }
#endif/* NEW_BSL */
#ifdef WORKAROUND
    if (patchRequired)
    {
      printf("Load and verify patch \"%s\"...\n", patchFile);
      /* Programming and verification is done in one pass.
       * The patch file is only read and parsed once.
       */
      if ((error= programTIText(patchFile, /* File to program */
                                ACTION_PROGRAM | ACTION_VERIFY)) != ERR_NONE)
      {
        return(signOff(error, FALSE));
      }
      patchLoaded= TRUE;
    }
#endif /* WORKAROUND */
#ifdef NEW_BSL
  if ((newBSLFile != NULL) && (passwdFile != NULL) && toDo.MassErase)
  {
    /* Erase the flash memory completely (with mass erase command): */
    printf("Mass Erase...\n");
    if ((error= bslTxRx(BSL_MERAS, /* Command: Mass Erase            */
                        0xff00,    /* Any address within flash memory. */
                        0xa506,    /* Required setting for mass erase! */
                        NULL, blkin)) != ERR_NONE)
    {
      return(signOff(error, FALSE));
    }
    passwdFile= NULL; /* No password file required! */
  }
#endif /* NEW_BSL*/
  if (!toDo.OnePass)
  {
    if (toDo.EraseCheck)
    {
      /* Parse file in TXT-Format and
       * check the erasure of required flash cells.
       */
      printf("Erase Check by file \"%s\"...\n", filename);
      if ((error= programTIText(filename, ACTION_ERASE_CHECK)) != ERR_NONE)
      {
        return(signOff(error, FALSE));
      }
    }
```

TEXAS
INSTRUMENTS

```c
    if (toDo.Program)
    {
      /* Parse file in TXT-Format and program data into flash memory. */
      printf("Program \"%s\"...\n", filename);
      if ((error= programTIText(filename, ACTION_PROGRAM)) != ERR_NONE)
      {
        return(signOff(error, FALSE));
      }
      else
      {
        printf("%i bytes programmed.\n", byteCtr);
      }
    }

    if (toDo.Verify)
    {
      /* Verify programmed data: */
      printf("Verify \"%s\"...\n", filename);
      if ((error= programTIText(filename, ACTION_VERIFY)) != ERR_NONE)
      {
        return(signOff(error, FALSE));
      }
    }
  }
  else
  {
    unsigned action= 0;
    if (toDo.EraseCheck)
    {
      action |= ACTION_ERASE_CHECK; printf("EraseCheck ");
    }
    if (toDo.Program)
    {
      action |= ACTION_PROGRAM;     printf("Program ");
    }
    if (toDo.Verify)
    {
      action |= ACTION_VERIFY;      printf("Verify ");
    }
    if (action != 0)
    {
      printf("\"%s\" ...\n", filename);
      error= programTIText(filename, action);
      if (error != ERR_NONE)
      {
        return(signOff(error, FALSE));
      }
      else
      {
        printf("%i bytes programmed.\n", byteCtr);
      }
    }
  }
```

```
  return(signOff(ERR_NONE, FALSE));
}
/* EOF */
```

A.7   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
```
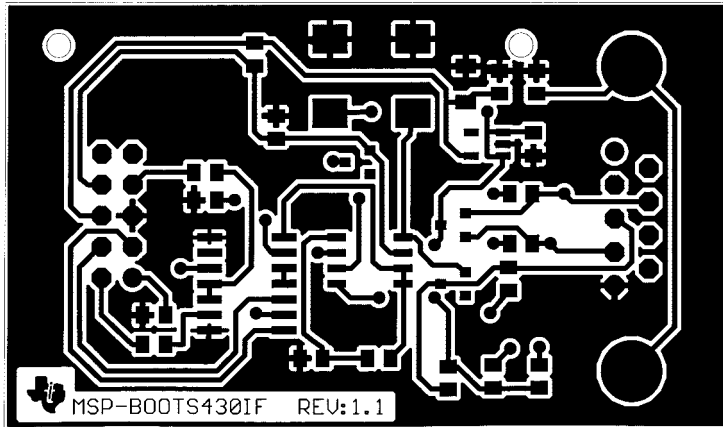
Appendix B   PCB Layout Suggestion

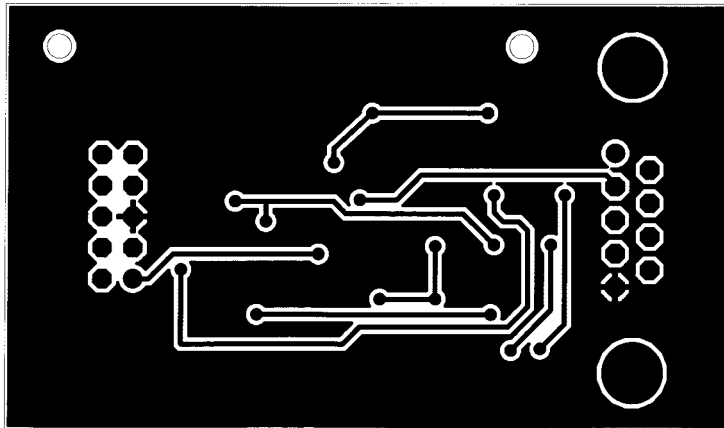**Figure B–1.   Universal BSL Interface PCB Layout—Top**

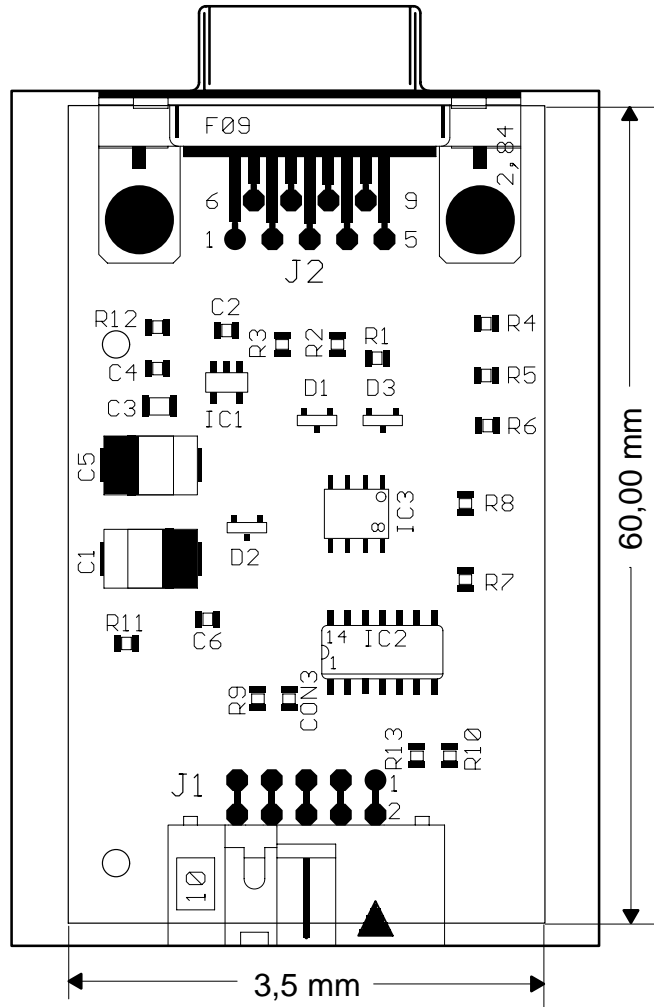**Figure B–2.   Universal BSL Interface PCB Layout—Bottom**

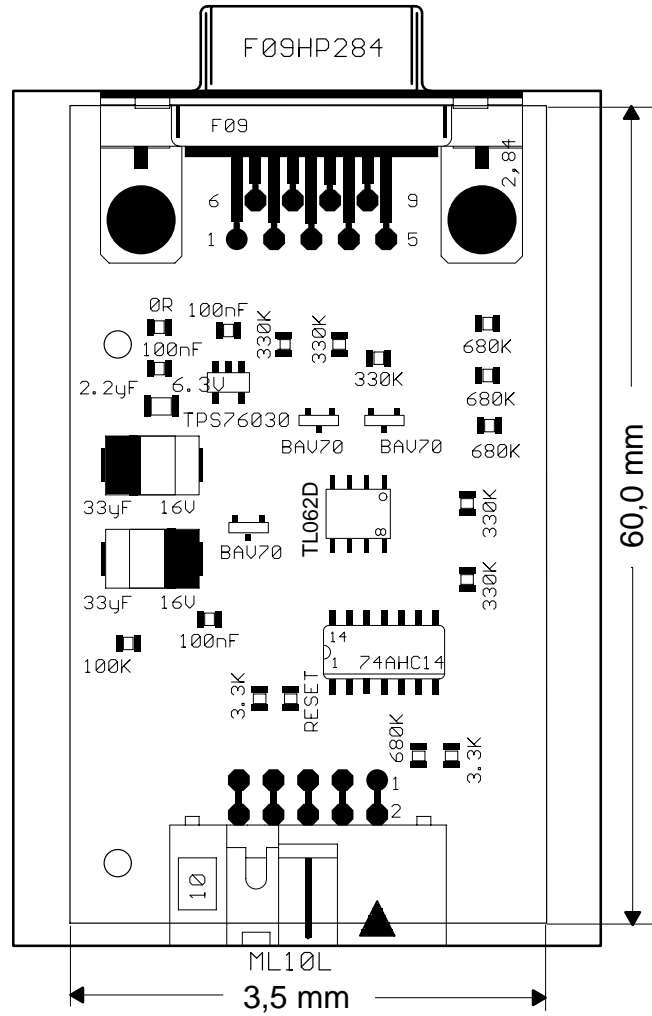**Figure B–3.   Universal BSL Interface Component Placement**

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
–m{num}	Specifies the number of mass erase cycles	–m20

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

MODIFIER	DESCRIPTION
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

NOTE: 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 some examples of the demonstration program's invocation.

Table C–3. Invocation Examples

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 needs to be enough memory to load the new loader.

NOTE: 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 note.

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

Gessler Electronic GmbH (Germany) offers a complete kit with bootstrap loader interface hardware and software: MSP430 Flash Programming Tool Kit. This information can be found by accessing the following url: http://www.gessler–electronic.de/msp430/

Gessler Electronic GmbH

Tel.: [+49]–9073–2509

Fax: [+49]–9073–3737

E-mail: info@gessler–electronic.de

Web site: www.gessler–electronic.de

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

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. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's 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, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2001, Texas Instruments Incorporated