Python Firmware Upgrader Tool for MSP430™ MCUs

MSP430 Applications

1 Introduction

With the advent of USB, end users can perform firmware upgrades in the field, simply by attaching the device via USB and executing the Python Firmware Upgrader application on the host PC. Such an approach has numerous advantages:

- Problems discovered after product release can still be fixed
- Reduced need for high-touch support, because problems can often be solved by instructing the user to upgrade firmware
- End users have a more positive experience with the product
- Product returns can be reduced

The MSP430 USB solution has been designed to make this simple and straightforward. The device contains a USB-based on-chip bootstrap loader, and TI provides a Python-based, customizable tool (the *Python Firmware Upgrader*) for communicating with it.

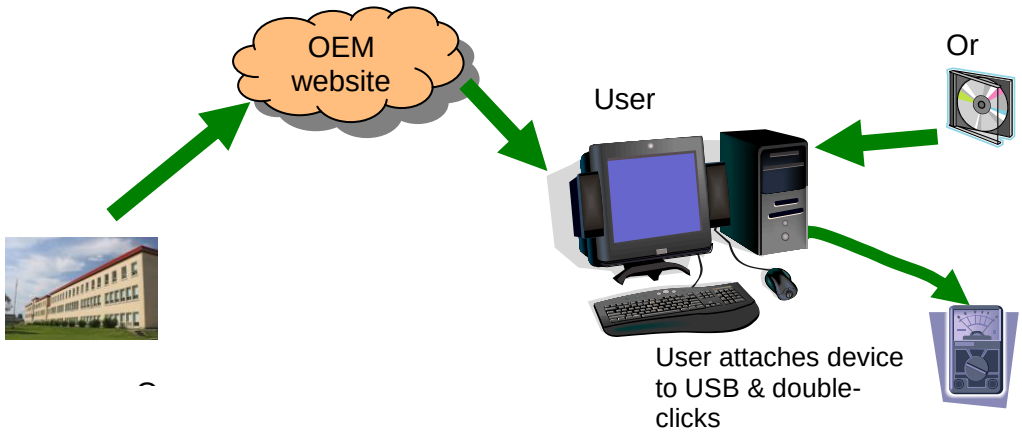

Figure 1. Firmware Upgrade Distribution Path

1.1 Relationship with Related Documentation

TI-MSP430 has two other application notes related to this one. They address the subject of USB firmware update from different angles:

- *MSP430 Programming via the Bootstrap Loader (slau319)*
- *Creating a Custom Flash-Based BSL (slaa450)*

slau319 discusses how to *interact* with the MSP430 BSL – sending commands, etc. It addresses the BSL in all MSP430 devices, most of which aren't USB-based. It serves as a reference for the BSL commands, and the Python Firmware Upgrader in this document uses these commands.

slaa450 discusses the *source* for the BSL program inside MSP430 flash, and ways you can customize it. It again is not specific to USB, and aside from the source code itself, doesn't include much detail about the USB BSL.

In contrast, this app note:

- Is specific to USB, and highlights critical items that might be missed in the other documents
- Discusses system-level considerations unique to USB BSL, which will affect the end user's experience if using the USB BSL for updates in the field
- Includes a host application/GUI that can be given to end users

slau319 and *slaa450* are useful backup references to this document, and they will be highlighted occasionally.

1.2 Python Firmware Upgrader

Previous to the release of v4.0 of the MSP430 USB Developers Package, TI made available a Windows USB firmware update application, based on Visual Studio.

Corresponding with the release of v4.0, TI now supports a tool written in Python, for cross-platform support. This application is built upon an open source toolset located at https://pypi.python.org/pypi/python-msp430-tools.

The Python Firmware Upgrader can be invoked either from a command-line, or as a GUI.

2 MSP430's USB Field Firmware Update Process: Overview

MSP430 firmware upgrade over USB is implemented using the on-chip bootstrap loader (BSL) program. The BSL is located in a dedicated section of flash, outside of main program flash. TI has always provided a BSL on nearly every MSP430 device derivative, pre-programmed in the factory; but unlike other devices, the BSL on USB-equipped MSP430s uses USB as the hardware interface. And whereas previous BSLs are invoked with a special waveform on the I/O pins, the BSL on USB-equipped devices is invoked through one of the means shown in Table 1.

Table 1. Methods By Which the USB BSL Can Be Invoked

Method	When Used
Blank RESET vector. If the RESET entry in the vector table is blank (0xFFFF) after a BOR reset (see below), the BSL is invoked.	Initial device production. (After initial programming, the vector will no longer be blank.)
Software. If the user application* transfers control of execution to the BSL (jump to 0x1000), the BSL is invoked.	Firmware upgrades in the field, initiated by the end user.
Hardware. If the PUR pin is held high by external circuitry while a BOR reset occurs on the MSP430, the BSL is invoked. Often this involves the end user pressing a button while re-inserting the battery (depending on the application).	Mostly used as a failsafe, in the rare event that the other two methods can't be used to recover from an interrupted update session. It can also be used as a primary means of invocation.

* In this document, *user application* refers to software running in main flash, located at the address pointed to by the reset vector.

The software method is recommended as the primary means of entry when performing field upgrades, because it usually provides the best experience for the end user. The other two methods can serve backup roles for failsafe operation.

A high-level overview is shown in Figure 2.

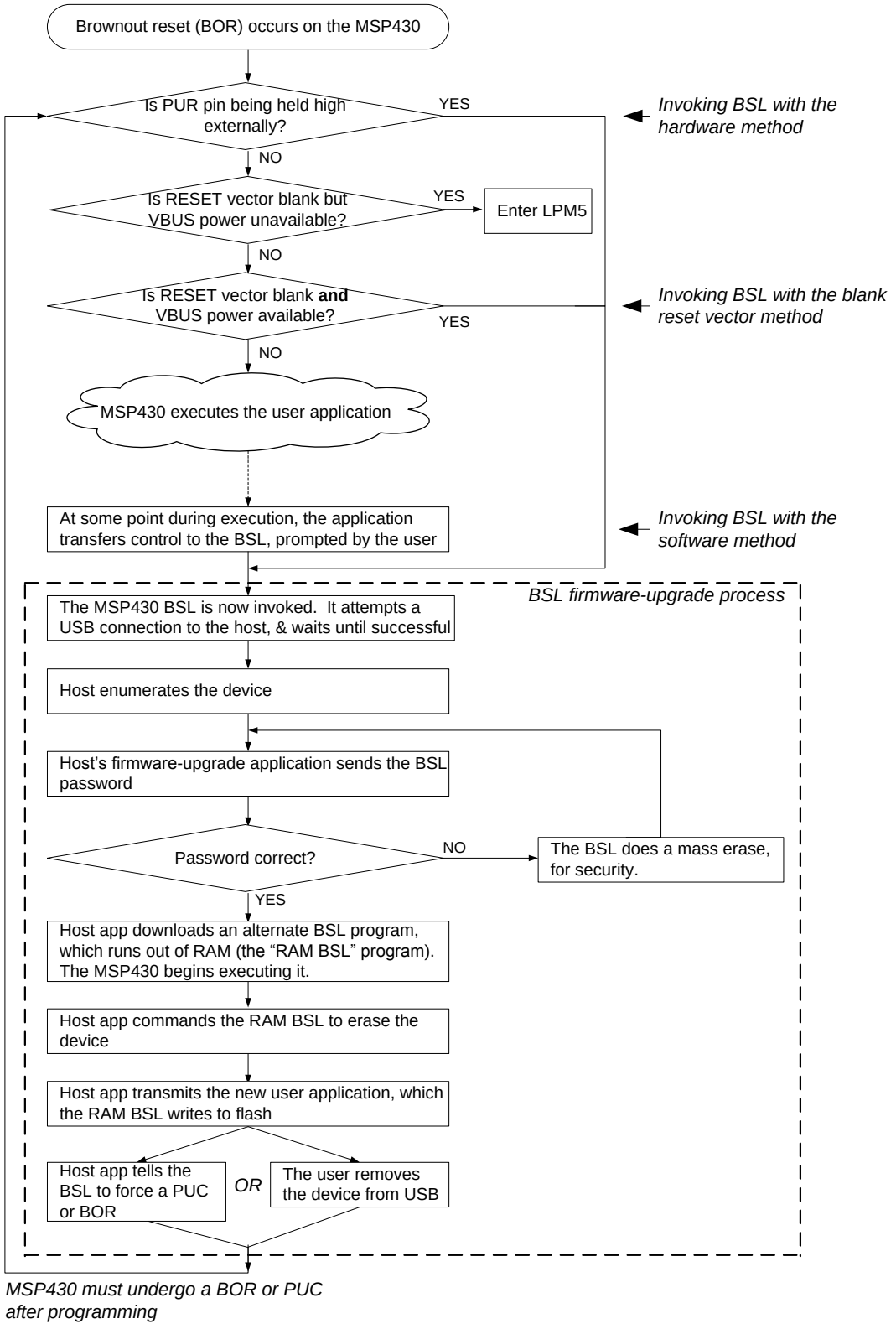

Figure 2. System-Level Description of MSP430 USB Firmware Updates

Notice the presence of a *RAM BSL* in the flowchart. The RAM BSL exists because the BSL flash space on the MSP430 is not large enough to hold both a USB stack and the BSL functionality. Therefore, the BSL in flash only holds a very minimal USB stack; its task is only to download a more complete copy of the BSL, which is downloaded into RAM, and execution is transferred to it. The RAM BSL contains both a complete USB stack and complete BSL functionality. It is the RAM BSL that writes to main flash.

An image of the RAM BSL must therefore be stored within the host application. The flash and RAM BSL's originate from the same software project, as different compiler options. Both can usually be used without the need to be changed, although there are two such circumstances discussed in this document.

The Python Firmware Upgrader tool performs all the host functions shown in the flowchart in Figure 2, and can be customized as needed. All other actions the engineer must perform are described in this document.

3 Before Implementation: System Considerations

This section addresses topics the developer should understand before beginning development.

3.1 What Host Drivers or Applications are Required?

No special kernel-mode drivers are required on the host. This is because the MSP430 reports itself as a Human Interface Device (HID), which has built-in support on all major host operating systems. HID is a great fit for firmware updates, for these reasons:

- It has perhaps the widest native support among host operating systems. In contrast, other update systems might require a kernel driver installation, which adds risk that the user will experience problems if something goes wrong.
- It loads silently onto the host – which means that no user intervention is required for "installation" of the device after it's attached.
- HID's throughput (64KB/sec) fits well with MSP430's range of memory sizes, both in existing devices and future ones. Downloading firmware finishes quickly.

HID is easier for the end user, and this can translate to less support and development costs for the OEM.

A host user-mode application is still required, for driving the download process. This is what the Python Firmware Upgrader tool provides.

3.2 Generation of a Brownout Reset (BOR) on the MSP430

As Section 2 indicated, two of the three invocation methods involve a BOR event. A BOR is also necessary for the BSL to hand control back to the new user application. Although the name BOR includes the word "brownout", an actual power brownout event is only one means by which a BOR can be triggered. BOR is simply a name for the deepest reset an MSP430F5xx can undergo.

These events can generate a BOR:

1. Initial power-up of the device
2. An edge on the RST pin (perhaps with a dedicated button)
3. Setting the PMMSWBOR bit with software
4. Removing the device from USB while the BSL is in control of the MSP430
5. A brownout event on DVCC
6. Other events. (See the SYS Module chapter of the *MSP430F5xx Family User's Guide*)

BOR events are a critical part of the BSL flow if the invocation method is the hardware or blank RESET vector methods. Some user-level events might generate a BOR "automatically" – for example, when the user removes the device from USB while the BSL is in control, or if the device is purely bus-powered and the user attaches it to the host, causing a power-up event. In other cases, the user might need to be given special instructions to cause it – for example, with a special reset button, or by removing the battery prior to attaching it to USB (causing a power-up event when attached). The system designer must ensure the end user does what is necessary to cause the BOR at the proper time.

3.3 VID/PIDs in the USB BSL

Any USB device contains a pair of 16-bit values called a vendor ID (VID) and product ID (PID). The VID is associated with the OEM, and the PID is associated with a product sold by that vendor. A unique combination of a VID and PID allows a USB host to identify one USB product type from another.

The BSL is considered a separate USB device from any that might exist in the main user application. Therefore, it has a VID/PID. If there's also a USB function in the user application, it has its own VID/PID. Having different VID/PIDs is what allows the host to differentiate whether the user application is in control of the device, or whether the device is making itself available for firmware update.

The VID/PID implemented by the default USB BSL is 0x2047/0x0200. 0x2047 is a TI-owned VID, and 0x0200 is a PID unique to the BSL for the F552x/F550x/F663x device families.

If it's desired to provide the BSL's VID/PID a unique value, then it is necessary to change it in three locations:

- the on-chip flash BSL
- the host application (Python Firmware Upgrader tool)
- the RAM BSL program that is downloaded from the Python Firmware Upgrader.

The host application stores this in hid.py (see Sec. 6.3).

The other two items require re-compiling the BSL source, which is provided with the application report *Creating a Custom Flash-Based BSL (slaa450).* After modifying the VID/PID in *BSL_Device_File.h,* the BSL must be compiled in two different ways: as a flash BSL and as a RAM BSL. Refer to *slaa450* for more information.

The new flash BSL image must be downloaded into every target MSP430 device that will be using this VID/PID for BSL.

The new RAM BSL image must be built as an MSP430 TI-TXT file (*.txt), and placed into the Python Firmware Upgrader project's \bsl5 folder.

When all three components contain the new VID/PID, the system will function in the same way as if the defaults are used.

3.4 BSL Password

When the host attempts to interface with the device's BSL, the BSL requires a password. (This is the same password mechanism used in non-USB MSP430 BSL versions.) The password consists of the device's contents at addresses 0xFFE0 to 0xFFFF that exist at the time of BSL access. This range is a subset of the vector table. For more information on the password and security measures of the BSL, see the *MSP430 Programming via the Bootstrap Loader (SLAU319)* After the correct password is entered, the firmware can be upgraded. If the password is incorrect, the BSL automatically performs a mass erase of the device's entire main flash area.

Some applications store data in flash during normal operation that needs to be preserved after the update. Developers of these applications should always report the *correct* password to avoid the mass erase. It's recommended to ensure the password never changes during the product's lifetime, by fixing the locations of the ISRs (whose addresses comprise the BSL password) to consistent locations in memory. If this isn't done, then the password will likely change after each revision; allowing this to happen makes it necessary to upgrade the versions in succession (v1, v2, v3, etc.) to ensure that the right password is used each time, if data contents are to be preserved. It's usually difficult to guarantee the user will do this.

Fixing the ISRs in controlled locations solves this problem. It can be done during development of the MSP430 application, using a variety of methods. One is to create a dedicated linker segment for each ISR with a fixed address, and assigning the ISR functions to those segments.

However, some applications don't need to preserve data after the update, which means that a mass erase causes no harm. These might be better served by reporting a blank (i.e., invalid) password. The resulting mass erase means that the host always knows the correct password on the second try: all 0xFFFF's. This method is actually easier than reporting the correct password, because it avoids the planning process described above.

See Sec. 6.6 for a discussion of how to manage this with the Python Firmware Upgrader.

3.5 Distributing Firmware Updates to the Field

If distributing firmware updates to end users, this can happen via the OEM's website, a CD-ROM, or other methods. To create new host applications for each revision using software based on the Firmware Upgrader, it is only necessary to replace the firmware image in the source project.

It should be noted that the MSP430 firmware within the host application is not encrypted. If encryption is desired, a custom on-chip BSL can be written that decrypts the image after transmission from the host.

3.6 Interruption of the Upgrade Process

When upgrading firmware in an MCU, there is always a chance that the process could be interrupted, potentially corrupting the code. Updates conducted over USB are more susceptible to this, because the end user is able to remove the USB cable at any point in the process.

An MSP430 firmware update system should have two layers of security to assist in recovering from such an event. One is to maximize the time in which the reset vector is blank. The "blank reset vector" method ensures that a device reset under this condition invokes the BSL, allowing recovery simply by re-attempting the upgrade.

This can be accomplished by ensuring that the vector table is erased at the very beginning of the process, and then ensuring that the vector table (and specfically the reset vector) is the last thing written. If done this way, then the only remaining window of vulnerability is during the erase, which takes approximately 25 ms (see the device data sheet for specifics).

The probability that the user will remove the USB cable during this vulnerability window might be calculated as the time for the erase (approximately 25 ms) divided by the overall update session (for example, 25 seconds). With these assumptions, there would be one unrecoverable session in a thousand interrupted sessions. The vast majority of users will not remove the cable, making the odds more remote. In the same scenario, if 5% of users remove the cable during the upgrade, then about one in 20,000 upgrades will experience a corruption that is unrecoverable by the reset vector method.

(Note that the Python Firmware Upgrader GUI described in this application note does not follow this procedure; it writes memory incrementally, making it vulnerable to unrecoverable interruptions from the time that 0x10000 is written, to the time the last byte of upper memory is written. In large programs, this can be several seconds. See Sec. 6.7 for more discussion.)

If this interrupt happens, recovery of the device would require the hardware method (or JTAG). The hardware method will always succeed at reprogramming the device in the field, with the tradeoff being the cost of implementing a button on the board. If the odds of irrecoverable failure with the reset vector method are too high, a BSL button should be implemented (see Section 4.5).

3.7 USB PLL Reference Clock and XT2 Oscillator

The MSP430 USB module requires a clock to serve as a reference for the USB PLL. The clock can either be generated by the XT2 oscillator by applying a crystal, or a clock source from elsewhere on the board can be applied to XT2 in bypass mode. In the former case, the minimum frequency is 4MHz; in the latter, it's 1.5MHz. (See the device datasheet for parametric details.) The PLL is programmable and can accept a wide variety of frequencies. (See the *MSP430x5xx/6xx Family User's Guide (SLAU208)* for architectural details.)

The on-chip BSL needs to know the reference frequency in order to configure the PLL. By default, it's able to detect the presence of four frequencies on XT2:

- 4MHz
- 8MHz
- 12MHz
- 24MHz

If it finds one of these, it programs the PLL accordingly. If any other frequency is to be used, the BSL must be modified and re-programmed into every device. Without this, USB will not function properly, and the BSL will not be able to enumerate on the host.

The process of customizing this is described below. Note that if one of the four auto-detect frequencies is used, this step can be eliminated.

Please note that the default BSL does not support XT2 bypass mode, in which the crystal is replaced by an external clock fed into XT2. If this is desired, then the clock configuration code in the BSL source must be customized – see the application note *Creating a Custom Flash-Based BSL (SLAA450).*

To modify the XT2 frequency, the BSL source code must be modified and downloaded to each device. The BSL's source has been arranged to make this simple and straightforward, for those wishing only to make this small modification.

The BSL source is available as part of the application note *Creating a Custom Flash-Based BSL (SLAA450).* This source comes in the form of IAR projects; developers not already using IAR can download IAR Kickstart from TI's website at no charge. The BSL fits within Kickstart's size limitation.

The BSL source has several constants that need to be modified to reflect the new values. The constants are located near the top of the file *BSL_Device_File.h.*

Table 2. Customizing XT2 Frequency in the BSL Source

Constants	Description
SPEED_1 SPEED_2 SPEED_3 SPEED_4	Values related to the BSL's frequency auto-scan feature. The number is equal to the frequency, in Hz. It is recommended to change all four to the target value.
SPEED_1_PLL SPEED_2_PLL SPEED_3_PLL SPEED_4_PLL	Values related to the BSL's frequency auto-scan feature. These values correspond one-for-one with SPEED_x above. The value assigned to these constants should be one from the MSP430 header file, of format USBPLL_SETCLK_xx_y. It is recommended to change all four to the target value.

Once the modifications are made, the program can be built and downloaded into a device using IAR and an emulation tool like the FET430UIF FET tool. Once performed, development can proceed as normal with a non-BSL project on this device, and the BSL contents will remain updated.

A different approach is needed for programming devices in production. A single *msp430-txt* image file is needed that contains both the BSL object code and the main application object code. Production programmers can then use this file to update each device. This *msp430-txt* file must be created manually by concatenating the application to the end of the customized BSL image. The example below demonstrates this process.

First, edit the XT2 frequency in the BSL's IAR source project as described earlier, and output an *msp430-txt* file for the BSL. It will appear similar to below when viewed with a text editor. Note that the starting address is 0x1000, the location of the BSL in MSP430F5xx memory.

msp430-txt file for the BSL:

```
@1000
2C 3C 06 3C FF 3F FF 3F FF 3F FF 3F FF 3F FF 3F
3D 90 AD DE 04 20 3E 90 EF BE 01 20 03 3C 0C 43
.
.
.
0F 4C 0F 5D 03 3C CC 43 00 00 1C 53 0C 9F FB 23
10 01 80 00 76 44 80 00 7A 44 FF 3F
@FFFE
00 44
q
```

Now output the *msp430-txt* file of the main application. This code is positioned higher in memory.

msp430-txt file for the user application:

```
@4400
31 40 00 44 3C 40 00 24 3D 40 01 00 B0 13 60 44
.
.
0F 4C 0F 5D 03 3C CC 43 00 00 1C 53 0C 9F FB 23
10 01 80 00 76 44 80 00 7A 44 FF 3F
@FFE6
18 44
@FFFE
00 44
q
```

The two should then be combined with a text editor. Open the BSL output and delete anything in the range of the interrupt vectors (0xFF80-FFFF), as well as the “q” character (which denotes the end of the file). Then, append the application’s output. When combined, they should look like:

msp430-txt file for the two combined:

```
@1000
2C 3C 06 3C FF 3F FF 3F FF 3F FF 3F FF 3F FF 3F
3D 90 AD DE 04 20 3E 90 EF BE 01 20 03 3C 0C 43
.
.
.
3B 00 00 00 1F 42 44 1C CF 4C 01 00 E2 43 00 1C
34 21 78 56 80 00 3A 17 21 83 B1 40 4C 04 00 00
02 3C B1 53 00 00 81 93 00 00 FB 23 21 53 18 02
B2 40 00 51 CE 03 92 B3 CE 03 FD 2B 10 01 1D 15
10 01
@17F0
FF FF 2A 10 A5 3C 5A C3 FF FF 00 10
@17FC
FF FF FF FF
@4400
31 40 00 44 3C 40 00 24 3D 40 01 00 B0 13 60 44
.
.
0F 4C 0F 5D 03 3C CC 43 00 00 1C 53 0C 9F FB 23
10 01 80 00 76 44 80 00 7A 44 FF 3F
@FFE6
18 44
@FFFE
00 44
q
```

The combined file can then be downloaded into each target MSP430 device using a production programmer, like the GANG430.

4 Field Firmware Upgrade: System Models

Hardware sold by TI for USB-equipped MSP430 devices -- the F5529 LaunchPad, for example -- comes with a pushbutton marked “BSL”. This button implements the hardware method of invoking the BSL, by pulling the PUR pin high.

This is convenient for these bare PCBs, but may not be desirable in production hardware. The button adds cost. It also might be difficult to access them, once an enclosure is placed around the board.

Therefore, this section discussed alternative ways to invoke the BSL in a finished product. Five ways are described. Each represents a different way the end user will experience the update process. They also vary to some degree in the amount of development effort required.

Table 3 shows models for normal update sessions, and Table 4 shows failsafe models (for recovery from an interrupted session). Under most circumstances:

1. one “normal” model should be chosen
2. the reset vector failsafe model should be planned for
3. the upgrade button failsafe model should be considered, as described in Section 4.5.

Table 3. Primary Usage Model Options

Application Characteristics	How the User Would Invoke the BSL	Described Where
The user application doesn’t contain a USB function (in other words, USB is only used for firmware update)	BSL is invoked any time the user attaches the device to a USB host	Section 4.1
The user application contains its own USB function (in other words, USB is used for two purposes – in the user application and firmware update)	User accesses the device’s user interface to enter a special “update mode”; once in this mode, the BSL is invoked any time the user attaches the device to a USB host	Section 4.2
The user application contains its own USB function, and the above method isn’t possible, or it’s desired to invoke the BSL from host control	With the device already attached to the host for the normal USB function, the user accesses a host application to update firmware	Section 4.3

Table 4. Failsafe Usage Models (For Recovering from an Interrupted Session)

Application Characteristics	How the User Would Invoke the BSL	Described Where
The interruption occurred while the reset vector was blank (true of most interrupted sessions, if following the procedure in Sec. 6.7)	User attaches the device to any USB host, and then generates a BOR.	Section. 4.4
The interruption occurred while the reset vector was *not* blank (rare case)	User performs an action that pulls the PUR pin high while generating a BOR (probably using one or two dedicated switches)	Section. 4.5

4.1 If the User Application Does Not Contain USB Functionality

This is the simplest model. A USB interrupt will be generated when the device is attached to a USB host. If there's no other USB function, it is reasonable for the application to interpret this as the user attempting to update firmware. Code can be placed in the USB ISR (if USBVECINT points to a VBUS-on event) to invoke the BSL using the software method:

```
void main(void)
{
  volatile unsigned char fInvokeBSL = 0;

  WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer -- not used in this example

  // Activate the sensing of 5V VBUS on the USB cable
  USBKEYPID  =  0x9628;        // Unlock the USB registers
  USBPWRCTL  |= VBONIE;        // Enable the VBUS-on interrupt
  USBKEYPID  =  0x9600;        // Lock the USB registers

  while(1)
  {
    __bis_SR_register(LPM3_bits + GIE); // Go into LPM3; keep ints enabled.

__  disable_interrupt();         // Ensure no application interrupts fire during BSL
    if(fInvokeBSL)
      ((void (*)())0x1000)();  // This sends execution to the BSL.  When execution
                               // returns to the user app, it will be via the reset
  }                            // vector, meaning execution will re-start.
}

#pragma vector=USB_UBM_VECTOR  // USB Interrupt Service Routine
__interrupt void iUsbInterruptHandler(void)
{
  switch (__even_in_range(USBVECINT & 0x3f, USBVECINT_OUTPUT_ENDPOINT7))
  {
    case USBVECINT_PWR_VBUSOn:
      fInvokeBSL = 1;                         // Set flag
      __bic_SR_register_on_exit(LPM3_bits);  // Exit LPM3 when exiting the ISR.
      break;

    default:
      break;
  }
}
```

Note that the BSL isn't invoked directly from within the ISR. It's best to return from the ISR, and then invoke the BSL.

Whenever the device is removed from USB while the BSL is in control, the BSL jumps to the reset vector. This puts the user application back into control.

If the user attached to the USB host errantly, the BSL will be invoked, but unless a host application tries to interface with it, the device shouldn't experience any negative effects. Once removed from the host, a BOR reset will occur, and code will execute from the beginning. While the device is attached to USB, with the default BSL in control, buttons will not be monitored, and no display elements will be driven, so the device might appear to be "dead" from the user's perspective during this time.

4.2 If the User Application Does Contain USB Functionality

This model is exactly the same as above, except in this case the user must put the device into a special mode before a USB attachment triggers a BSL invocation. For example, the user might use a menu system on an LCD display, or be required to press and hold two buttons for five seconds. Once in this mode, any USB attachment would cause the application to invoke the BSL using the software method.

The implementation is similar to the method described above, with the addition of code to evaluate the mode. If one of TI's MSP430 USB API stacks are being used, it's recommended to place this code in the `handleVbusOnEvent()` handler, rather than directly in the USB ISR:

```
BYTE USB_handleVbusOnEvent()
{
  if(inFWUpdateMode)
  {
    fInvokeBSL = 1;                  // Set flag triggering main() to invoke BSL
  }
  else
  {
    if(USB_enable() == kUSB_succeed) //Connect to USB, for user application function
    {
        USB_reset();
        USB_connect();
    }
  }
  return TRUE;
}
```

fInvokeBSL can then be evaluated in the main loop to determine whether to jump to 0x1000 in the manner shown in the previous section.

4.3 If Host Control of Firmware Updating is Desired

This model can only be used if there's a USB function in the user application, additional to USB being used for firmware update. It's presumed that a host application program already exists for communication with the user application, in order to carry out this normal USB function. The firmware upgrade process then becomes a feature within this application, rather than being a standalone application.

In this model, the device is already attached to the USB host, and therefore enumerated under the control of the user application. The following procedure takes place:

1. The user selects a control in the host application program, to update the firmware.
2. The host application sends an application-specific command to the device, telling it to invoke the BSL
3. The MSP430 user application recognizes the command. It disconnects from USB, disables general interrupts, then uses the software method to invoke the BSL (jumps to 0x1000).
4. Under BSL control, the device re-enumerates on the USB host

5. The host application recognizes that the BSL device has enumerated. It then uses BSL commands to update the firmware, in the same manner described earlier.

6. The host application issues a BSL command telling the device to issues a BOR or a PUC. This time the device enumerates under control of the new user application.

From the end user's perspective, this entire process is triggered simply by pressing a button or selecting a menu item in the host application program.

Implementation of this model is more application-specific than the previous methods. It requires that a command be placed within the USB traffic of the user application, by which the host application can tell the device to invoke the BSL. This requires changes to both the host and MSP430 applications and the MSP430 application must recognize the command to invoke the BSL.

4.4 Recovering from an Interrupted Update Session when the Reset Vector is Blank

If the Python Upgrader is used in the manner described in Sec. 6.7, then the vast majority of interrupted update sessions will experience the interruption while the reset vector is blank. This ensures easy recovery the next time the device is attached.

If a BOR occurs while the reset vector is blank, and if the device is already attached to an active USB host (5V VBUS is present), then the BSL will be invoked. VBUS must be present when the BSL is invoked, otherwise the BSL goes into an invalid state and another BOR is required.

To ensure the BOR happens, it might be necessary to provide the end user with special instructions that cause a BOR. For example:

1. Remove power from the device, then restore it *after* the device has been attached to USB. (A power-up event generates a BOR.)

2. Push a button that causes an edge on the MSP430's RST pin. (For example, a "paperclip switch" on the back of the device.)

The user interface (buttons/display) should not be replied upon for this, since the application firmware is presumed corrupted at this point and unable to process the interface.

If the device receives power from the USB, it might only be necessary to remove the battery and attach to the host. Upon attachment, the device will power up and VBUS will be present; the two conditions would be met, and therefore the BSL would be invoked.

If the device only receives power from a battery, and not from USB, the device could be attached to USB and then the battery re-inserted.

If the device has a rechargeable battery, the reset switch is probably required. This is because most such systems don't have removable batteries.

Although unlikely, the interrupted session might result in the reset vector being corrupted (not blank). Under this condition, a reset will result in unpredictable execution. In this situation, only the method in the Section 4.5 (or JTAG) can recover the device.

4.5 Recovering from an Interrupted Update Session Using a "Firmware Update Button"

This model always works, no matter the contents of user flash. It can either be used as a backup to one of the others, or it can be used as a primary method of performing updates. The downside of this option is adding the cost of a button to every board produced.

As described in Section 3.6, the probability of a failure that needs this model for recovery is quite low. If the engineer is willing to accept the risk, he or she could choose to omit this method. If absolute recovery is needed, this model can be implemented.

In this model, a button is implemented which forces a pull-up on the PUR pin. The user holds this button and then causes a BOR on the MSP430. (In this way, it invokes the BSL using the hardware method.) This button must be dedicated to this function; it cannot be part of the device's normal user interface. An example might be a "paperclip switch" underneath the battery cover.

While the button is held down, a BOR must be generated, possibly using one of the methods described in Section 4.4.

An example procedure is described below. It assumes an application in which a "firmware upgrade button" is implemented using a paperclip switch in the battery compartment, and in which the device is able to operate from the USB host in the absence of a battery:

1. User removes the battery. This also exposes the switch.
2. User keeps the button held down while attaching to USB. (Note this might be difficult to do on a small handheld device using two hands.)
3. The device enumerates under BSL control; the user can now release the button.
4. The host updates the firmware, in the same manner described elsewhere in this document.

Disadvantages of this model are the cost of the switch, and in some applications the possibly awkward process for the end user to hold down the button while causing a reset.

An example circuit that can be used to implement this is shown in the figure below. (The TS430PN80USB FET target board uses this circuit.)

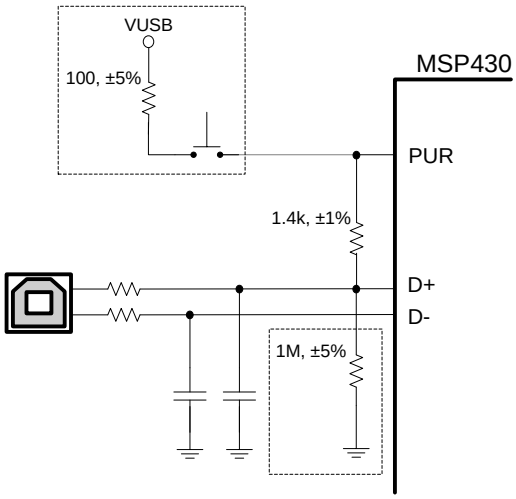

Figure 3. Example Circuit for Hardware Invocation Circuit

Prior to software driving PUR high to signal USB attachment, the 1MΩ pulldown resistor keeps PUR low in cases where the switch isn't being pressed. After software drives PUR high, the pull-down is weak enough to not affect the operation of D+.

5 Python Firmware Upgrader Tool

The Python Firmware Upgrader tool provides both a GUI interface and a command-line interface for upgrading MSP430 target firmware on USB-equipped MSP430 devices. The Python Firmware Upgrader is located within the USB Developers Package.

This section describes the tool, as distributed. Section 6 includes specific steps on how to customize it.

5.1 System Requirements

The \Python_Firmware_Upgrader project contains source code, BSL file, RAM_BSL file, documentation, license manifest, executable and example files. The tool is built upon an existing open-source toolset for downloading MSP430 code, located at https://pypi.python.org/pypi/python-msp430-tools. This package is replicated within the MSP430 Python Firmware Upgrader.

Python version information is contained in the *release_Notes_Example_Python_Firmware_Upgrader.html* file.

The files and folders contained in the Python Firmware Upgrader directory are as follows:

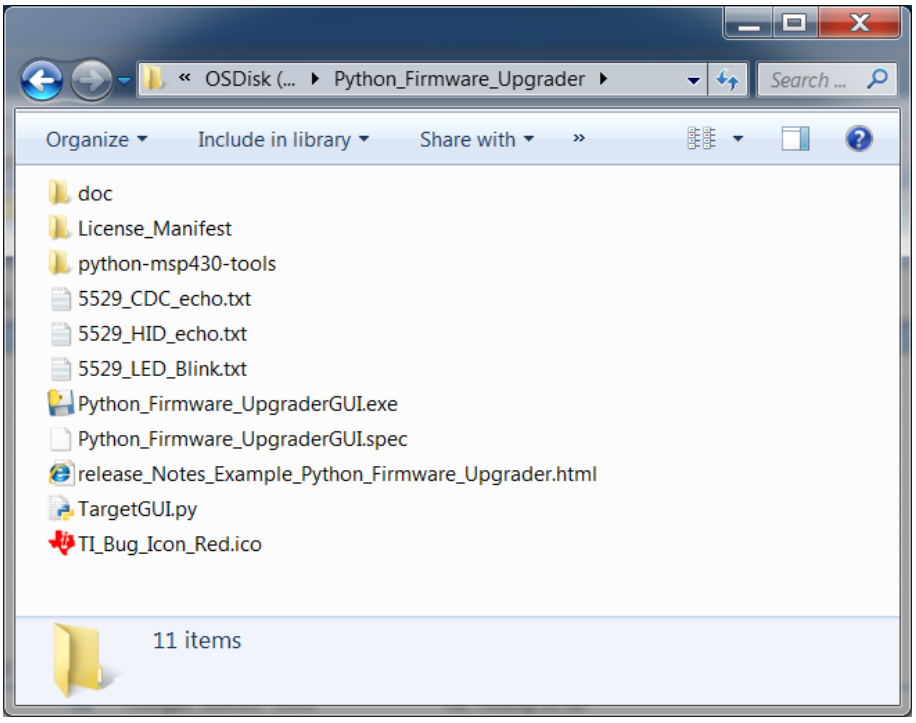

Figure 4. Python Firmware Upgrader Directory Structure

5.2 Launching the Tool

5.2.1 Command-Line

The python-msp430-tools can be invoked by command line on Windows or Linux. This bypasses the GUI (TargetGUI.py) that TI has built on top of the open source toolset.

The *release_Notes_Example_Python_Firmware_Upgrader.html* file, in the tool's directory, has some example command-line invocations. The documentation for the python-msp430-tools provide additional help.

5.2.2 Windows

For Windows XP and Windows 7, an executable file has been placed in the tool's directory: *Python_Firmware_UpgraderGUI.exe.* This was built from TargetGUI.py and the python-msp430-tools. It was built using the *pyinstaller* Python module.

5.2.3 MAC

Python is available for the MacOS, but the Upgrader has not been tested on the Mac.

5.2.4 Linux

The tool should be launched from a command line. See the instructions listed in the *Release_Notes_Example_Python_Firmware_Upgrader.html* file found in the installed directory for the tool.

5.3 Using the GUI

When opening the Python Firmware Upgrader program, the following screen is displayed if the device if an MSP430 USB BSL device can't be found. It determines this by looking for the MSP430 USB BSL's vendor ID and product ID (VID/PID), 0x2047/0x0200.

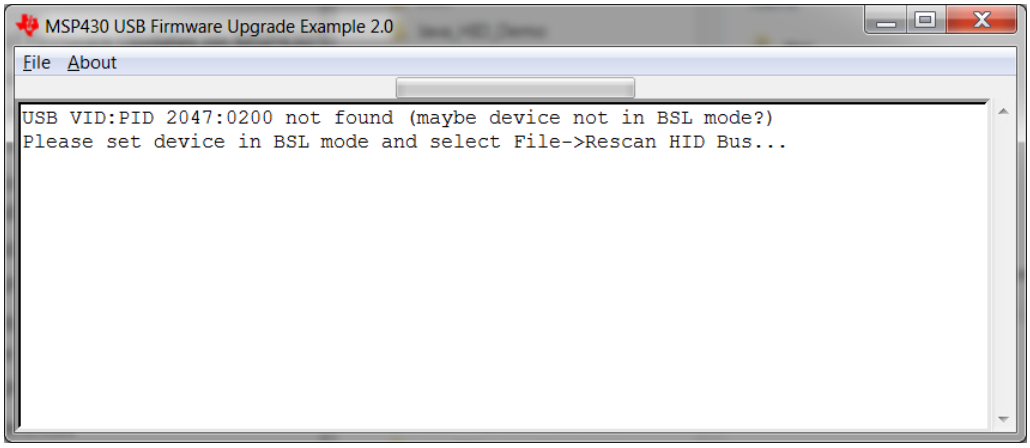

Figure 5. Non-BSL Mode View

If the USB BSL device is found, the following screen is displayed.

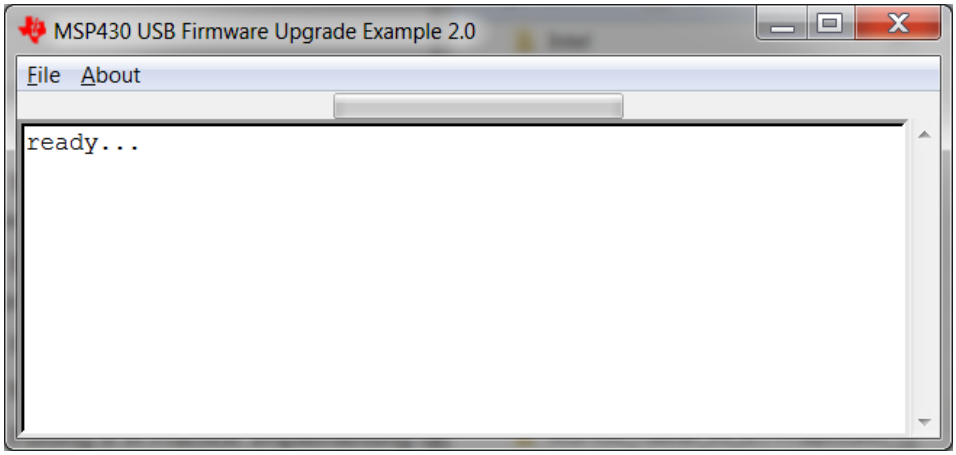

Figure 6. BSL Mode View

As discussed in Sec. 2, there are three ways by which the USB BSL can be invoked. If you're using TI-provided hardware, like the MSP430F5529 LaunchPad or Experimenter's Board, there is always a "BSL button". Press this button while attaching the device to the USB host, hold it for a half-second longer, then release. Then, in the Upgrader, select File→Rescan.

When the GUI detects the presence of an MSP430-based device under the control of the BSL, the "Load Demo Firmware" and "Open User Firmware" menu options are activated in the "File" menu. When you hover the cursor over the "Load Demo Firmware" selection, three example programs are listed. If an example program is selected for download, the Python Firmware Upgrader tool initializes the USB BSL and downloads the firmware . Several example programs are provided.

There is also an option to select an external file. You can use this option to select a file you've created. The file selection must be of format *msp430-txt*, which can be generated by standard MSP430 IDEs. In IAR, this format is called *msp430-txt.* In CCS, it's called *TI-TXT.*

5.4 Source

5.4.1 Project Structure

The Python Firmware Upgrader folder contains:

- TargetGUI.py file
- *Python_Firmware_UpgraderGUI.exe* file (Windows executable version of TargetGUI.py and its dependent files)
- the python-msp430-tools directory downloaded from https://pypi.python.org/pypi/python-msp430-tools, and updated with the latest BSL and RAM_BSL file.
- Three example file images in msp430-txt format: 5529_CDC_Echo.txt, 5529_HID_Echo.txt, 5529_LED_blink.txt
- Release notes

5.4.2 Code Organization

The TargetGUI.py source file contains code that creates the GUI and directs the python-msp430-tools to perform the firmware upgrade. Most of the code is located within these tools.

5.4.3 Icons

The icon for this project is found in the Python Firmware Upgrader folder.

5.4.4 Limitations

The Python Firmware Upgrader GUI has these limitations:

- When programming the target MSP430, it writes from the lowest address to the highest address. In target images containing locations above 0x10000, there is a vulnerability to interruption: if power is interrupted, or if the USB cable is removed, while memory above 0x10000 is being written, the device may become unrecoverable. (See Sec. 3.6.) This could be avoided by skipping the reset vector (0xFFFE) and writing it only after all of flash

has been written; the Python MSP430 Tools do not take this step, and thus it is vulnerable to these interruptions.

- The password scheme assumes that the vector table of the new and old images are the same. If the vector table changed, the BSL password will be incorrect, and a mass erase will occur. The main consequences of this are the loss of any stored data in flash, and possibly a longer flash update time.

If using the python-msp430-tools from command line, these limitations can be bypassed. The GUI could also be modified to address them. See Sec. 6.6 and Sec. 6.7.

6 Customizing the Python Firmware Upgrader

Some suggested modifications are discussed here, both of the GUI, and in various ways to invoke via command line.

6.1 Using the Application As-Is

It's possible that the application is perfectly acceptable for your use, as-is.

Be aware of the limitations discussed in Sec. 5.4.4. The command line options have more flexibility in addressing these, if this isn't acceptable.

6.2 Customizing the GUI Look-and-Feel

If you wish to use the GUI, you may wish to customize its look-and-feel – for example, add your organization's logo.

6.3 Modifying the VID/PID

As discussed in Sec. 3.3, if you prefer not to use TI's MSP430 BSL VID/PID of 0x2047/0x0200, you can change it.

Sec. 3.3, explained how to change it in the MSP430's BSL. To change it in the Python app, it must be changed within the python-msp430-tools: in the hid.py file, in the method HIDBSL5.open().

6.4 Updating the RAM BSL

If you change the flash BSL in your target MSP430, you should also change the RAM BSL in this application.

The RAM BSL is located in \python-msp430-tools\msp430\bsl5\, of the format RAM_BSL_____.txt.

6.5 Causing the GUI to Automatically Download a Single Target Image

The GUI version of the tool asks the user to select a target image; either one of three defaults, or any TI-TXT file.

You may prefer the tool to simply download your target and exit, without any GUI besides a status dialog box. Simply modify the Python GUI application to call TargetGUI.doLoad() upon the GUI opening, specifying your specified target file.

6.6 Customizing the Password

As discussed in Sec. 3.4, the main reason to report a valid password is to avoid a mass erase that would erase any critical data stored in flash from previous execution. If this isn't a concern, then the mass erase causes no problem.

If data needs to be retained, then a valid password should be reported, and then segment erases need to be used, instead of a mass erase. These segment erases can avoid writing over data portions of flash.

As described earlier, the Python Firmware Upgrader GUI forces a mass erase. However, the command-line options provide the ability to avoid this. First, the command line option –*password* allows a password file to be input and used during connection with the target MSP430. It also has an option –*b* that causes only the segments that will be occupied by the new image to be erased; this will preserve any data stored elsewhere.

The GUI could be modified to read an external password file, allowing it to be sent to the target MSP430 when the open_connection() method is called. The erase_by_file() method can then be called, to only erase the relevant segments.

6.7 Addressing the Potential for Unrecoverability after Interruption

As discussed in Sec. 3.6, applications where the new target image contains code above the address 0x10000 are at risk of unrecoverability, if the process is interrupted by power loss or the removal of the USB cable. The Python GUI does not account for this possibility.

The command-line access provides some flexibility to solve this. Two separate BSL sessions could be used:

1. Create a target image where the vector table is blank, but has valid code everywhere else. Invoke the python-msp430-tools to write this image
2. Then create a small target image that only writes the reset vector, or perhaps the entire vector table. (Be sure to use the correct password, otherwise this second session will cause a mass erase of your first session's code.)

What this does is ensure the reset vector stays blank throughout the entire write process. This way, if the process is interrupted, the device will be able to re-enumerate as a BSL device, allowing it to be recovered.