MSPDebugStack Developer's Guide

MSP430

ABSTRACT

The MSPDebugStack is a dynamic library that provides functions for controlling/debugging Texas Instruments MSP430 Ultra-low Power microcontrollers during software development phase. For this purpose the MSP430 microcontroller is controlled by the MSPDebugStack using the MSP430 device's JTAG interface. The MSPDebugStack provides device control (e.g. Run, Stop), memory programming and debugging functionality (e.g. Breakpoints).

Standard 4-wire JTAG and the low pin count debug interface called Spy-bi-Wire (2-wire JTAG) are supported by the MSPDebugStack. Furthermore all MSP430 debuggers can be used in combination with the MSPDebugStack.

The MSPDebugStack greatly simplifies the control of the MSP430 microcontroller, as the user is completely isolated from the complexities of the JTAG protocol.

This application note provides an overview of the MSPDebugStack and its usage to control MSP430 microcontrollers. Additional information is provided in the library's C-Header files. Furthermore, several sample programs and flow charts are showing the practical use of the MSPDebugStack.

NOTE: This application note assumes knowledge of the C, C++ language, the Dynamic Link Library mechanism, the MSP430, and MSP430 JTAG mechanism.

NOTE: Refer to the MSP430 Hardware Tools User's Guide (SLAU278) for information on actual hardware connection to the devices' JTAG pins. For further details on the MSP430 specific JTAG implementation in silicon refer to the MSP430 Memory Programming User's Guide (SLAU320).

Contents

Figures

Revisions

Table 1. Document Revision History

Revision	Date	Author	Notes
0.1	06/2005	W. Lutsch	Initial draft
0.2	09/2005	W. Lutsch	Added Appendix B: Installation of VCP for MSP-FET430UIF
0.3	10/2005	W. Lutsch	Added Spy-bi-Wire information Added Figure 5. Configuring the JTAG protocol Added Abbreviations
0.4	03/2006	W. Lutsch	Added Speed up Flash Programming
0.5	06/2006	W. Lutsch	Added Supporting more than one MSP-FET430UIF
0.6	05/29/2007	W. Lutsch	Added Attach to a running device
0.7	02/10/2009	W. Lutsch	Added eZ430 tool information (both eZ430-F2013 and eZ430-RF2500, Supporting eZ430 emulator dongles) Added information about certified VCP driver (affected: Supporting MSP-FET430UIF, Appendix B Installation of VCP for MSP-FET430UIF) Added Appendix C Switching between certified and non-certified VCP driver Added UseCases
0.8	06/16/2009	W. Lutsch	Added NOTE to abstract which references to SLAU278 & SLAU265
0.9	03/23/2010	F.Berenbrinker	Added notes for new API functions and the automatic protocol scan
1.0	08/09/2011	F.Berenbrinker	Added Code Examples Added MSP430 Flasher as an example for MSP430.dll usage
1.1	08/22/2011	F.Berenbrinker	Remove EEMgui. Example Remove Appendix C
1.2	01/24/2014	F.Berenbrinker	CleanUp – Remove PIF, VCP Add MSP-FET and eZ-FET information
1.3	02/20/2014	F.Berenbrinker	CleanUp – Remove PIF, VCP Add HID recovery Update flow charts – UML stile CleanUp code examples Remove deprecated appendixes
1.4	02/04/2016	F. Fischer	Update code examples and flow charts
1.5	02/08/2016	F.Berenbrinker	F.Berenbrinker – Remove HIL.dll reverences
1.6	24/05/2016	F. Fischer	Updated references to MSP430.dll

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Abbreviations

- **MSP-FET430UIF:** Official product designation of Texas Instruments MSP430 USB JTAG interface (USB FET).
  http://www.ti.com/tool/msp-fet430uif
- **MSP-FET:** Successor of MSP-FET430UIF
  http://www.ti.com/tool/msp-fet
- **MSP-EXP430F5529LP:** Official product designation of Texas Instruments MSP-EXP430F5529LP. (includes eZ-FET Lite debugger)
  http://www.ti.com/tool/msp-exp430f5529lp
- **eZ430-RF2500:** Official product designation of Texas Instruments MSP430 Wireless Development Tool (includes eZ430- debugger).
  http://focus.ti.com/docs/toolsw/folders/print/ez430-rf2500.html
- **SBW:** Spy-bi-Wire JTAG debug interface utilized on MSP430 low pin count devices.
- **CDC:** Communication Device Class
- **MSPDebugStack:** Official name of the software stack
- **MSPDS:** MSPDebugStack
- **USB-FET: Synonym for different debuggers:** MSP-FET430UIF, MSP-FET, eZ-FET or eZ-FET lite

Developer's Package Folder and File Structure

The MSPDdebugStack developer package is composed of the following folders and files. Installing the provided developer package will create the following folders and files in the selected installation destination directory.

- **ApplicationExamples**: This folder contains a set of application examples on how to apply the MSPDebugStack functionality. Refer to section Application Examples for specific details on each code example.
- **Doc**: This folder contains the complete API documentation of the MSPDebugStack in HTML and Compressed HTML format. Furthermore, this document is part of the Doc folder.
- **Driver**: This folder contains according driver setup and files for
  - **CDC**: TI's CDC driver (DLL V3 only) supporting the MSP-FET430UIF, the MSP-FET and the eZ-FET JTAG interfaces.
  - **VCP**: TI's VCP driver – deprecated! (DLL V2 only) supporting the MSP-FET430UIF JTAG interfaces.
  - **INF**: MS-Windows driver information file for MSP430 Application UART available with eZ430-RF2500 (eZ430 debuggers) emulator dongles (refer to Supporting eZ430

emulator dongles for details). Refer also to the **eZ430-RF2500 Development User's Guide** (SLAU227) for further information.
This folder also contains a subfolder called 'PreinstallCDC'. It contains an example source code which shows how to install the driver INF file on a MS-Windows PC.

- **Inc**: This folder contains all needed C-Header files to use the MSPDebugStack inside an application. API functions are documented in detail inside these files. Furthermore, function prototypes, function parameters and function return values are documented. Also all needed typedefs, #defines, enumerations, and data are part of the C-Header files.
  - **MSP430.h**: This file is the main header file for the MSPDdebugStack, and provides the function prototypes, typedefs, #defines, enumerations, and data structures for the library functions. This file is normally located in the same directory as your application's source file, and should be #included by your application's source file. This file is used during compile-time *(refer to* General application and device handling *for more general information).*
  - **MSP430_Debug.h**: This file is a header file for the MSPDdebugStack, and provides the function prototypes, typedefs, #defines, enumerations, and data structures for the debugging functions of the library. This file is normally located in the same directory as your application's source file, and should be #included by your application's source file. This file is used during compile-time *(refer to* Controlling device program execution *for more general information).*
  - **MSP430_EEM.h**: This file is a header file for the MSPDdebugStack, and provides the function prototypes, typedefs, #defines, enumerations, and data structures for the **enhanced** debugging functions of the library. This file is normally located in the same directory as your application's source file, and should be #included by your application's source file. This file is used during compile-time *(refer to* Enhanced Emulation Module (EEM) Access – EEM API *for more general information).*
  - **MSP430_FET.h**: This file is a header file for the MSPDdebugStack, and provides the function prototypes, typedefs, #defines, enumerations, and data structures for MSP-FET430UIF maintenance functions of the library. This file is normally located in the same directory as your application's source file, and should be #included by your application's source file. This file is used during compile-time *(refer to* MSP-FET430UIF Firmware Update Support *for more general information).*
- **Lib**: This folder contains according library files.
  - **MSP430.lib**: This file is the library file for the MSPDdebugStack and is required to access functions of the library. This file is normally located in the same directory as your application's source file and should be added to the Linker Object/Library Modules list of your application. This file is used during link-time.
- **MSP430.dll**: This file is the dynamic link library and contains the device control functions. This file is normally located in the same directory as your application's executable file, or in your computer system's default DLL folder. This file is used during run-time.
- **libmsp430.so**: This file is the dynamic library for Linux 32 bit and contains the device control functions. This file is normally located in the same directory as your application's executable file, or in your computer system's default library search path. This file is used during run-time.

- **libmsp430_64.so**: This file is the dynamic library for Linux 64 bit and contains the device control functions. This file is normally located in the same directory as your application's executable file, or in your computer system's default library search path. This file is used during run-time.
- **libmsp430.dylib**: This file is the dynamic library for Mac OS X and contains the device control functions. This file is normally located in the same directory as your application's executable file, or in your computer system's default library search path. This file is used during run-time.
- **revisions.txt**: This file provides information about added features of dedicated versions of the MSPDdebugStack.
- **Objects**:
  - **Libusb:** This folder contains the source files of the libusb version that is used for the Linux binary of the MSPDebugSyack.
  - **Linux32:** This folder contains static libraries for Linux 32 bit for recompiling the MSPDebugStack with a custom libusb.
  - **Linux64:** This folder contains static libraries for Linux 64 bit for recompiling the MSPDebugStack with a custom libusb.
  - **Makefile:** This is the Makefile for recompiling the MSPDebugStack with a custom libusb.
  - **README.txt:** This text file contains instructions on how to recompile the MSPDebugStack with a custom libusb.

Using the MSPDebugStack

General application and device handling

The usage of the MSPDdebugStack is straightforward. The functions of the library are sequenced as follows:

1. The interface is initialized: MSP430_Initialize()
2. The target architecture is chosen (MSP430 or MSP432_M4): MSP430_SetTargetArchitecture()
3. The device Vcc is set: MSP430_GetExtVoltage(), MSP430_VCC(), MSP430_GetCurVCCT()
4. Configuring the JTAG protocol (Spy-bi-Wire 2-Wire JTAG, 4-wire JTAG) is optional. By default the protocol is selected automatically: MSP430_Configure()
5. Connect and identify target device: MSP430_OpenDevice()
6. Return the identified device: MSP430_GetFoundDevice()
7. The device memory is manipulated using:
    - Execute erase operation: MSP430_Erase()
    - Read/write device memory: MSP430_Memory(), MSP430_ReadOutFile(), MSP430_ProgramFile()
    - Execute verify operation: MSP430_VerifyFile(), MSP430_VerifyMem(), MSP430_EraseCheck()
8. The device functionality is manipulated by:
    - Secure device – disable JTAG access: MSP430_Secure()
    - Execute device reset: MSP430_Reset()
    - Start device code execution: MSP430_Run()
    - Stop device code execution: MSP430_State()
9. Shutdown device connection and CDC port: MSP430_Close()
10. Error handling: MSP430_Error_Number(), MSP430_Error_String()

Figure 1 shows the startup flow of an MSP430 debug session using the MSPDdebugStack.

Figure 2 contains the example flow for starting a debug session including all needed error handling executed by the MSP430_Error_Number() and MSP430_Error_String() functions. The MSP430_DLL.chm help file offers detailed information on all library functions, their parameters and return values.

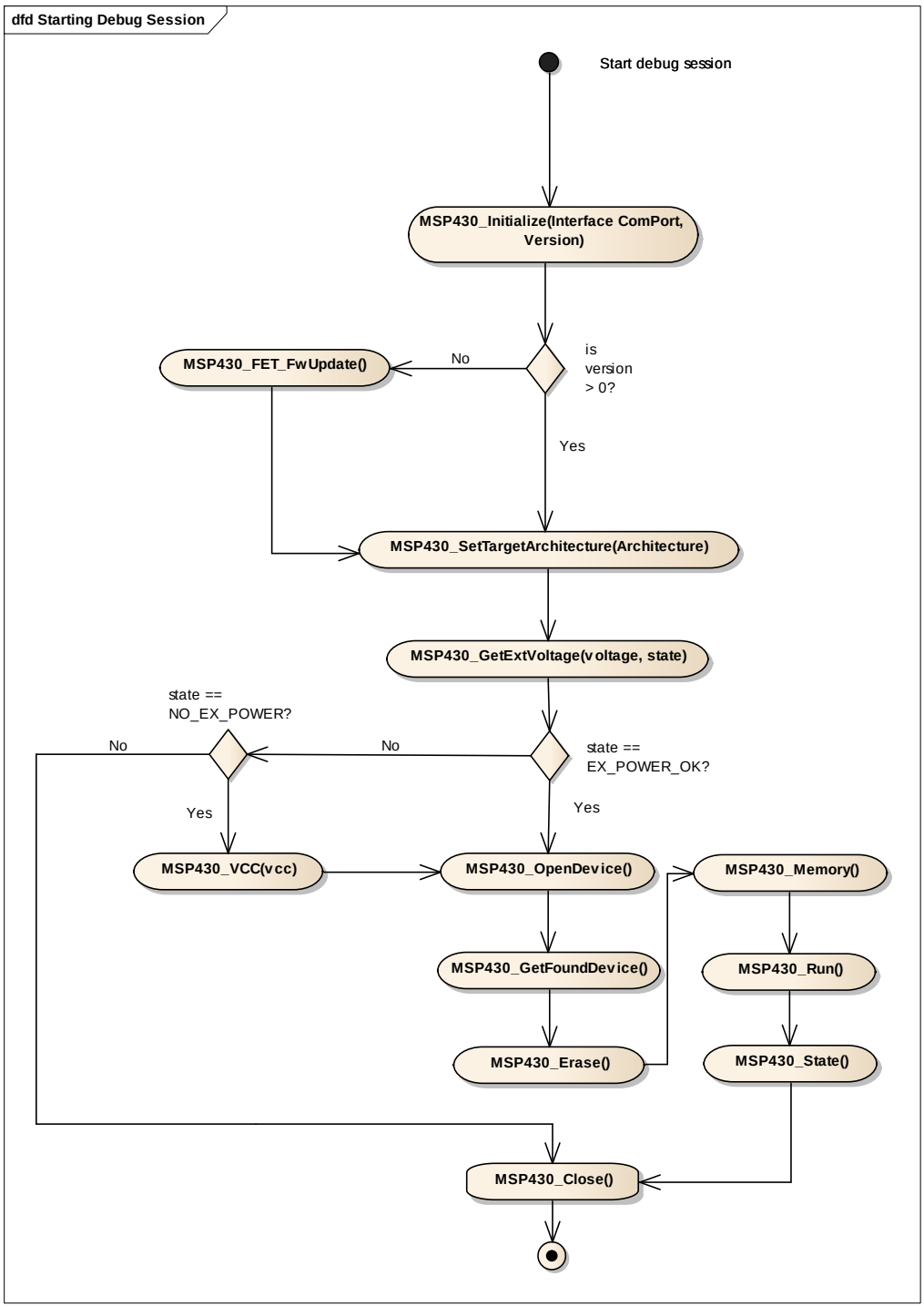

Figure 1: Recommended flow to start an MSP430 debug session

```
#include "stdio.h"
#include "MSP430_FET.h"
#include "MSP430_Debug.h"
#include "MSP430.h"

int32_t lVersion;          // MSPDebugStack version
long verify = 0;  // verify the filetransfer?
int32_t passwordLen = 8;
char password[] = "0x34127856"; // password is sent to device in following order: 0x12 0x34 0x56 0x78

// init JTAG interface – TIUSB will use first connected debugger
printf("MSP430_Initialize()\n");
if(MSP430_Initialize("TIUSB", &lVersion) == STATUS_ERROR)
{
   printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));      // print error string
   MSP430_Close(1);  // close the debug session
}

// Set target architecture
if (MSP430_SetTargetArchitecture(MSP430) == STATUS_ERROR)
{
        printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));              // print error string
        MSP430_Close(1);  // close the debug session and turn VCC off
}

// Check firmware compatibility
if(lVersion < 0)           // firmware outdated?
{
   // perform firmware update
   printf("MSP430_FET_FwUpdate()\n");
   if(MSP430_FET_FwUpdate(NULL, NULL, NULL) == STATUS_ERROR)
   {
        printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));              // print error string
        MSP430_Close(1);  // close the debug session and turn VCC off
   }
}
// power up the target device
printf("MSP430_VCC()\n");
if(MSP430_VCC(3000) == STATUS_ERROR)     // target VCC in millivolts
{
   printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));      // print error string
   MSP430_Close(1);  // close the debug session and turn VCC off
}
// configure interface - this is optional! automatic interface selection is the default
printf("MSP430_Configure()\n");
if(MSP430_Configure(INTERFACE_MODE, AUTOMATIC_IF) == STATUS_ERROR)
{
   printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));      // print error string
   MSP430_Close(1);  // close the debug session and turn VCC off
}
// open the device
printf("MSP430_OpenDevice()\n");
// If the device is password protected, use MSP430_OpenDevice with appropriate password
if(passwordLen > 0)
{
   if(MSP430_OpenDevice("DEVICE_UNKNOWN",password,passwordLen,0,DEVICE_UNKNOWN) ==
   STATUS_ERROR)
   {
        printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));              // print error string
        MSP430_Close(1);  // close the debug session and turn VCC off
```

```
    }
  }
  else
  {
    if(MSP430_OpenDevice("DEVICE_UNKNOWN","",0,0,DEVICE_UNKNOWN) == STATUS_ERROR)
    {
            printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));          // print error string
            MSP430_Close(1);  // close the debug session and turn VCC off
    }
  }
  // program .txt file into device memory (optional)
  printf("MSP430_ProgramFile()\n");
  if(MSP430_ProgramFile("C:\file.txt", ERASE_ALL, verify) == STATUS_ERROR)
  {
    printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));      // print error string
    MSP430_Close(1);  // close the debug session and turn VCC off
  }
  /*************************** debug session is started ***************************/
```

Figure 2: Code example to start a debug session

Attach to a running device

The MSPDdebugStack offers the possibility to connect to a running MSP430 target device without stopping/affecting the target program execution. This feature can be used for debugging an application, which has been running for a while on the target device. During this special startup only the JTAG interface has to be initialized. No reset of the target device will be performed, because it might change the application context of the target device. The running application could contain various information of interest for the debug session (Error states of long runtime errors → stack overflow)

Establishing the physical JTAG connection to the target device is not trivial, especially when the RST signal of the target processor is connected to the JTAG header. A successful connection is subject to stable signals on the JTAG connector (a bouncing signal on the RST pin will definitely perform a reset of the connected microcontroller).

Note: Attach to running target is only available with external power supply. Using internal power supply, which is generated by the USB-FET would reset the device during VCC supply startup.

Following the flow shown in Figure 3 creates the highest probability for successfully attaching to a running target. Figure 4 contains an according code example.

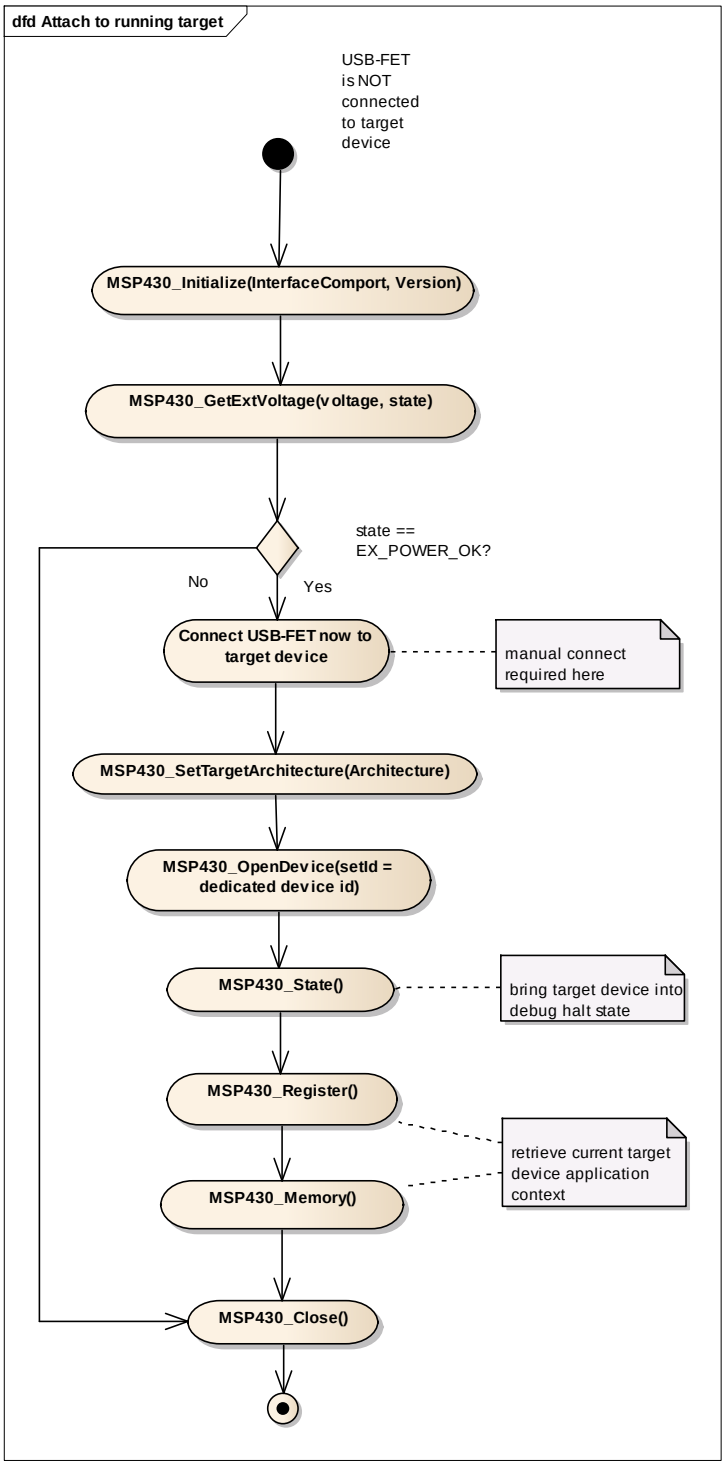

Figure 3: Attach to running target

```
int32_t lVersion, state, pCpuCycles;
DEVICE_T TargetDevice;
// get device information - determine device id
printf("MSP430_GetFoundDevice()\n");
if(MSP430_GetFoundDevice((char*)&TargetDevice, sizeof(TargetDevice.buffer)) == STATUS_ERROR)
{
   printf("%s\n", MSP430_Error_String(MSP430_Error_Number()));
   MSP430_Close(1);
}

// release the target from JTAG control
printf("MSP430_Run(FREE_RUN, release from JTAG)\n");
if(MSP430_Run(FREE_RUN, TRUE) == STATUS_ERROR)
{
   printf("%s\n", MSP430_Error_String(MSP430_Error_Number()));
   MSP430_Close(1);
}

printf("MSP430_Close(VccOff = false)\n");  // close the interface connection
if(MSP430_Close(FALSE) == STATUS_ERROR) // do NOT turn off Vcc power supply
{
   printf("%s\n", MSP430_Error_String(MSP430_Error_Number()));
   MSP430_Close(1);
}

Sleep(100);      // wait a few milliseconds
// initialize the interface again
printf("MSP430_Initialize()\n");
if(MSP430_Initialize("TIUSB", &lVersion) == STATUS_ERROR)
{
   printf("%s\n", MSP430_Error_String(MSP430_Error_Number()));
   MSP430_Close(1);
}

// Set target architecture
if (MSP430_SetTargetArchitecture(MSP430) == STATUS_ERROR)
{
            printf("Error: %s\n", MSP430_Error_String(MSP430_Error_Number()));          // print error string
            MSP430_Close(1);  // close the debug session
}

// attach to the running target with correct device string and/or device id
printf("MSP430_OpenDevice(DeviceNameString,…,…, TargetDevice.id)\n");
if(MSP430_OpenDevice((char*)TargetDevice.string,"", 0, 0, TargetDevice.id) == STATUS_ERROR)
{
   printf("%s\n", MSP430_Error_String(MSP430_Error_Number()));
   MSP430_Close(1);
}

// check CPU state - state should be "RUNNING"
printf("MSP430_State(...,stop = FALSE,...) -> check CPU state\n");
if(MSP430_State(&state, FALSE, &pCpuCycles) == STATUS_ERROR)
{
   printf("%s\n", MSP430_Error_String(MSP430_Error_Number()));
   MSP430_Close(1);
}
```

Figure 4: Code Example for "Attach to running target" – Open a debug session prior to utilizing this code (see. Fig. 2)

Supporting multiple USB-FET Debuggers

The MSPDdebugStack can handle multiple USB-FET debuggers connected to the computer. For this purpose two MSP430 USB-FET support functions are available inside the MSP430.h file.

- MSP430_GetNumberOfUsbIfs()
- MSP430_GetNameOfUsbIf()

Before calling MSP430_Initialize() (open the USB-FET corresponding COM port) the two functions above have to be executed in correct order.

- MSP430_GetNumberOfUsbIfs()
    - First determine how many USB-FETs are connected to the PC system
- MSP430_GetNameOfUsbIf()
    - Get name (e.g. COM5, COM19,…) and status of CDC com port assigned to a certain USB-FET tool/debugger.

After all information about how many and which CDC ports are available on the PC system has been retrieved, a dedicated USB-FET tool can be employed directly by passing the CDC port name to MSP430_Initialize() function e.g. MSP430_Initialize("COM5",…).

Figure 5 shows the typical flow, which is executed to retrieve all needed information about connected USB-FET tools/debuggers.

Figure 6 offers an example code for initializing multiple USB-FETs/debuggers one by one.

Please also refer to the example project MultipleUifs for a possible application implementation proposal.

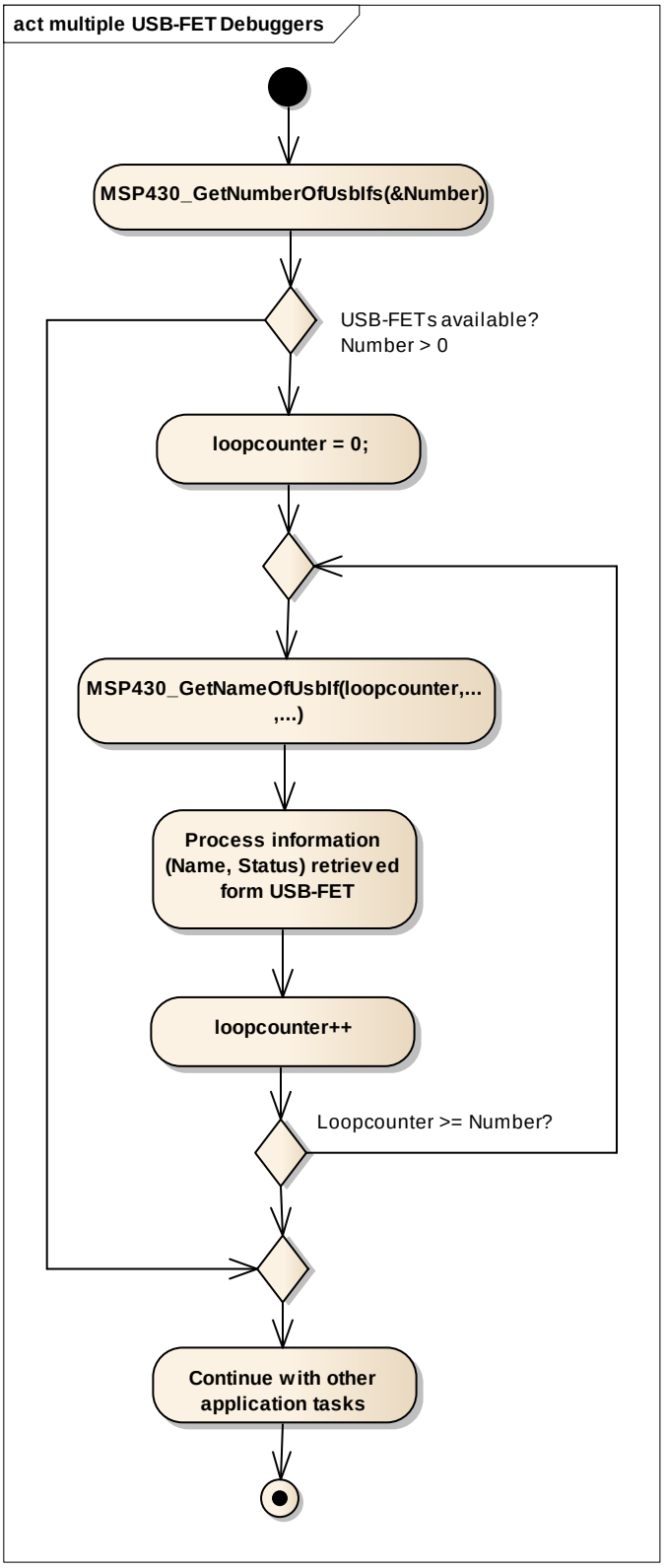

Figure 5: Retrieve info about available USB-FETs/debuggers

```
#include "stdio.h"
#include "MSP430.h"

Int32_t number, count, status, lVersion, lErrorNumber;
char * name;

// determine the number of connected UIFs
printf("MSP430_GetNumberOfUsbIfs()\n");
if(MSP430_GetNumberOfUsbIfs(&number) == STATUS_ERROR)
{
    printf("Error: Could not determine number of UIFs!\n");
    lErrorNumber = MSP430_Error_Number();
    printf("Reason: %s\n", MSP430_Error_String(lErrorNumber));
}
else
{
    printf("Found %d UIF(s).\n", number);
    for(count = 0; count < number; count++)
    {
    // get the com port name
        printf("MSP430_GetNameOfUsbIf()\n");
        if(MSP430_GetNameOfUsbIf(count, &name, &status) == STATUS_ERROR)
        {
            printf("Error: Could not obtain com port name for UIF %d.\n", count+1);
            lErrorNumber = MSP430_Error_Number();
            printf("Reason: %s\n", MSP430_Error_String(lErrorNumber));
        }
        else
        {
            // initialize the interface
            printf("Initializing UIF @ %s.\n", name);
            printf("MSP430_Initialize(UIF %d)\n", count+1);
            if(MSP430_Initialize(name, &lVersion) == STATUS_ERROR)
            {
                lErrorNumber = MSP430_Error_Number();
                printf("Error: %s\n", MSP430_Error_String(lErrorNumber));
            }
            else
            {
                printf("Success!\n");

                // commence with debug session start here...

                // close the interface
                printf("MSP430_Close()\n");
                MSP430_Close(1);
            }
        }
    }
}
```

Figure 6: Code Example for communication with multiple USB-FETs/debuggers

Configuring the JTAG protocol

By default the MSPDdebugStack is configured to perform an automatic protocol scan before starting communication with MSP430 devices. This default configuration can be overwritten manually by using the **INTERFACE_MODE** configuration (refer to MSP430.h file for details). Four different interface modes are available and can be used for debugging the connected MSP430 device.

- **JTAG_IF**: The normal standard 4-wire JTAG communication (Note: Not supported by eZ debuggers)
- **SPYBIWIRE_IF**: Spy-bi-Wire (2-wire) JTAG protocol
- **SPYBIWIREJTAG_IF**: Standard 4-wire JTAG communication for MSP430 devices which also support Spy-bi-Wire (a special entry sequence is needed to switch these MSP430 derivatives into 4-wire mode which cannot be applied to any MSP430 devices) (Note: Not supported by eZ debuggers)
- **AUTOMATIC_IF**: JTAG communication protocol is selected automatically by the MSPDdebugStack (default)

If MSP430_Configure() is called to configure the JTAG protocol manually, it must be done before MSP430_OpenDevice() is called.

Speed up Flash Programming

The API routines MSP430_Erase() and MSP430_Memory() enable manipulation of the target devices Flash/RAM and FRAM Memory.

If Flash memory is programmed by the MSPDdebugStack, the target device RAM is used by the flash programming routines. Because of this the Ram content of the target devices has to be preserved before programming Flash memory. After successfully programming the original RAM content has to be restored.

The above described RAM preserve mechanism is used to allow Flash Memory manipulation during an active debug session without corrupting/changing any RAM content. Anyway, it takes perceivable time to preserve/restore RAM contents. Thus this mechanism might be considered to be not very useful under some circumstances, e.g. during an initial Flash Programming at the beginning of a debug session.

Therefore, the RAM preserve/restore mechanism can be disabled by an additional MSP430_Configure () function call. This additional configuration mode is called **RAM_PRESERVE_MODE**.

The following sequence might be used, e.g. for an initial Flash Programming sequence:

(1) MSP430_Configure(RAM_PRESERVE_MODE, DISABLE);

(2) MSP430_Erase(ERASE_ALL,..,..);

(3) MSP430_Memory(..., ..., ..., WRITE);

(4) MSP430_Memory(..., ..., ..., READ);

..... Flash Programming/Download finished

(n) MSP430_Configure(RAM_PRESERVE_MODE, ENABLE);

Controlling device program execution

The MSPDdebugStack provides additional debugging functions to developers of third party tools for the MSP430. The debugging functions include execution control (free run, run to breakpoints, single step, state, stop, set breakpoint), device control (read/write registers, reset, clock configuration, device configuration), and low-level access to the advanced features of the Enhanced Emulation Module (EEM) that provides such features as complex breakpoints, trace buffers, etc.. The low-level access to EEM registers (namely Read/Write EEM register) is basically kept in the library due to compatibility reasons (EEM API).

Enhanced Emulation Module (EEM) Access – EEM API

The MSPDdebugStack provides an enhanced debug API that allows access to MSP430's Enhanced Emulation Module functionality. Refer to source code of application examples, on how to use the EEM API.

***Note**: Some deprecated API functions are no longer allowed to be called in case EEM API is used. These functions are namely:*

- MSP430_Configure() with parameter 'mode' set to CLK_CNTRL_MODE
- MSP430_Configure() with parameter 'mode' set to MCLK_CNTRL_MODE
- MSP430_State() with parameter 'stop' set to FALSE
- MSP430_EEM_Open()
- MSP430_EEM_Read_Register()
- MSP430_EEM_Read_Register_Test()
- MSP430_EEM_Write_Register()
- MSP430_EEM_Close()

Refer to the detailed documentation in MSP430_EEM.h.

Error handling

All functions of the MSPDebugStack return an indication of success (STATUS_OK) or failure (STATUS_ERROR). If STATUS_ERROR is returned, MSP430_Error_Number() can be used to obtain a detailed error code. MSP430.h contains an enumeration of all error codes, and lists the error codes returned by each API function. MSP430_Error_String() will return the string corresponding to the error code parameter.

STATUS_ERROR is returned at the first error condition. The library typically does not attempt to retry and/or recover from the error condition. It is the responsibility of the application to retry the failed operation, and to possibly implement some sort of "back-out" recovery mechanism.

Miscellaneous

The MSPDdebugStack is a *partially intrusive* tool; accessing the device via JTAG can affect the device (i.e. clocking the Watchdog mechanism). However, steps are taken within the MSPDdebugStack to minimize the effects upon the device caused by JTAG.

IMPORTANT NOTES

Do not unplug the JTAG cable during an active debug session! This might cause unknown device behavior!

MSP-FET430UIF Firmware Update Support

With every new version of MSPDebugStack the firmware of the connected USB JTAG interface might require an update. The library includes a binary image of the corresponding USB-FET/debugger firmware. Calling MSP430_FET_FwUpdate() as described in the flow chart below assures consistency between USB-FET firmware and loaded MSPDebugStack.

With the release of the version 3 of the MSPDebugStack (formerly DLLv3) the firmware has changed and now consists of different independent parts (USB communication Core, JTAG stack, low level debugger hardware access VCC generation and UART backchannel), which can be updated independently. Therefore it was necessary to extend the firmware update mechanism to execute DLLv2 to DLLv3 updates. As you can see in figure 7, MSP430_Initialize() returns either -3, -2 , -1 or the actual MSPDebugStack firmware Version.

In case MSP430_Initialize() returns -3, a major firmware version update (DLLv2 to DLLv3) is required. Afterwards MSP430_FET_FwUpdate() has to be called again to update the firmware with the MSPDebugStack internal binary image. In this special update case a given update file will be ignored.

If MSP430_Initialize() returns -1, the USB-FET firmware has been already updated to DLLv3 firmware. In this case either the communication core , JTAG stack, or HIL module does not match the MSPDebugStack version. By calling the MSP430_FET_FwUpdate() function the internal library binary images are used for USB-FET update.

If MSP430_Initialize() returns -2, the USB-FET firmware needs recovery because of major system corruption. A corrupted USB-FET will always enumerate as HID-FET. The MSPDebugStack will detect the HID-FET and raise a message, that a connected USB-FET needs recovery. Please refer to "Figure 8 HID recovery flow" for details.

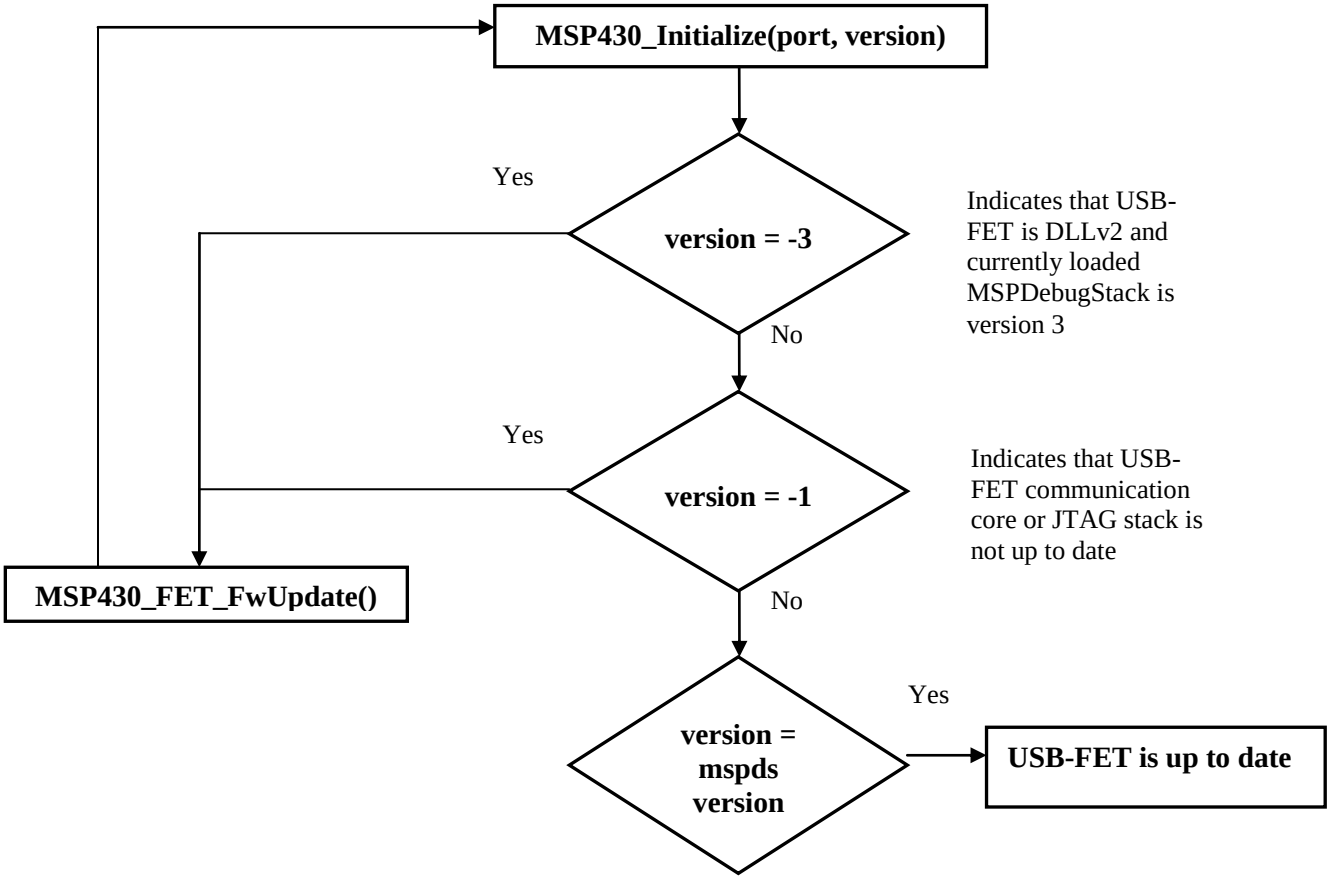

Figure 7: General firmware Update Flow

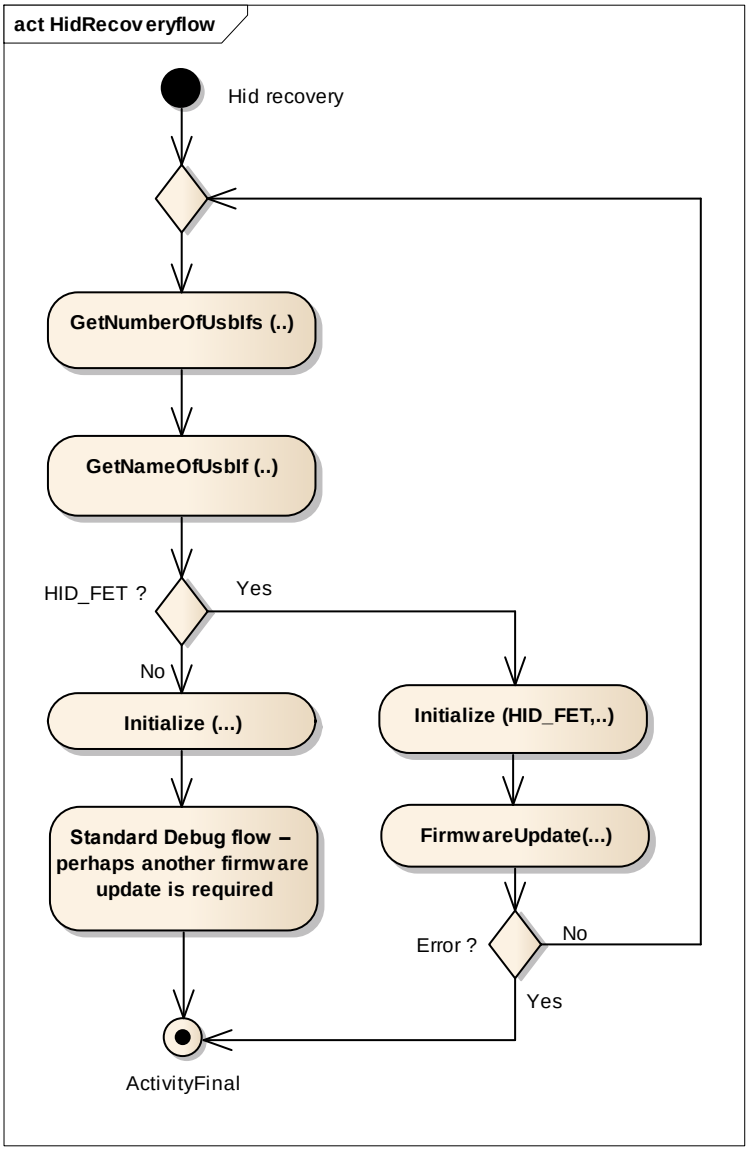

Figure 8: USB-FET HID recovery flow

Firmware update with Update-Tool

An update of the MSP-FET430UIF firmware without an IDE can be executed using the command line based Update-Tool. This tool can only be used with the MSP-FET430UIF.
The Update-Tool also provides the possibility of firmware up/downgrade between major firmware versions.
For detailed information how to update your MSP-FET430UIF please refer to
http://processors.wiki.ti.com/index.php/MSPDS_Debugger_Up-_and_Downgrade

Please refer to Appendix B to determine if you are using an MSP-FET430UIF revision 1.3 because it requires additional update steps

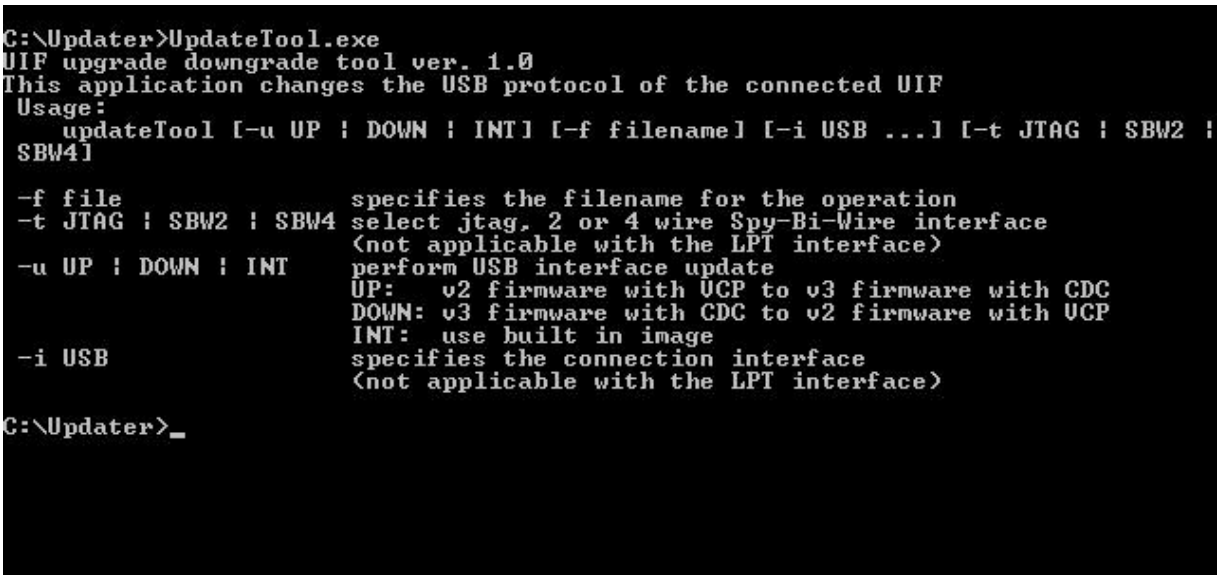

Figure 9: Update-Tool

Available commands:

updateTool –u **UP**: updates the UIF's major firmware version (e.g. version 2 to 3)

updateTool –u **DOWN**: downgrades UIF's major firmware version through the binary image stored in Uifv3Downgrader.txt

updateTool –u **INT**: updates the UIF with the MSPDebugStack internal firmware image

Important Note: Make sure that the CDC driver is already installed before performing a major firmware update. Also, a file called "CDC.log" with the content "True" must be placed in the same folder as the MSPDebugStack library. It indicates that the CDC driver was installed successfully. Otherwise the update process will return an update error.

Additional update step for MSP-FET430UIF hardware revision 1.3

After calling updateTool –u UP the update process starts and you can see the following command line window

```
C:\Updater>UpdateTool -u UP
   Initialize: done
   MSP430_FET_GetFwVersion()
   Firmware Version: 20409001
Status: Starting firmware update with built in image!

   Initializing bootloader...
   Erasing interrupt vectors...
   Erasing firmware...
   Programming new firmware...
   100 percent done

   Finishing...
```

On finishing, the TUSB3410 should be reset and the UIF should show up as a CDC device.
Due to the reason mentioned in Appendix B that is not possible, so it is necessary to disconnect the MSP-FET430UIF and reconnect it.
After doing so, the update process will continue.

```
   Initializing bootloader...
   Erasing firmware...
   Programming new firmware...
   100 percent done

   Finishing...
   Update complete.


   Update complete.


 Status: Firmware update performed successfully
```

Supporting eZ430 emulator dongles

There are several different versions of the eZ430 emulator.

- eZ430-RF2500: The dongle enumerates as a Human Interface Device (HID). The HID class driver is part of the Windows operation system, thus the enumeration does not require any user interaction. The HID interface is used for the JTAG communication to the target device. Besides the HID channel the dongle also tries to enumerate a Virtual Com Port (which is called MSP430 Application UART).This driver is based as well on the Communication Device Class (CDC) interface. This CDC driver class is also part of the Windows operating system but it requires an INF file for installation. The provided INF file (430CDC.inf, can be found in folder Driver/Inf) is certified for MS-Windows operating systems XP32, XP64, Vista32, Vista64, Win7-32 and Win7-64. The folder Driver/Inf contains a subfolder PreinstallCDC. This subfolder contains an example source code that shows how to install the INF file on a MS-Windows PC. It is recommended to install the INF file like described in the example. If not done like that the Windows Hardware Wizard will pop up as soon as the user connects the tools to the PC. Afterwards the user has to manually point the Wizard to the correct location of the INF file.
- Other supported eZ430 tools that make use of the HID interface are the eZ430 Chronos, the Launchpad and the MSP-EXP430FR5739 FRAM Experimenter's board.

Application Examples

The MSPDebugStack Developer's Package features a series of example projects to illustrate the usage of different functions. It is recommended to use Windows and Visual Studio 2013 for building these examples projects. After the **rebuild**, the executables can be found in ApplicationExample/Executables. Refer to the source code for details on how to call API functions and correctly pass parameters to those functions.

Example

Example is a simple example project that demonstrates how the basic functions of the MSPDebugStack are called to initialize the interface, identify and configure the device, manipulate the device memory (erase, program, verify, read), secure the device, reset the device, close the interface, and handle error conditions. Refer to the source file *Example.c*.

ExampleDebug

ExampleDebug is an example project that demonstrates how the functions of the MSPDebugStack are called to initialize the interface, identify and configure the device, manipulate the device memory (erase, program, verify, read), read the device registers, set device breakpoints, run the device (free, with breakpoints, single step), reset the device, close the interface, and handle error conditions. Refer to the source file *Example Debug.c*.

UifUpdate

UifUpdate is an example project that demonstrates how to perform an USB-FET firmware update by calling MSP430_FET_FwUpdate() including handling of the notify callback mechanism during the update process. Refer to the MSPDebugStack API documentation of MSP430_FET_FwUpdate() for technical details.

MultipleUifs

MultipleUifs is an example project that demonstrates how to support multiple MSP-FET430UIF tools connected to one PC system. The example project comes along with a GUI that shows a possible support implementation.

Appendix A. Installation of CDC for USB-FET debuggers

The UIF tries to enumerate a Virtual Com Port, which is based on Communication Device Class (CDC) driver. This CDC driver class is part of the Windows operating system but it requires an INF file for installation.
After plugging in the USB-FET, Windows recognizes a new hardware called MSP-FET430UIF or MSP Debug interface and the following dialog appears.

Figure 10: New hardware

Afterwards the hardware wizard opens a new dialog window.
If CCS version 5/6 or IAR IDE is already installed, select "Install the software automatically". If no IDE has been installed select the msp430tools.inf, which is part of Developer's package (MSP430_DLL_Developer_Package_Rev_x_x_x_x\Driver\CDC) and install the driver manually.

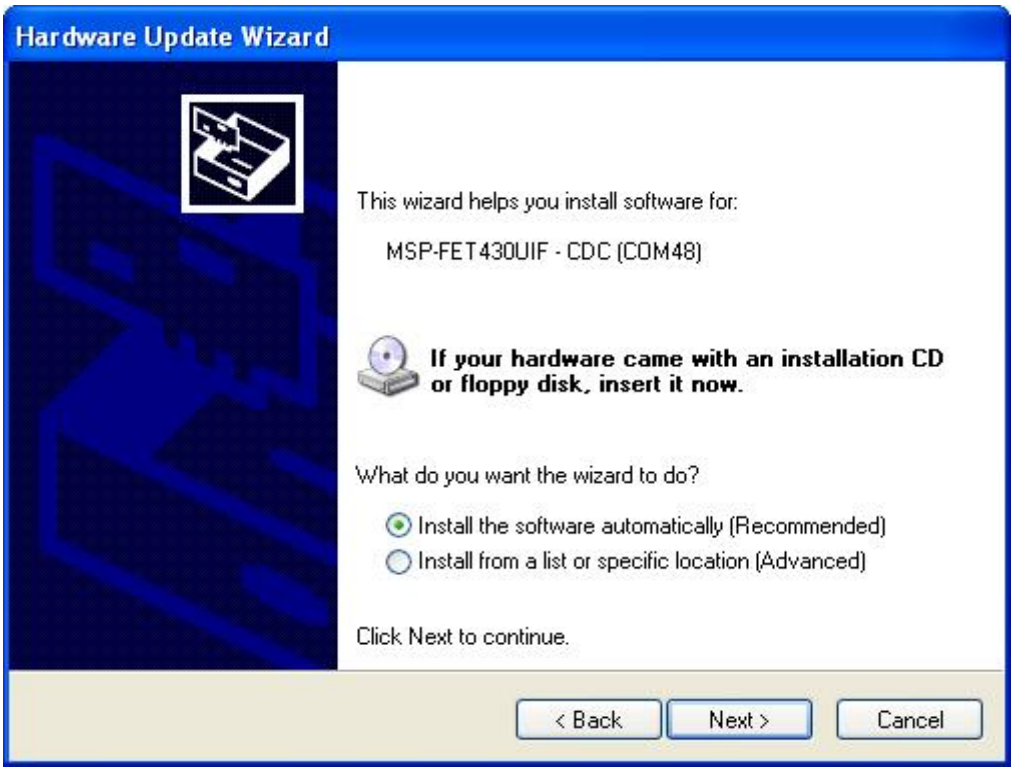

Figure 11: Update Wizard

Appendix B. Update MSP-FET430UIF with hardware revision 1.3

If you are using a MSP-FET430UIF with hardware revision 1.3 your update process includes one additional step, due to the fact that it is not possible to reset the TUSB3410 USB port controller during firmware update.
Without a reset the TUSB can't change the VCP protocol to CDC and afterwards install the new communication core and JTAG stack. It is necessary to reset the device manually by disconnecting the MSP-FET430UIF and connect it to the PC again.
For IDE specific information on how to update an MSP-FET430UIF revision 1.3 please refer to the MSP-FET430UIF Debug FAQ (CCS > v5.1 and IAR EW > 5.40)

First you have to make sure that you are using an MSP-FET430UIF with hardware revision 1.3. As you can see in Figure 11 and Figure 12 revision 1.3 has a CE sign on the front and no label with a revision number on the rear side.

Figure 12: UIF Revision 1.3

Figure 13: UIF Revision 1.4

A.1 References

TUSB3410 RS232/IrDA Serial-to-USB Converter
http://focus.ti.com/docs/prod/folders/print/tusb3410.html

MSPDebugStack Software Tools
http://www.ti.com/tool/mspds