USING THE C8051FXXX/TXXX ON-CHIP INTERFACE UTILITIES DLL

Relevant Devices

This application note applies to all Silicon Laboratories C8051Fxxx/Txxx devices.

1. Introduction

The Interface Utilities Dynamic Link Library (DLL) provides the following functionality: download an Intel hex file to Flash; connect and disconnect to a C8051Fxxx/Txxx processor; “Run” and “Halt” the microprocessor; and read and write internal, external, and code memory spaces. All this functionality is provided via a PC’s COM port or USB port and a Silicon Laboratories Debug Adapter.

When using C++, the functions described in this document must be declared as imported functions. Add the **C++ Prototype** to the header file (*.h) of the source file (*.cpp) from which the function will be called.

The procedures and guidelines presented in this document illustrate how to link the Interface Utilities DLL to a client executable. A DLL is not a standalone application. It is a library of exported functions that are linked at run-time and called by a Microsoft Windows® application. To write a client using Visual Basic, please refer to Section 14 on page 20 for more information.

An example Interface Utilities application (including program source code) is provided along with the Interface Utilities DLL as a means of testing or using the Utilities DLL as a base application. The Interface Utilities application uses implicit linking, which requires the DLL to be placed in a specific directory (see Section 11 on page 16).

Note: In version 2.60 major changes in the generation of error codes were implemented. See Section 12 on page 16 for the new error code definition and function.

2. Files and Compatibility

The latest versions of the “SiUtil.dll” and “SiUtil.lib” are available at www.silabs.com. The DLL is a Win32 MFC Regular DLL, meaning it uses the Microsoft Foundation Class (MFC) libraries. It can be loaded by any Win32 programming environment, and only exports “C” style functions. Two versions are currently available, one in which the MFC library is statically linked in the DLL, and another version in which the MFC library is dynamically linked in the DLL.

The statically linked MFC version includes a copy of all the MFC library code it needs and is thus self contained. No external MFC linking is required. With the MFC library code included, the statically linked DLL will be larger in size than the dynamically linked version.

Although the dynamically linked DLL is smaller in size than the static version, the dynamically linked DLL requires that files “MFC42.dll” and “MSVCRT.dll” be present on the target machine. This is not a problem if the client program is dynamically linked to the same version (Version 4.2) or newer of the MFC library (i.e., uses MFC as a shared library). The required MFC DLLs, “MFC42.dll” and “MSVCRT.dll”, are provided along with the dynamically linked MFC version of the Utility Programmer DLL. Do not replace equivalent or newer versions of these files if they are already present on the target machine.

3. Communications Functions

The following communication functions are available for use in the Interface Utilities DLL.

`Connect()`	- Connects to a target C8051Fxxx device using a Serial Adapter.
`Disconnect()`	- Disconnects from a target C8051Fxxx device using a Serial Adapter.
`ConnectUSB()`	- Connects to a target C8051Fxxx device using a USB Debug Adapter.
`DisconnectUSB()`	- Disconnects from a target C8051Fxxx device using a USB Debug Adapter.
`Connected()`	- Returns the connection state of the target C8051Fxxx device.

3.1. Connect()

Description: This function is used to connect to a target C8051Fxxx device using a Serial Adapter. Establishing a valid connection is necessary for all memory operations to succeed.

Supported Debug Adapters: Serial Adapter

C++ Prototype:
```
extern “C” __declspec(dllimport) HRESULT__stdcall Connect
(int nComPort=1, int nDisableDialogBoxes=0, int nECprotocol=0,
int nBaudRateIndex=0);
```

Parameters:

1. *nComPort*—Target COM port to establish a connection. The default is 1.
2. *nDisableDialogBoxes*—Disable (1) or enable (0) dialogs boxes within the DLL. The default is 0.
3. *nECprotocol*—Connection protocol used by the target device; JTAG (0) or Silicon Laboratories 2-Wire (C2) (1). The default is 0. The C2 interface is used with C8051F3xx derivative devices and the JTAG interface is used with C8051F0xx, C8051F1xx, and C8051F2xx derivative devices.
4. *nBaudRateIndex*—Target baud rate to establish a connection; Autobaud detection (0), 115200 (1), 57600 (2), 38400 (3), 9600 (4) or 2400 (5). The default is 0.

Return Value: See Section 12 on page 16 for error return information.

3.2. Disconnect()

Description: This function is used to disconnect from a target C8051Fxxx device using a Serial Adapter.

Supported Debug Adapters: Serial Adapter

C++ Prototype:
```
extern “C” __declspec(dllimport)
HRESULT__stdcall Disconnect(int nComPort=1);
```

Parameters:

1. *nComPort* - COM port in use by current connection. The default is 1.

Return Value: See Section 12 on page 16 for error return information.

3.3. ConnectUSB()

Description: This function is used to connect to a target C8051Fxxx device using a USB Debug Adapter. Establishing a valid connection is necessary for all memory operations to succeed.

Supported Debug Adapters: USB Debug Adapter

C++ Prototype:

```
extern “C” __declspec(dllimport) HRESULT__stdcall ConnectUSB(
const char * sSerialNum=””, int nECprotocol=0, int nPowerTarget=0,
int nDisableDialogBoxes=0);
```

Parameters:

1. *sSerialNumber*—The serial number of the USB Debug Adapter. See Section 8 for information on obtaining the serial number of each USB Debug Adapter connected. If only one USB Debug Adapter is connected, an empty string can be used. The default is an empty string.
2. *nECprotocol*—Connection protocol used by the target device; JTAG (0) or Silicon Laboratories 2-Wire (C2) (1). The default is 0. The C2 interface is used with C8051F3xx derivative devices and the JTAG interface is used with C8051F0xx, C8051F1xx, and C8051F2xx derivative devices.
3. *nPowerTarget*—If this parameter is set to 1, the USB Debug Adapter will be configured to continue supplying power after it has been disconnected from the target device. The default is 0, configuring the adapter to discontinue supplying power when disconnected.
4. *nDisableDialogBoxes*—Disable (1) or enable (0) dialogs boxes within the DLL. The default is 0.

Return Value: See Section 12 on page 16 for error return information.

3.4. DisconnectUSB()

Description: This function is used to disconnect from a target C8051Fxxx device using a USB Debug Adapter.

Supported Debug Adapters: USB Debug Adapter

C++ Prototype: `extern “C” __declspec(dllimport) HRESULT__stdcall DisconnectUSB();`

Parameters: none

Return Value: See Section 12 on page 16 for error return information.

3.5. Connected()

Description: This function returns a value representing the connection state of the target C8051Fxxx.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype: `extern “C” __declspec(dllimport) BOOL__stdcall Connected();`

Parameters: none

Return Value: = 0 (not connected) or
= 1 (connected)

4. Program Interface Functions

The following program interface functions are available for use in the Interface Utilities DLL.

`Download()`	- Downloads a hex file to the target C8051Fxxx device.
`ISupportBanking()`	- Checks to see if banking is supported.
`GetSAFirmwareVersion()`	- Returns the Serial Adapter firmware version.
`GetUSBFirmwareVersion()`	- Returns the USB Debug Adapter firmware version.
`GetDLLVersion()`	- Returns the Utilities DLL version.
`GetDeviceName()`	- Returns the name of the target C8051Fxxx device.

4.1. Download()

Description: This function is used to download a hex file to a target C8051Fxxx device. After a successful exit from the *Download()* function, the target C8051xxx will be in a "Halt" state. If the device is left in the "Halt" state, it will not begin code execution until the device is reset by a Power-On reset or by a *SetTargetGo()* DLL function call.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern “C” __declspec(dllimport) HRESULT__stdcall Download(
char * sDownloadFile, int nDeviceErase=0, int nDisableDialogBoxes=0,
int nDownloadScratchPadSFLE=0, int nBankSelect=-1, int nLockFlash=0),
BOOL bPersistFlash=1);
```

Parameters:

1. *sDownloadFile*—A character pointer to the beginning of a character array (string) containing the full path and filename of the file to be downloaded.
2. *nDevice Erase*—When set to 1, performs a device erase before the download initiates. If set to 0, the part will not be erased. A device erase will erase the entire contents of the device's Flash. The default is 0.
3. *nDisableDialogBoxes*—Disable (1) or enable (0) dialogs boxes within the DLL. The default is 0.
4. *nDownloadScratchPadSFLE*—This parameter is only for use with devices that have a Scratchpad Flash memory block. Currently, this includes the C8051F02x, C8051F04x, C8051F06x, and C8051F12x devices. For all other devices, this parameter should be left in the default state. Set this parameter to 1 in order to download to Scratchpad memory. When accessing and downloading to Scratchpad memory, the only valid address range is 0x0000 to 0x007F. The default is 0.
5. *nBankSelect*—This parameter is only for use with C8051F12x devices. For all other devices, this parameter should be left in the default state. When using a C8051F12x derivative, set this parameter to 1, 2, or 3 in order to download to a specific bank. The default is –1.
6. *nLockFlash*—Set this parameter to 1 to lock the Flash following the download. If Flash is locked, the DLL will no longer be able to connect to the device.
7. *bPersistFlash*—If set to 1, the contents of Flash will be read prior to programming. Flash pages are erased prior to programming. If the pages to be programmed contain any data in Flash that need to be preserved, then set this parameter to 1.

Return Value: See Section 12 on page 16 for error return information.

4.2. ISupportBanking()

Description: This function checks to see if banking is supported.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern “C” __declspec(dllimport) HRESULT__stdcall ISupportBanking(
int * nSupportedBanks);
```

Parameters: **1.** *nSupportedBanks*—The *ISupportBanking()* function expects to receive a valid pointer to an integer value. The *ISupportBanking()* function will set *nSupportedBanks* to the number of banks supported on the target device, or 0 if none exist.

Return Value: See Section 12 on page 16 for error return information.

4.3. GetSAFirmwareVersion()

Description: This function is used to retrieve the version of the Serial Adapter firmware.

Supported Debug Adapters: Serial Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) int __stdcall GetSAFirmwareVersion();
```

Parameters: none

Return Value: Serial Adapter Firmware version

4.4. GetUSBFirmwareVersion()

Description: This function is used to retrieve the version of the USB Debug Adapter firmware.

Supported Debug Adapters: USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) int __stdcall GetUSBFirmwareVersion();
```

Parameters: none

Return Value: USB Debug Adapter Firmware version

4.5. GetDLLVersion()

Description: This function returns the current version of the Utilities DLL.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) char* __stdcall GetDLLVersion();
```

Parameters: none

Return Value: A string containing the Utilities DLL version

4.6. GetDeviceName()

Description: This function returns the name of the target C8051Fxxx device that is currently supported.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall GetDeviceName(
const char **psDeviceName);
```

Parameters: 1. *psDeviceName* - A pointer to a character string location where the device name will be copied.

Return Value: See Section 12 on page 16 for error return information.

5. Get Memory Functions

The following functions for reading device memory locations are available for use in the Interface Utilities DLL.

`GetRAMMemory()`	- Read RAM memory from a specified address.
`GetXRAMMemory()`	- Read XRAM memory from a specified address.
`GetCodeMemory()`	- Read Code memory from a specified address.

The “GetMemory” functions all expect to receive a pointer to an initialized, unsigned char (BYTE) array of *nLength* as the first parameter. If the “GetMemory” functions complete successfully, the *ptrMem* variable will contain the requested memory.

The following example shows how to properly initialize an array in C++:

```
unsigned char* ptrMem;

ptrMem = new unsigned char[length]; //Assumes that length has been declared
                                    //and set elsewhere

// next populate the array with the bytes to write in memory
```

Alternatively:

```
BYTE ptrMem[10] = {0x00};  // Must initialize the array prior to passing
                           // it into the DLL
```

5.1. GetRAMMemory()

Description: This function will read the requested memory from the Internal Data Address Space. The requested RAM memory must be located in the target device’s Internal Data Address Space.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern “C” __declspec(dllimport) HRESULT__stdcall GetRAMMemory(
BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);
```

Parameters:
1. *ptrMem*—Pointer to the receive buffer; an initialized, unsigned char (BYTE) array of length *nLength*. If the function completes successfully, the *ptrMem* variable will contain the requested memory.
2. *wStartAddress*—The start address of the memory location to be referenced.
3. *nLength*—The number of bytes to read from memory.

Return Value: See Section 12 on page 16 for error return information.

5.2. GetXRAMMemory()

Description: This function will read the requested memory from the External Data Address Space. The requested XRAM memory must be located in the target device’s External Data Address Space. Pay special attention to insure proper referencing of the External Data Address Space.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern “C” __declspec(dllimport) HRESULT__stdcall GetXRAMMemory(
BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);
```

Parameters:

1. *ptrMem*—Pointer to the receive buffer; an initialized, unsigned char (BYTE) array of length *nLength*. If the function completes successfully, the *ptrMem* variable will contain the requested memory.
2. *wStartAddress*—The start address of the memory location to be referenced.
3. *nLength*—The number of bytes to read from memory.

Return Value: See Section 12 on page 16 for error return information.

5.3. GetCodeMemory()

Description: This function will read the requested memory from the program memory space, including the lock byte(s). The requested Code memory must be located in the target device’s program memory space. Use caution when reading from a sector that has been read locked. Reading from a sector that has been read locked will always return 0s. Also, reading from the reserved space is not allowed and will return an error.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern “C” __declspec(dllimport) HRESULT__stdcall GetCodeMemory(
BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);
```

Parameters:

1. *ptrMem*—Pointer to the receive buffer, an initialized unsigned char (BYTE) array of length *nLength*. If the function completes successfully, the *ptrMem* variable will contain the requested memory.
2. *wStartAddress*—The start address of the memory location to be referenced.
3. *nLength*—The number of bytes to read from memory.

Return Value: See Section 12 on page 16 for error return information.

6. Set Memory Functions

The following functions for writing to device memory locations are available for use in the Interface Utilities DLL.

`SetRAMMemory()`	- Writes value to a specified address in RAM memory.
`SetXRAMMemory()`	- Writes value to a specified address in XRAM memory.
`SetCodeMemory()`	- Writes value to a specified address in code memory.

The “SetMemory” functions expect to receive a pointer to an initialized unsigned char (BYTE) array of *nLength* as the first parameter. This array should contain *nLength* number of elements initialized prior to calling into the DLL’s “SetMemory” functions. If the “SetMemory” functions complete successfully, the *ptrMem* variable will have successfully programmed the requested memory.

The following example shows how to properly initialize an array in C++:

```
unsigned char* ptrMem;

ptrMem = new unsigned char[length]; //Assumes that length has been declared
                                    //and set elsewhere

// next populate your array with the bytes that you want to set in memory
```

Alternatively:

```
// Must initialize the array prior to calling the DLL
BYTE ptrMem[10] = {0x00, 0x14, 0xAE, 0x50, 0xAD, 0x66, 0x01, 0x05, 0x77, 0xFF};
```

6.1. SetRAMMemory()

Description: This function will write to memory in the Internal Data Address Space. The target RAM memory must be located in the target device’s Internal Data Address Space.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall SetRAMMemory(
BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);
```

Parameters:
1. *ptrMem*—Pointer to the transmit buffer, an initialized unsigned char (BYTE) array of length *nLength*. If the function completes successfully, the *ptrMem* variable will have successfully programmed the requested memory.
2. *wStartAddress*—The start address of the memory location to be referenced.
3. *nLength* —The length of memory to be read.

Return Value: See Section 12 on page 16 for error return information.

6.2. SetXRAMMemory()

Description: This function will write to memory in the External Data Address Space. The target XRAM memory must be located in the target device's External Data Address Space. Pay special attention to insure proper referencing of the External Data Address Space.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall SetXRAMMemory(
BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);
```

Parameters:

1. *ptrMem*—Pointer to the transmit buffer, an initialized unsigned char (BYTE) array of length *nLength*. If the function completes successfully, the *ptrMem* variable will have successfully programmed the requested memory.
2. *wStartAddress*—The start address of the memory location to be referenced.
3. *nLength*—The length of memory to be read.

Return Value: See Section 12 on page 16 for error return information.

6.3. SetCodeMemory()

Description: This function will write to memory in the program memory space. If a client tries to write more than one page of data at one time, or tries to write to the reserved area of Flash or to a write/erase locked sector, the write function will not complete successfully. To write to a read/write lock byte, *wStartAddress* should correspond to the device read/write lock byte address and *nLength* should have a value of 1 or 2. Please refer to the relevant device data sheets for additional information. If *SetCodeMemory()* completes successfully, only the specified range, *wStartAddress* + *nLength*, will have successfully been written. All other values within the page will retain their initial values.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall SetCodeMemory(
BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength,
int nDisableDialogs=0);
```

Parameters:

1. *ptrMem*—Pointer to the transmit buffer, an initialized unsigned char (BYTE) array of length *nLength*. If the function completes successfully, the *ptrMem* variable will have successfully programmed the requested memory.
2. *wStartAddress*—The start address of the memory location to be referenced.
3. *nLength*—The length of memory to be read.
4. *nDisableDialogs*—Disable (1) or enable (0) dialogs boxes within the DLL. The default is 0.

Return Value: See Section 12 on page 16 for error return information.

7. Target Control Functions

The following functions for controlling a device are available for use in the Interface Utilities DLL.

`SetTargetGo()`	- Puts the target C8051Fxxx device in a “Run” state.
`SetTargetHalt()`	- Puts the target C8051Fxxx device in a “Halt” state.

7.1. SetTargetGo()

Description: After a successful exit from the *SetTargetGo()* function, the target C8051xxx will be in a “Run” state.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype: `extern “C” __declspec(dllimport) HRESULT__stdcall SetTargetGo();`

Parameters: none

Return Value: See Section 12 on page 16 for error return information.

7.2. SetTargetHalt()

Description: After a successful exit from the *SetTargetHalt()* function, the target C8051xxx will be in a “Halt” state.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype: `extern “C” __declspec(dllimport) HRESULT__stdcall SetTargetHalt();`

Parameters: none

Return Value: See Section "12. Test Results" on page 16 for error return information.

8. USB Debug Adapter Communication Functions

The following functions are available in the Interface Utilities DLL to query and configure a USB Debug Adapter. These functions are useful when more than one USB Debug Adapter is connected to the PC. Call *USBDebugDevices()* first to determine the number of USB Debug Adapters connected. Next, use *GetUSBDeviceSN()* to determine the serial number of each adapter. The serial number can then be used with other functions to communicate with an individual USB Debug Adapter.

`USBDebugDevices()`	- Determines how many USB Debug Adapters are present.
`GetUSBDeviceSN()`	- Obtains the serial number of the enumerated USB Debug Adapters.
`GetUSBDLLVersion()`	- Returns the version of the USB Debug Adapter driver file.

8.1. USBDebugDevices()

Description: This function will query the USB bus and identify how many USB Debug Adapter devices are present.

Supported Debug Adapters: USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall USBDebugDevices(
DWORD * dwDevices);
```

Parameters: **1.** *dwDevices*—A pointer to the value that will contain the number of devices that are found.

Return Value: See Section 12 on page 16 for error return information.

8.2. GetUSBDeviceSN()

Description: This function obtains a serial number string for the USB Debug Adapter specified by an index passed in the dw*DeviceNum* parameter. The index of the first device is 0 and the index of the last device is the value returned by *USBDebugDevices()* – 1.

Supported Debug Adapters: USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall GetUSBDeviceSN(
DWORD dwDeviceNum, const char ** psSerialNum);
```

Parameters:

1. *dwDeviceNum*—Index of the device for which the serial number is desired. To obtain the serial number of the first device, use 0.

2. *psSerialNum*—A pointer to a character string location where the serial number will be copied.

Return Value: See Section 12 on page 16 for error return information.

8.3. GetUSBDLLVersion()

Description: This function will return the version of the driver for the USB Debug Adapter.

Supported Debug Adapters: USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall GetUSBDLLVersion(
const char ** pVersionString);
```

Parameters: **1.** *pVersionString*—Pointer to a character string location where the version string of the USB Debug Adapter driver will be copied.

Return Value: See Section 12 on page 16 for error return information.

9. Stand-Alone Functions

The following stand-alone functions are available for use in the Interface Utilities DLL. These functions do not require the use of the “Connect” or “Disconnect” functions and do not interact with any other functions.

`FLASHErase()`	- Erase the Flash program memory using a Serial Adapter.
`FLASHEraseUSB()`	- Erase the Flash program memory using a USB Debug Adapter.

9.1. FLASHErase()

Description: This function is used to erase the Flash program memory of a device using a Serial Adapter. This function must be used to “unlock” a device whose Flash read and/or write lock bytes have been written.

Supported Debug Adapters: Serial Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall FLASHErase(
int nComPort=1, int nDisableDialogBoxes=0, int nECprotocol=0);
```

Parameters:
1. *nComPort*—Target COM port to establish a connection. The default is 1.
2. *nDisableDialogBoxes*—Disable (1) or enable (0) dialogs boxes within the DLL. The default is 0.
3. *nECprotocol*—Connection protocol used by the target device; JTAG (0) or Silicon Laboratories 2-Wire (C2) (1). The default is 0. The C2 interface is used with C8051F3xx derivative devices and the JTAG interface is used with C8051F0xx, C8051F1xx, and C8051F2xx derivative devices.

Return Value: See Section 12 on page 16 for error return information.

9.2. FLASHEraseUSB()

Description: This function is used to erase the Flash program memory of a device using a USB Debug Adapter. This function must be used to “unlock” a device whose Flash read and/or write lock bytes have been written.

Supported Debug Adapters: USB Debug Adapter

C++ Prototype:
```
extern "C" __declspec(dllimport) HRESULT__stdcall FLASHEraseUSB(
const char * sSerialNum, int nDisableDialogBoxes=0, int nECprotocol=0);
```

Parameters:
1. *sSerialNumber*—The serial number of the USB Debug Adapter. See Section 8 for information on obtaining the serial number of each USB Debug Adapter connected. If only one USB Debug Adapter is connected, an empty string can be used. The default is an empty string.
2. *nDisableDialogBoxes*—Disable (1) or enable (0) dialogs boxes within the DLL. The default is 0.
3. *nECprotocol*—Connection protocol used by the target device; JTAG (0) or Silicon Laboratories 2-Wire (C2) (1). The default is 0. The C2 interface is used with C8051F3xx derivative devices and the JTAG interface is used with C8051F0xx, C8051F1xx, and C8051F2xx derivative devices.

Return Value: See Section 12 on page 16 for error return information.

10. Multi-Device JTAG Programming

Multiple Silicon Laboratories C8051Fxxx devices that use the JTAG communications protocol can be connected and programmed in a JTAG chain. Currently this includes the C8051F0xx, C8051F1xx, and C8051F2xx device families. Configure the JTAG chain as shown in Figure 1. It will be necessary to know the instruction register length of each device in the JTAG chain. The instruction register length of all Silicon Laboratories JTAG devices is 16 bits. The following functions to configure and connect to a C8051Fxxx device on a JTAG chain is available for use in the Interface Utilities DLL.

`SetJTAGDeviceAndConnect()`	- Configure a connection to a C8051Fxxx device on a JTAG chain using a Serial Adapter.
`SetJTAGDeviceAndConnectUSB()`	- Configure a connection to a C8051Fxxx device on a JTAG chain using a USB Debug Adapter.

Once the “SetJTAGDeviceAndConnect” function has returned successfully and a connection has been established with a Silicon Laboratories device in the JTAG chain, any of the previously mentioned Communication, Program Interface, and memory functions may be used on the isolated device. When finished interfacing with the device, call a “Disconnect” function as usual to disconnect from the device.

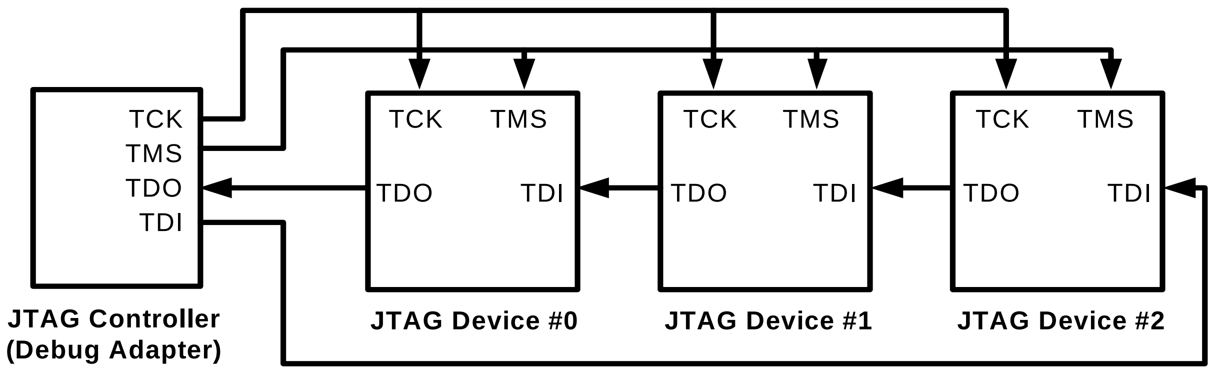


Figure 1. JTAG Chain Connection

10.1. SetJTAGDeviceAndConnect()

Description: This function is used to connect to a single target JTAG device in a JTAG chain using a Serial Adapter.

Supported Debug Adapters: Serial Adapter

C++ Prototype:

```
extern "C" __declspec(dllimport) HRESULT__stdcall SetJTAGDeviceAndConnect(
int nComPort=1, int nDisableDialogBoxes=0,
BYTE DevicesBeforeTarget=0, BYTE DevicesAfterTarget=0,
WORD IRBitsBeforeTarget=0, WORD IRBitsAfterTarget=0);
```

Parameters:

1. *nComPort*—Target COM port to establish a connection. The default is 1.
2. *nDisableDialogBoxes*—Disable (1) or enable (0) dialogs boxes within the DLL. The default is 0.
3. *DevicesBeforeTarget*—Number of devices in the JTAG chain before the target device. The default is 0.
4. *DevicesAfterTarget*—Number of devices in the JTAG chain after the target device. The default is 0.
5. *IRBitsBeforeTarget*—The sum of instruction register bits in the JTAG chain before the target device. The default is 0.
6. *IRBitsAfterTarget*—The sum of instruction register bits in the JTAG chain after the target device. The default is 0.

Following Figure 1, assuming all devices in the JTAG chain are Silicon Laboratories devices, call *SetJTAGDeviceAndConnect()* as shown in the following examples:

To access JTAG Device #0:

```
SetJTAGDeviceAndConnect(1, 0, 0, 2, 0, 32);
```

nComPort=1
nDisableDialogBoxes=0
DevicesBeforeTarget=0
DevicesAfterTarget=2
IRBitsBeforeTarget=0
IRBitsAfterTarget=32

To access JTAG Device #1:

```
SetJTAGDeviceAndConnect(1, 0, 1, 1, 16, 16);
```

nComPort=1
nDisableDialogBoxes=0
DevicesBeforeTarget=1
DevicesAfterTarget=1
IRBitsBeforeTarget=16
IRBitsAfterTarget=16

To access JTAG Device #2:

```
SetJTAGDeviceAndConnect(1, 0, 2, 0, 32, 0);
```

nComPort=1
nDisableDialogBoxes=0
DevicesBeforeTarget=2
DevicesAfterTarget=0
IRBitsBeforeTarget=32
IRBitsAfterTarget=0

Return Value: See Section 12 on page 16 for error return information.

10.2. SetJTAGDeviceAndConnectUSB()

Description: This function is used to connect to a single target JTAG device in a JTAG chain using a USB Debug Adapter.

Supported Debug Adapters: USB Debug Adapter

C++ Prototype:

```
extern "C" __declspec(dllimport) HRESULT__stdcall SetJTAGDeviceAndConnectUSB(
const char * sSerialNum, int nPowerTarget=0, int nDisableDialogBoxes=0,
BYTE DevicesBeforeTarget=0, BYTE DevicesAfterTarget=0,
WORD IRBitsBeforeTarget=0, WORD IRBitsAfterTarget=0);
```

Parameters:

1. *sSerialNumber*—The serial number of the USB Debug Adapter. See Section 8 for information on obtaining the serial number of each USB Debug Adapter connected. If only one USB Debug Adapter is connected, an empty string can be used. The default is an empty string.
2. *nPowerTarget*—If this parameter is set to 1, the USB Debug Adapter will be configured to continue supplying power after it has been disconnected from the target device. The default is 0, which configures the adapter to discontinue supplying power when disconnected.
3. *nDisableDialogBoxes*—Disable (1) or enable (0) dialogs boxes within the DLL. The default is 0.
4. DevicesBeforeTarget—Number of devices in the JTAG chain before the target device. The default is 0.
5. *DevicesAfterTarget*—Number of devices in the JTAG chain after the target device. The default is 0.
6. IRBitsBeforeTarget—The sum of instruction register bits in the JTAG chain before the target device. The default is 0.
7. *IRBitsAfterTarget*—The sum of instruction register bits in the JTAG chain after the target device. The default is 0.

Following Figure 1, assuming all devices in the JTAG chain are Silicon Laboratories devices, call *SetJTAGDeviceAndConnectUSB()* as shown in the following examples:

To access JTAG Device #0:

```
SetJTAGDeviceAndConnectUSB(““, 0, 0, 0, 2, 0, 32);
```

sSerialNumber=""DevicesBeforeTarget=0
nPowerTarget=0DevicesAfterTarget=2
nDisableDialogBoxes=0IRBitsBeforeTarget=0
IRBitsAfterTarget=32

To access JTAG Device #1:

```
SetJTAGDeviceAndConnectUSB(““, 0, 0, 1, 1, 16, 16);
```

sSerialNumber=""DevicesBeforeTarget=1
nPowerTarget=0DevicesAfterTarget=1
nDisableDialogBoxes=0IRBitsBeforeTarget=16
IRBitsAfterTarget=16

To access JTAG Device #2:

```
SetJTAGDeviceAndConnectUSB(““, 0, 0, 2, 0, 32, 0);
```

sSerialNumber=""DevicesBeforeTarget=2
nPowerTarget=0DevicesAfterTarget=0
nDisableDialogBoxes=0IRBitsBeforeTarget=32
IRBitsAfterTarget=32

Return Value: See Section 12 on page 16 for error return information.

11. Linking

Unless using explicit linking, it is necessary to provide the linker with the path of the “SiUtil.lib” library file before building the client executable. In Microsoft Visual C++, this is accomplished by selecting *Settings*… from the Project menu and then the *Link* tab. In the Object/library modules box, enter the full path and filename of the library file. For example, “c:\project\release\SiUtil.lib” is the path and filename you would use for the library file titled, “SiUtil.” The library file is not needed after the client executable is built.

If the DLL is implicitly linked, the DLL must be placed in one the following directories:

1. The directory containing the EXE client file.
2. The process’s current directory.
3. The Windows system directory.
4. The Windows directory.
5. A directory listed in the PATH environment variable.

12. Test Results

12.1. Error Code Definition

The DLL implements standard Windows HRESULT error codes. An HRESULT is a 32-bit value that includes a 16-bit error code in addition to information about the origin and severity of the error. The following information on the structure of an HRESULT code can be found in WINERROR.H:

```
//
//  HRESULTs are 32-bit values laid out as follows:
//
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +-+-+-+-+-+---------------------+-------------------------------+
//  |S|R|C|N|r|    Facility         |               Code            |
//  +-+-+-+-+-+---------------------+-------------------------------+
//
//  where
//
//      S - Severity - indicates success/fail
//
//          0 - Success
//          1 - Fail (COERROR)
//
//      R - reserved portion of the facility code, corresponds to
//          NT's second severity bit.
//
//      C - reserved portion of the facility code, corresponds to
//          NT's C field.
//
//      N - reserved portion of the facility code. Used to indicate
//          a mapped NT status value.
//
//      r - reserved portion of the facility code. Reserved for
//          internal use. Used to indicate HRESULT values that are
//          not status values, but are instead message ids for
//          display strings.
//
//      Facility - is the facility code
//
//      Code - is the facility's status code
//
```

The Windows HRESULT implementation also provides macros that should be used to test the function's success or failure:

```
//
// Generic test for success on any status value (non-negative
// numbers indicate success).
//

#define SUCCEEDED(Status) ((HRESULT)(Status) >= 0)

//
// and the inverse
//

#define FAILED(Status) ((HRESULT)(Status)<0)
```

Note that it is possible for functions to return non-zero positive "success" error codes. Therefore, you should always check for success or failure by using either the SUCCEEDED() or FAILED() macros.

Bit 15 of the Code field is the Target Disconnected Flag. If Bit 15 is set, then this indicates that the target device has been disconnected as a result of invoking the function. The following macro may be used to help test this bit:

```
#define TARGET_DISCONNECTED_FLAG 0x8000
```

When a non-zero HRESULT value is returned, the Facility field will indicate the source of the error.

It is possible to receive the following standard Windows errors (as defined in winerror.h):

E_INVALIDARG	*// One or more arguments is invalid*
E_HANDLE	*// Invalid handle*

Additional error return values for each Facility and Code value are defined in Tables 1, 2, 3, and 4.

Table 1. Facility Code 100 Error Description Table

Error Code	Error Definition	Error Code	Error Definition
0x101	Unable to write to Flash.	0x119	Target is not in the HALT state and did not respond to the query.
0x102	Unable to lock Flash.	0x11A	A Flash failure occurred.
0x103	Target device is not halted.	0x11B	Not connected to target. (This is a disconnect error, so the target disconnected flag bit will also be set.)
0x104	Invalid path.	0x11C	The target is not responding.
0x105	Unable to open COM Port.	0x11D	The target reported that the command failed.
0x106	Unable to download program.	0x11E	Communication port time out.
0x107	Reset failed.	0x11F	Could not clear communication port error.
0x108	Erase failed.	0x120	Error receiving byte.
0x109	Unable to close COM Port.	0x121	Receive Buffer Failure.
0x10A	USB Debug Adapter DLL not found.	0x122	Error occurred during purge of communication port.
0x10B	Unable to open USB Debug Adapter.	0x123	Error occurred during firmware update.
0x10C	External memory is not supported on this device.	0x124	A CRC comparison failure has occurred.
0x10D	Target device is connected.	0x125	An error occurred during device reset.
0x10E	Unknown device.	0x126	This operation is not supported on this device.
0x10F	Invalid target response.	0x127	Flash is locked.
0x110	Invalid arguments.	0x128	An incorrect response has been received.
0x111	Target did not respond and may be disconnected.	0x129	Valid adapter selection has not been assigned.
0x112	Target did not respond with enough data.	0x12A	The USB Debug Adapter handle is void.
0x113	The communication port is busy.	0x12B	USB Debug Adapter unknown function.
0x114	The communication port is not open.	0x12C	Flash erase has been cancelled.
0x115	Timer resource not available.	0x12D	Part not loaded in XML services.
0x116	Unable to write all the data to the target.	0x12E	The MCU product configuration data is missing information about this device.
0x117	Serial Interface had no new data to return.	0x12F	Unable to access MCU product configuration data.
0x118	UART has not been enabled via GPIO configuration.	0x130	The version of the device you connected to is older than the minimum version supported.

Table 2. Facility Code 102 Error Description Table

Error Code	Error Definition
0x101	Download cancelled.
0x102	Read failed.

Table 3. Facility Code 103 Error Description Table

Error Code	Error Definition
0x101	A valid adapter has not been defined.
0x102	Target could not be reset.
0x103	Unable to set up and open COM Port.
0x104	Unable to halt the target device.
0x105	Maximum number of breakpoints reached.
0x106	Target configuration failure.
0x107	Bootloader failure.

Table 4. Facility Code 104 Error Description Table

Error Code	Error Definition
0x101	Invalid handle.
0x102	Read error.
0x103	Write error.
0x105	Device I/O failed.
0x106	*Error string supplied by Windows*.
0x107	Read time out error.
0x108	SN access error.
0x109	Reset response failure.
0x10A	TLS invalid.
0x10B	Target not connected. (This is a disconnect error, so the target disconnected flag bit will also be set.)
0x10C	USBHID.DLL not initialized.
0x10D	No device selected.
0x1FF	Device not found.

The *GetErrorMsg()* function can be used to get a string describing the error so that an application can display a message with error information.

12.2. GetErrorMsg()

Description: This function returns a string describing an error which has been generated by the DLL.

Supported Debug Adapters: Serial Adapter, USB Debug Adapter

C++ Prototype:

```
extern "C" __declspec(dllexport)
char* __stdcall GetErrorMsg (HRESULT errorCode);
```

Parameters: **1.** *errorCode* - An HRESULT value for an error that should be interpreted.

Return Value: A string description of the errorCode.

13. Known Limitations

Upon invocation of the DLL in the debug mode of a client process, dialog messaging may not function properly. Dialog messaging provides a client with a way to get instant information that may not otherwise be available. Dialog messaging supports a progress indicator that provides a client with information on the progress of memory operations that may take time to complete. Calling into the DLL with a client (in debug version) may cause the DLL to misinterpret the correct window handle used to display the dialog boxes. The recommended course of action is to set all functions that have a *nDisableDialogBoxes* parameter to 1 before calling DLL functions. All *nDisableDialogBoxes* parameters default to 0. This problem will not occur in the release mode of a client process.

14. Visual Basic Information

When writing a Visual Basic client, it is important to note that the Interface Utilities DLL is written using Visual C++. Thus, you must take into account variable type differences between the two languages. Specifically, the VC++ boolean type and the VB boolean type are incompatible. In VC++, TRUE = 1 and FALSE = 0, whereas in VB TRUE = –1 and FALSE = 0. To resolve this issue, you must use an integer instead of a boolean when writing your VB client and send an integer value 1 for TRUE and integer value 0 for FALSE.

Document Change List

Revision 2.2 to Revision 2.3

- Instructions for declaring functions moved to Section 1.
- Reference to the SetJTAGDeviceAndConnect() function moved from Section 9 to Section 10.
- Replaced incorrect function descriptions in the Section 9 function list.

Revision 2.3 to Revision 2.4

- “Supported Debug Adapter” field added to each function definition.
- The following new functions added to support the USB Debug Adapter: ConnectUSB(), DisconnectUSB(), USBDebugDevices(), GetUSBDeviceSN(), GetHIDDLLVersion(), FLASHEraseUSB(), GetUSBFirmwareVersion(), SetJTAGDeviceAndConnectUSB()
- Section 4, “Program Interface Functions” split into 2 sections, creating Section 7, “Target Control Functions”.
- Section 8, “USB Debug Adapter Communication Functions” added.
- Added Return Codes 9 and 10 to Table 1, “Return Codes”.

Revision 2.4 to Revision 2.5

- Pointer format in prototypes corrected for consistency.
- `Std::string` types of variables changed to `const char * variables`.
- Section 8.3, “GetUSBDLLVersion()” function name changed from GetHIDDLLVersion to GetUSBDLLVersion.
- Section 10.2, “SetJTAGDeviceAndConnectUSB()” prototype corrected.

Revision 2.5 to Revision 2.6

- New parameter *nLockFlash* added to Download() function, Section 4.1.
- GetDLLVersion() function returns a "char*" value instead of an "int" value, Section 4.5.

Revision 2.6 to Revision 2.7

- Added description for new *GetDeviceName()* function as Section 4.6 on page 5.
- Moved function descriptions for *GetSAFirmwareVersion()*, *GetUSBFirmwareVersion()*, and *GetDLLVersion()* to Section 4, “Program Interface Functions”.

Revision 2.7 to Revision 2.8

- Updated Section 4.6 on page 5.
- Updated Section 8.2 on page 11.
  - Replaced “dllexport” with “dllimport”.

Revision 2.8 to Revision 2.9

- Error code return type changed from int_stdcall to HRESULT_stdcall.
- New parameter, bPersistFlash, added to Section "4.1. Download()" on page 4.
- Section “12. Test Results” replaced with new error code implementation information.

Revision 2.9 to Revision 3.0

- In Section "7.2. SetTargetHalt()" on page 10, return value type changed from BOOL to HRESULT with the function using the new error code implementation.

Revision 3.0 to Revision 3.1

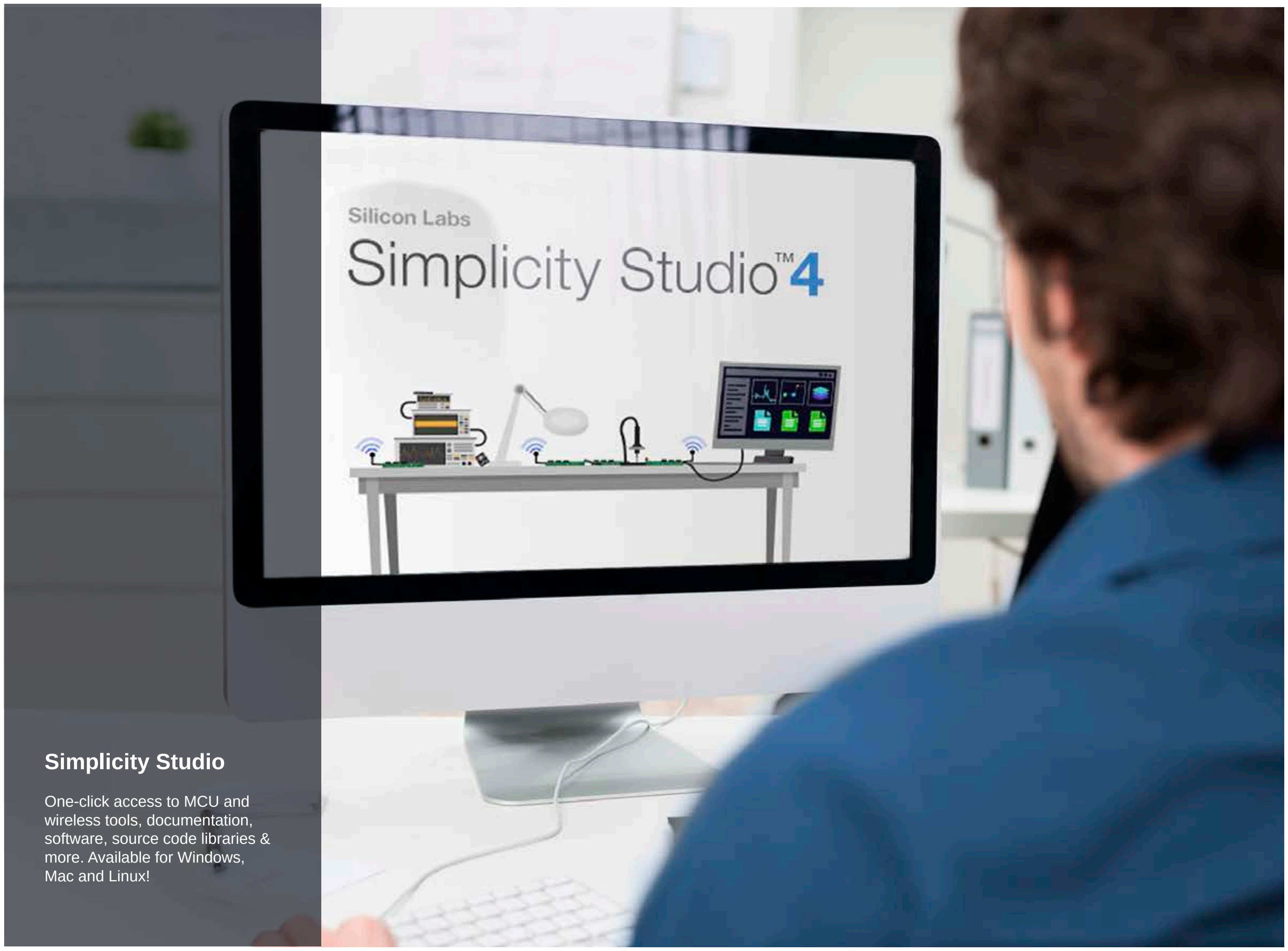
- Updated "12.1. Error Code Definition" on page 16.
- Updated Table 1, “Facility Code 100 Error Description Table,” on page 18.
- Updated Table 2, “Facility Code 102 Error Description Table,” on page 19.
- Updated Table 3, “Facility Code 103 Error Description Table,” on page 19.
- Updated Table 4, “Facility Code 104 Error Description Table,” on page 19.

Revision 3.1 to Revision 3.2

-

Revision 3.2 to Revision 3.3

- Updated document’s scope.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com