CYGNAL Application Note

# Using the C8051Fxxx On-Chip Interface Utilities DLL

## Relevant Devices

This application note applies to the following devices:
C8051F000, C8051F001, C8051F002, C8051F005, C8051F006, C8051F007, C8051F010, C8051F011, C8051F012, C8051F015, C8051F016, C8051F017, C8051F020, C8051F021, C8051F022, C8051F023, C8051F040, C8051F041, C8051F042, C8051F043, C8051F120, C8051F121, C8051F122, C8051F123, C8051F124, C8051F125, C8051F126, C8051F127, C8051F206, C8051F220, C8051F221, C8051F226, C8051F230, C8051F231, C8051F236, C8051F300, C8051F301, C8051F302, C8051F303, C8051F304, C8051F305, C8051F310, C8051F311

## 1.0. Introduction

The Interface Utilities DLL (Dynamic Link Library) provides functions to: download an Intel hex file to FLASH; connect and disconnect to a C8051Fxxx processor; "Run" and "Halt" the microprocessor; and read and write internal, external, and code memory spaces. All this functionality is provided via the PC's COM port and Cygnal Serial Adapter. The Connect function must be called first to establish a gateway to the C8051Fxxx target board.

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 stand alone application: it is a library of exported functions which are linked at run-time and called by a "Microsoft Windows™" application. If you are intending to write a client using Visual Basic please refer to section 7.0. for factors to take into account.

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 without having to write an application from scratch. The Interface Utilities application uses implicit linking, which requires the DLL to be placed in a specific directory (see Section 4.0.).

## 2.0. Files and Compatibility

The latest versions of the "CygUtil.dll" and "CygUtil.lib", are available at http://www.cygnal.com. The DLL is a Win32 MFC Regular DLL meaning it uses the Microsoft Foundation Class libraries; it can be loaded by any Win32 programming environment, and it only exports 'C' style functions. Two versions are available: one in which the MFC (Microsoft Foundations Classes) library is statically linked in the DLL, and another version in which the MFC library is dynamically linked in the DLL.


4301 Westbank Drive
Suite B-100
Austin, TX 78746
www.cygnal.com
*Copyright © 2002 Cygnal Integrated Products, Inc.*
*(All rights reserved)*

The statically linked MFC version includes a copy of all the MFC library code it needs and is thus self contained. No exterior MFC linking is required. With the MFC library code included, the statically linked DLL is approximately 220 KB.

The dynamically linked MFC version is approximately 92 KB. However, 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.

# 3.0. Calling the DLL’s exported functions from a client program

The DLL exports 14 functions. Their prototypes and relevant descriptions are listed below:

*int Connect(int nComPort=1, int nDisableDialogBoxes=0, int nECprotocol=0, int nBaudRateIndex=0);*
*BOOL Connected();*
*int Disconnect(int nComPort=1);*
*int Download(char* sDownloadFile, int nDeviceErase=0, int nDisableDialogBoxes=0, int nDownloadScratchPadSFLE = 0, int nBankSelect = -1);*
*int ISupportBanking(int * nSupportedBanks);*
*int SetTargetGo();*
*BOOL SetTargetHalt();*
*int GetRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int SetRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int GetXRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int SetXRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int GetCodeMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int SetCodeMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength, int nDisableDialogs=0);*
*int FLASHErase(int nComPort=1, int nDisableDialogBoxes=0, int nECprotocol=0);*
*int SetJTAGDeviceAndConnect(int nComPort, int nDisableDialogBoxes, BYTE DevicesBeforeTarget, BYTE DevicesAfterTarget, WORD BitsBeforeTargetIR, WORD BitsAfterTargetIR);*

## 3.1. Communications Functions

*int Connect(int nComPort=1, int nDisableDialogBoxes=0, int nECprotocol=0);*
*BOOL Connected();*
*int Disconnect(int nComPort=1);*

 © 2002 Cygnal Integrated Products, Inc.

### 3.1.1 Connect()

The Connect() function returns an integer whose value can be decoded as described in Section 5.0. The Connect() function accepts four default parameters of type int and bool. The nComPort input parameter represents the COM port to establish a connection with. The nDisableDialogs input parameter is a integer value indicating whether to disable (1) or enable (0) dialogs within the DLL. The nECProtocol input parameter is a integer value indicating which EC Protocol you are using. The default is ‘0’ which indicates that the JTAG debug interface is being used. ‘1’ indicates that the Cygnal 2-Wire debug interface is being used. The Cygnal 2-Wire debug interface is used with C8051F3XX derivative devices and the JTAG interface is used with all other C8051F derivatives. The nBaudRateIndex input parameter is an integer which selects what baud rate you wish to connect at. The default is ‘0’ which select Autobaud detection, ‘1’ selects 115200 baudrate, ‘2’ selects 57600 baudrate, ‘3’ selects 38400 baudrate, ‘4’ selects 9600 baudrate, and ‘5’ selects 2400 baudrate.

Please note that, establishing a valid connection is necessary for all memory operations to succeed.

When using C++, the Connect() function must be declared as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) from which the function will be called:

*extern "C" __declspec(dllexport) int __stdcall Connect(int nComPort=1, BOOL bDisableDialogBoxes=0, bool bECprotocol=0, int nBaudRateIndex=0);*

In the source file, call the function as shown in the following example:

*int r = Connect(nNewCOMPort, bDisableDialogs, bECprotocol, nBaudRateIndex);*

where nNewCOMPort, nDisableDialogs, and nECProtocol are variables which have been declared and initialized prior to the “Connect” function call.

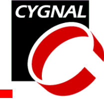
### 3.1.2 Connected()

The *Connected()* function returns a boolean (false ‘0’ equals not connected, and true ‘1’ equals connected) whose value represents the connection state of the target C8051xxx.

When using C++, you must declare the *Connected()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) from which the function will be called:

*extern "C" __declspec(dllexport) BOOL __stdcall Connected();*

In the source file, call the function as shown in the following example:




*BOOL r = Connected();*

### 3.1.3 Disconnect()

The *Disconnect()* function returns an integer whose value can be decoded as described in Section 5.0. The *Disconnect()* function accepts one default parameter of type *int*. The *nComPort* input parameter represents the COM port to terminate a connection with.

When using C++, declare the *Disconnect()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) from which the function will be called:

*extern "C" __declspec(dllexport) int __stdcall Disconnect(int nComPort=1);*

In the source file, call the function as shown in the following example:

*int r = Disconnect(nComPort);*

where *nComPort* is a variable which has already been declared and initialized prior to the “Disconnect” function call.

## 3.2. Program Interface Functions

*int Download(char* sDownloadFile, int nDeviceErase=0, int nDisableDialogBoxes=0, int nDownloadScratchPadSFLE = 0, int nBankSelect = -1);*
*int SetTargetGo();*
*BOOL SetTargetHalt();*
*int ISupportBanking(int * nSupportedBanks);*

### 3.2.1 Download()

The *Download()* function returns an integer whose value can be decoded as described in Section 5.0. The function accepts five parameters (four default parameters): *char* sDownloadFile, int nDeviceErase, int nDisableDialogs, int nDownloadScratchPadSFLE*, and *int nBankSelect*. The *sDownloadfile* input parameter must be a character pointer to the beginning of a character array (string) containing the full path and filename of the file to be downloaded. The n*DeviceErase* input parameter is a integer value that 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 *nDownloadScratchPadSFLE* input parameter is only for use with C8051F02X devices, C8051F04x devices and C8051F12x devices, otherwise this parameter should be left in it’s default state. If you are using a C8051F02X derivative, C8051F04x derivative, or C8051F12x derivative set this parameter to ‘1’ in order to download to ScratchPad memory. When accessing and downloading to ScratchPad memory please use the address range 0x0000 to 0x007F hexadecimal, writes out of this range will not be recognized. The *nBankSelect* input parameter is only for use with with C8051F12X devices, otherwise this parameter should be left in it’s default state.If you are using a C8051F12X derivative set this parameter to ‘1’, ‘2’, or

 © 2002 Cygnal Integrated Products, Inc.

'3' in order to download to a sepcific bank. Please note that 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.

When using C++, you must declare the *Download*() function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) from which the function will be called:

*extern "C" __declspec(dllexport) int __stdcall Download(char* sDownloadFile, int nDeviceErase=0, int nDisableDialogs=0, int nDownloadScratchPadSFLE = 0, int nBankSelect = -1);*

In the source file, call the function as shown in the following example:

*int r = Download(m_sDownloadFile, m_nDeviceErase, m_nDisableDialogs, nDownloadScratchPadSFLE, m_nBankNum);*

where *m_sDownloadFile*, *m_nDeviceErase, m_nDisableDialogs, nDownloadScratchPadSFLE, and m_nBankNum* are variables which have been declared and initialized prior to the "Download" function call.

### 3.2.2 SetTargetGo()

The SetTargetGo() function returns an integer whose value can be decoded as described in Section 5.0. Please note, after a successful exit from the SetTargetGo() function, the target C8051xxx will be in a "Run" state.

When using C++, declare the SetTargetGo() function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) int __stdcall SetTargetGo();*

In the source file, call the function as shown in the following example:

*int r = SetTargetGo();*

### 3.2.3 SetTargetHalt()

The SetTargetHalt() function returns a boolean (false equals target would not "Halt", true equals target is now in the "Halt" state) whose value represents the success of the target C8051xxx "Halt" command.

When using C++, declare the *SetTargetHalt()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) BOOL __stdcall SetTargetHalt();*

In the source file, call the function as shown in the following example:

*BOOL r = SetTargetHalt();*

### 3.2.4 ISupportBanking()

The *ISupportBanking()* function returns an integer whose value can be decoded as described in Section 5.0. The function accepts one parameter: *int * nSupportedBanks*. The *nSupportedBanks* output parameter must contain a valid integer address. The *ISupportBanking()* function expects to receive a pointer to an int (int *), or a reference to an int (&int) as the first parameter. The *ISupportBanking()* function will set *nSupportedBanks* to the number of banks supported on the target device, or 0 if none exist.

When using C++, you must declare the *ISupportBanking()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) from which the function will be called:

*extern "C" __declspec(dllexport) int __stdcall ISupportBanking(int * nSupportedBanks);*

In the source file, call the function as shown in the following example:

*int r = ISupportBanking(&m_nBankNum);*

where *m_nBankNum* is a variable which have been declared and initialized prior to the “ISupport-Banking” function call.

## 3.3. Get Memory Functions

*int GetRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int GetXRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int GetCodeMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*

The GetRAMMemory(), GetXRAMMemory(), and GetCodeMemory() functions belong to the *Get Memory* function section and will be discussed together. All *Get Memory* functions return integers, values can be decoded as described in Section 5.0. The *Get Memory* functions all accept a BYTE pointer that points to the beginning of a BYTE array as the first parameter. The *Get Memory* functions all expect to receive a pointer to a unsigned char (BYTE *) initialized array of nLength as the first parameter. If the *Get Memory* functions complete successfully, the *ptrMem* variable will contain the requested memory. An example is given below showing 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*


© 2002 Cygnal Integrated Products, Inc.

*// 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*

The GetMemory functions all expect to receive a DWORD as the second parameter, wStartAddress. The wStartAddress parameter should be interpreted as the beginning position in memory to reference. The GetMemory functions all expect to receive an unsigned integer as the third parameter, nLength. The nLength parameter should contain the number of bytes to read from memory.

### 3.3.1 GetRAMMemory()

The *GetRAMMemory()* 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.When using C++, declare the *GetRAMMemory()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) __stdcall int GetRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*

In the source file, call the function as shown in the following example:

*int r = GetRAMMemory(ptrBuf, m_wStartAt, m_nBytes);*

where *ptrBuf, m_wStartAt,* and *m_nBytes* are variables which have already been declared and initialized prior to the "GetRAMMemory" function call.

### 3.3.2 GetXRAMMemory()

The *GetXRAMMemory()* 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. Special attention should be paid to insure proper referencing of the External Data Address Space. When using C++, declare the *GetRAMMemory()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) int __stdcall GetXRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*

In the source file, call the function as shown in the following example:

*int r = GetXRAMMemory(ptrBuf, m_wStartAt, m_nBytes);*

where *ptrBuf, m_wStartAt,* and *m_nBytes* are variables which have already been declared and initialized prior to the “GetXRAMMemory” function call.

### 3.3.3 GetCodeMemory()

The *GetCodeMemory()* function will read the requested memory from the Program Memory Space. The requested Code memory must be located in the target device’s Program Memory Space. Special attention should be paid when reading from a sector that has been read locked. Reading from a sector that has been read locked will always return 0’s. Also note, reading of the reserved space is not allowed. Reading from the reserve space will always return an error. When using C++, declare the *GetCodeMemory()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) int __stdcall GetCodeMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*

In the source file, call the function as shown in the following example:

*int r = GetCodeMemory(ptrBuf, m_wStartAt, m_nBytes);*

where *ptrBuf, m_wStartAt,* and *m_nBytes* are variables which have already been declared and initialized prior to the “GetCodeMemory” function call.

## 3.4 Set Memory Functions

*int SetRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int SetXRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*
*int SetCodeMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength, int nDisableDialogs=0);*

The SetRAMMemory(), SetXRAMMemory, and SetCodeMemory functions belong to the SetMemory function section and will be discussed together. The SetMemory functions all return integer values that can be decoded in the section 5.0. The SetMemory functions expect to receive a pointer to a unsigned char (BYTE *) initialized 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.

An example is given below showing 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*

 © 2002 Cygnal Integrated Products, Inc.

Alternatively:

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

The *SetMemory* functions all expect to receive a DWORD as the second parameter, *wStartAddress*. The *wStartAddress* parameter should be interpreted as the beginning position in memory to reference. The *SetMemory* functions all expect to receive an unsigned integer as the third parameter, *nLength*. The *nLength* parameter should contain the number of bytes to write to memory.

## 3.4.1 SetRAMMemory()

The *SetRAMMemory()* 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.When using C++, declare the *SetRAMMemory()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) int __stdcall SetRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*

In the source file, call the function as shown in the following example:

*int r = SetRAMMemory(ptrBuf, m_wStartAt, m_nBytes);*

where ptrBuf, m_wStartAt, and m_nBytes are variables which have already been declared and initialized prior to the "SetRAMMemory" function call.

## 3.4.2 SetXRAMMemory()

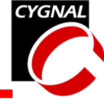
The *SetXRAMMemory()* function will write to memory in the External Data Address Space. The target XRAM memory must be located in the target device's External Data Address Space. Special attention should be paid to insure proper referencing of the External Data Address Space.When using C++, declare the *SetXRAMMemory()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) int __stdcall SetXRAMMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength);*

In the source file, call the function as shown in the following example:

*int r = SetXRAMMemory(ptrBuf, m_wStartAt, m_nBytes);*

where ptrBuf, m_wStartAt, and m_nBytes are variables which have already been declared and initialized prior to the "SetXRAMMemory" function call.



### 3.4.3 SetCodeMemory()

The SetCodeMemory() function will write to memory in the Program Memory Space. The SetCodeMemory() function includes an additional integer parameter, nDisableDialogs. This parameter determines whether to display dialogs within the DLL. The nDisableDialogs parameter defaults to 0. Please note, writing to FLASH will not complete successfully if a client tries to write to the reserved area of FLASH, write more than one page of data at one time, or write to a Write/Erase Locked sector. Also note, if writing to a page that contains the read/write/erase lock bytes, a device erase will be performed (a device erase will erase the entire contents of FLASH except the reserved area). 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.

When using C++, declare the SetCodeMemory() function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) int __stdcall SetCodeMemory(BYTE * ptrMem, DWORD wStartAddress, unsigned int nLength, int nDisableDialogs);*

In the source file, call the function as shown in the following example:

*int r = SetCodeMemory(ptrBuf, m_wStartAt, m_nBytes, m_nDisableDialogs);*

where *ptrBuf, m_wStartAt, m_nBytes, and m_nDisableDialogs* are variables which have already been declared and initialized prior to the “SetCodeMemory” function call.

## 3.5 Stand-alone Utilities

The following utilities are stand-alone, meaning that they do not require the use of the Connect or Disconnect functions and they do not interact with any other functions.

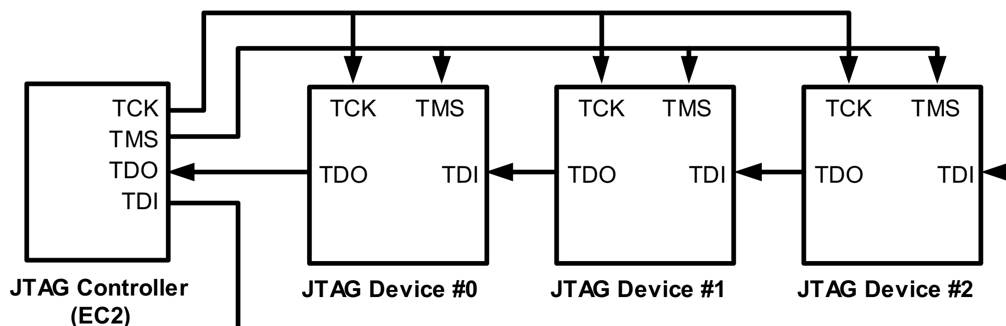



**Figure 1. JTAG Chain Connection**


© 2002 Cygnal Integrated Products, Inc.

### 3.5.1 FLASHErase()

The FLASHErase() function is used to erase the FLASH program memory of device whose FLASH read and/or write bytes have been written to. The nComPort input parameter represents the COM port to establish a connection with. The bDisableDialogs input parameter is a boolean value indicating whether to enable (TRUE) or disable (FALSE) dialogs within the DLL. The bECProtocol input parameter is a boolean value indicating which EC Protocol you are using. The default is FALSE or ‘0’ which indicates that the JTAG debug interface is being used. TRUE or ‘1’ indicates that the Cygnal 2-Wire debug interface is being used. The Cygnal 2-Wire debug interface is used with C8051F3XX derivative devices and the JTAG interface is used with all other C8051F derivatives. When using C++, declare the FLASHErase() function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

*extern "C" __declspec(dllexport) int __stdcall FLASHErase(int nComPort=1, bool bDisableDialogBoxes=0, bool nECprotocol=0);*

In the source file, call the function as shown in the following example:

*int r = FLASHErase(m_nComPort, m_bDisableDialogs, m_bECprotocol);*

where *m_nComPort, m_bDisableDialogs, and m_bECprotocol* are variables which have already been declared and initialized prior to the “SetCodeMemory” function call.

## 3.6 Multi-Device JTAG Programming

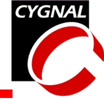
To program Cygnal 8051 devices in a JTAG chain you must configure your JTAG chain as shown in Figure 1. The JTAG Controller must be an EC2 Serial Adapter. If you have an EC1 device you will need to upgrade to an EC2. Also, you will need to know the instruction register length of each device in the JTAG chain. The instruction register length of all Cygnal JTAG devices is 16 bits.

### 3.6.1 SetJTAGDeviceAndConnect()

The *SetJTAGDeviceAndConnect()* function is used to isolate a single target JTAG device in the JTAG chain and connect to it. The function accepts six parameters (six default parameters): *nComPort, int nDisableDialogBoxes, BYTE DevicesBeforeTarget, BYTE DevicesAfterTarget, WORD IRBitsBeforeTarget, WORD IRBitsAfterTarget*.

The *nComPort* input parameter represents the PC COM port we will be using to communicate.

The *nDisableDialogs* input parameter is a integer value indicating whether to disable (1) or enable (0) dialogs within the DLL.



The *DevicesBeforeTarget* input parameter represents the number of devices in the JTAG chain before the target device.

The *DevicesAfterTarget* input parameter represents the number of devices in the JTAG chain after the target device.

For the last two parameters you must know the instruction register length of each device before and after the target device. Remember again that the instruction register length for each Cygnal JTAG device in the JTAG chain is 16 bits. So, the *IRBitsBeforeTarget* is the sum of instruction register bits in the JTAG chain before the target device, and the *IRBitsAfterTarget* is the sum of instruction register bits in the JTAG chain after the target device.

Once you have successfully called this function and connected to a Cygnal device in the JTAG chain you may then use any of the Program Interface Functions on the isolated device. When finished interfacing with the device simply call the *Disconnect*() function as usual to disconnect from the device. When using C++, declare the *SetJTAGDeviceAndConnect()* function as an imported function. Add the following line to the header file (*.h) of the source file (*.cpp) calling the function:

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

In the source file, call the function as shown in the following example:

*inr r = SetJTAGDeviceAndConnect(m_nComPort, m_nDisableDialogs, m_BDevicesBeforeTarget, m_BDevicesAfterTarget, m_WIRBitsBeforeTarget, m_WIRBitsAfterTarget);*

where *m_nComPort, m_nDisableDialogs, m_BDevicesBeforeTarget, m_BDevicesAfterTarget, m_WIRBitsBeforeTarget,* and *m_WIRBitsAfterTarget* are variables which have already been declared and initialized prior to the "*SetJTAGDeviceAndConnect*" function call. Following Figure 1, assuming all devices in the JTAG chain are Cygnal devices, to access JTAG Device #1 on COM 1, the variables would be set as follows:

*m_nComPort* = 1
*m_nDisableDialogs* = 0
*m_BDevicesBeforeTarget* = 0
*m_BDevicesAfterTarget* = 2
*m_WIRBitsBeforeTarget* = 0
*m_WIRBitsAfterTarget* = 32

To access JTAG Device #2:

*m_nComPort* = 1

 © 2002 Cygnal Integrated Products, Inc.

*m_nDisableDialogs* = 0
*m_BDevicesBeforeTarget* = 1
*m_BDevicesAfterTarget* = 1
*m_WIRBitsBeforeTarget* = 16
*m_WIRBitsAfterTarget* = 16

To access JTAG Device #3:

*m_nComPort* = 1
*m_nDisableDialogs* = 0
*m_BDevicesBeforeTarget* = 2
*m_BDevicesAfterTarget* = 0
*m_WIRBitsBeforeTarget* = 32
*m_WIRBitsAfterTarget* = 0

# 4.0. Linking

Unless using explicit linking, it is necessary to provide the linker with the path of the "CygUtil.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 next. For example, "c:\project\release\CygUtil.lib". 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.

# 5.0. Test Results

On exit, the DLL will return an integer value return code. If a fatal error occurs during the DLL's execution, the DLL will also display a message box, if display dialogs are enabled, stating the error and then exit. The return codes are:

**Return Codes**

| Return Code | Error | Status | Possible Causes |
|---|---|---|---|
| -3 | Flash Write Error | Failed | Invalid page write, writing to the reserved area of FLASH, etc.... |
| -2 | Target State Failure | Failed | Target not in Halt State |
| -1 | Target State Failure | Failed | Target not Con-nected |
| 0 | No error | Success | Function call com-pleted Successfully |
| 1 | File Name or path Error | Failed | Invalid path and/or file doesn't exist |
| 2 | COM Port Error | Failed | Can not establish a connection with the selected COM port, |
| 3 | Download Sequence Error | Failed | Invalid number of bytes, requested memory operation does not exist. |
| 4 | Reset Sequence Error | Failed | The target device failed to execute a reset sequence; ver-ify that a valid con-nection still exist |
| 5 | Device Erase Error | Failed | The target device failed to execute an Erase sequence; ver-ify Write/Erase Lock Bytes; verify that a valid connection still exists |

 © 2002 Cygnal Integrated Products, Inc.

**Return Codes**

| Return Code | Error | Status | Possible Causes |
|---|---|---|---|
| 7 | Error Closing COM Port | Failed | Could not establish a connection with the target to Close the COM port; verify that a connection exists |
| 8 | Invalid Parameter | Failed | Invalid parameter[s] passed into the DLL |

# 6.0. 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. Recommended course of action is to set all functions that have a *nDisableDialogs* parameter to '1' before calling into the DLL. All *nDisableDialogs* parameters default to '0'. This problem will not occur in the release mode of a client process.

# 7.0. 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 the 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.