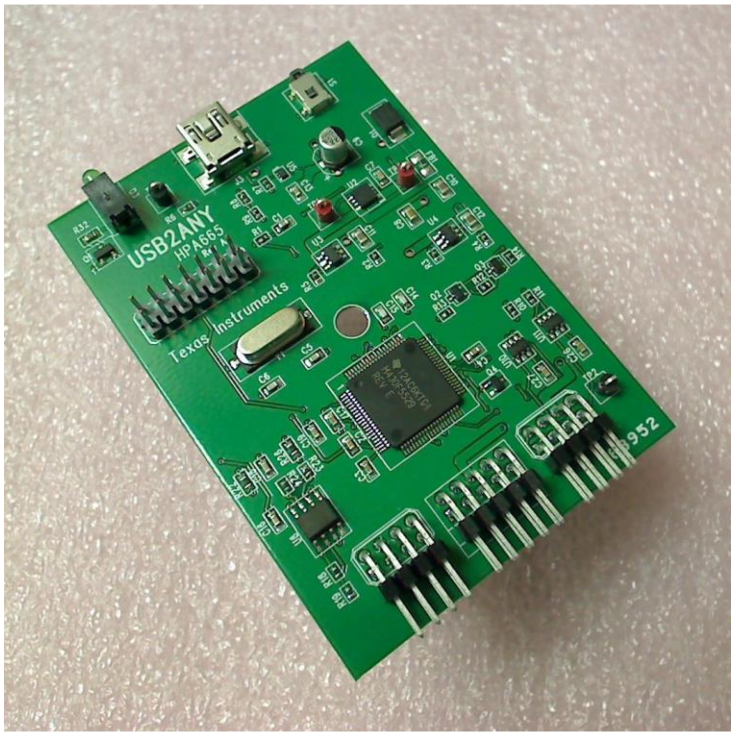
TEXAS INSTRUMENTS

API Reference for the USB2ANY Software Developer's Kit

Version V2.8.2



Author: Randy Turchik
May 12, 2017

Contents

Preface

Read This First

About This Manual

This user's guide describes the functions and operation of the USB2ANY Interface Adapter, from different aspects of hardware design, firmware programming, communication protocols, GUI and PC libraries, etc.

How to Use This Manual

This document contains the following chapters:
· Preface
· Chapter 1 – Introduction
· Chapter 2 – Overview of Hardware Design
· Chapter 3 – Programming and Communications Protocol
· Chapter 4 – API Summary
· Chapter 5 – API Function Reference
· Appendices

Information about Cautions and Warnings

FCC Warning

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case, the user, at his/her own expense, will be required to take whatever measures may be required to correct this interference.

1 Introduction

This document describes the API library (USB2ANY.DLL), which provides a set of functions to simplify the interfacing and operation of the USB2ANY controller board.

The USB2ANY Controller Board is a small dongle that, via a USB connection to a computer, enables access to devices using the following interfaces:

- I^2C
- SPI
- ADC
- DAC
- PWM
- UART
- GPIO
- Shared memory
- FEC
- Interrupts
- EasyScale™

The USB2ANY board can source regulated 3.3v and 5.0v DC power to the target device, up to a maximum of 300ma (combined 3.3v and 5.0v).

The API library is intended for use on computers running the Windows operating system (XP through Windows 7) and can be accessed by any language (e.g., Visual C++, Visual Basic, Python, etc.) that is capable of interfacing with a DLL (dynamic-linked library). The DLL can be linked manually, using Windows API functions, or automatically at run-time. An object library file (USB2ANY.LIB) is provided to simplify interfacing.

2 Overview of Hardware Design

See ***Appendix D:*** on page 99 for drawings.

TBD

3 Programming and Communications Protocol

See ***Appendix B: Exported Symbols*** on page 94 for the names required for linkage to the API functions.

See ***Appendix C: Visual Basic Interface (VB6)*** on page 96 for VB6 declarations of USB2ANY.DLL subroutines and functions.

TBD

4 API Summary

This section provides a quick summary of the API functions available in the USB2ANY.DLL library. The list is organized by function type, in order to expedite finding the desired function. The functions are listed by function name a brief one-line description for each function.

This list is intended only as a Quick Reference and assumes that the reader has experience and/or knowledge of the API. See section 5 for detailed descriptions of the functions.

Controller Functions	
u2aFindControllers	Scans the USB bus, enumerating USB2ANY devices and creating a list of the devices it finds. The list can be read using the u2aGetSerialNumber function.
u2aGetSerialNumber	Returns the serial number of one of the USB2ANY devices found by a previous call to the u2aFindControllers function.
u2aOpen	Opens communication with the USB2ANY controller that has the specified serial number.
u2aClose	Closes communication with the USB2ANY controller associated with the specified handle.
u2aReadResponse	Reads the response from any of the other functions that return data.
I^{2}C Functions	
I2C_Control	Sets the communications parameters (speed, address length, and pullup state) for the I^{2}C interface.
u2aI2C_RegisterRead	Reads a single byte from a register of a device on the I^{2}C bus.
u2aI2C_RegisterWrite	Writes a single byte to a register of a device on the I^{2}C bus.
u2aI2C_MultiRegisterRead	Reads data from multiple registers of a device on the I^{2}C bus.
u2aI2C_MultiRegisterWrite	Writes data to multiple registers of a device on the I^{2}C bus.
u2aI2C_InternalRead	Reads data from a device on the I^{2}C bus, using an internal address.
u2aI2C_InternalWrite	Writes data to an internal address of a device on the I^{2}C bus.
u2aI2C_RawRead	Reads raw data from a device on the I^{2}C bus.
u2aI2C_RawWrite	Writes raw data to a device on the I^{2}C bus.
SPI Functions	
u2aSPI_Control	Sets the parameters for SPI transactions.
u2aSPI_WriteAndRead	Simultaneously writes and reads data from an SPI device.
ADC, DAC, and PWM Functions	
u2aADC_Control	Sets the parameters for the ADC pins.
u2aADC_ConvertAndRead	Triggers an ADC conversion and reads the results.
u2aDACs_Write	Writes data to the specified DAC.
u2aPWM_Control	Sets the parameters for the PWM pins.

UART Functions	
u2aUART_Control	Sets the USART parameters for serial communication.
u2aUART_Write	Writes serial data via the USART.
GPIO Functions	
u2aGPIO_WriteControl	Writes control data to all of the GPIO pins, simultaneously.
u2aGPIO_WriteState	Sets the output state of all GPIO pins, simultaneously.
u2aGPIO_ReadState	Reads the input state of all of the GPIO pins, simultaneously.
u2aGPIO_SetPort	Configures a single GPIO pin as an output or input (with resistor options).
u2aGPIO_WritePort	Sets the state of a single GPIO output pin.
u2aGPIO_ReadPort	Reads the state of a single GPIO input pin.
Memory Functions	
u2aMSP430_ByteRead	Reads a single byte from the MSP430 memory.
u2aMSP430_ByteWrite	Writes a single byte to the MSP430 memory.
u2aMSP430_WordRead	Reads a single word from the MSP430 memory.
u2aMSP430_WordWrite	Writes a single word to the MSP430 memory.
Miscellaneous Functions	
u2aFirmwareVersion_Read	Reads the firmware version number of the USB2ANY controller.
u2aPower_WriteControl	Enables/disables the 3.3V and 5.0V power outputs.
u2aPower_ReadStatus	Reads the status of the 3.3V and 5.0V power outputs.
u2aLED_WriteControl	Used to control the LED.
u2aClock_Control	Sets the clock divider.

5 API Function Reference

5.1 Controller Functions

5.1.1 u2aFindControllers

int **u2aFindControllers**(void)

int **CU2AClass:: FindControllers**()

This function scans the USB bus, enumerating USB2ANY devices and creating a list of the devices it finds. The list can be read using the **u2aGetSerialNumber** function.

Parameters:

None

Return:

Returns the number of USB2ANY devices found or zero, if no devices were found.

5.1.2 u2aGetSerialNumber

int **u2aGetSerialNumber**(int ***index***, char ****SerialNumber***)

int **CU2AClass:: GetSerialNumber**(int ***index***, char ****SerialNumber***)

This function returns the serial number of one of the USB2ANY devices found by a previous call to the **u2aFindControllers** function.

Parameters:

index	An index into the list of USB2ANY devices that was returned by the **u2aFindControllers** function. This must be a number in the range of zero to n-1 where n is the number returned by the **u2aFindControllers** function. To get the serial number of the first device, set index to zero. To enumerate remaining devices, set index to -1. This may also be set to any valid index to get the serial number associated with that index. An error is returned if index is out of range.

SerialNumber	A char buffer to receive the selected serial number string. The buffer should have room for at least 40 characters (defined as **SERNUM_LEN** in USB2ANY_SDK.H).

Return:

Returns zero on success. If ***index*** is out of range, or there are no more devices to enumerate, **ERR_PARAM_OUT_OF_RANGE** is returned.

5.1.3 u2aOpen

HANDLE **u2aOpen** (char ****SerialNumber***)

HANDLE **CU2AClass:: Open** (char ****SerialNumber***)

This function opens communication with the USB2ANY controller that has the specified serial number.

Parameters:

SerialNumber	A serial number string that was returned by a call to the **u2aGetSerialNumber** function.

Comments:

This function supports an "Easy Open" method that may be utilized by simple applications that use only one USB2ANY device connected to the computer. To use this method, the ***SerialNumber*** parameter is set to an empty string ("") or NULL (0). This will cause the **u2aOpen** function to look for the first available USB2ANY, get its serial number, and then attempt to open it.

Return:

Returns a handle, which must be used for subsequent calls to API functions. The handle is always a positive number (never zero). If an error occurs, a negative error code is returned.

5.1.4 u2aClose

int **u2aClose** (U2A_HANDLE ***handle***)

int **CU2AClass:: Close** ()

This function closes communication with the USB2ANY controller associated with the specified handle.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.1.5 u2aReadResponse (Deprecated)

int **u2aReadResponse**(U2A_HANDLE ***handle***, BYTE ****pBuffer***, DWORD ***dwBufferSize***)

This function reads the response from any of the other functions that return data.

Warning: This function is deprecated. The function is not necessary and performs no useful function in the USB2ANY API V2.6.0.0 and later.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
pBuffer	Pointer to a buffer, which will receive the data. Care must be taken to ensure that the buffer is large enough to hold the expected data. To be safe, you can make the size of the buffer **MAXIMUM_DATA_SIZE** (defined in USB2ANY_SDK.H) bytes.
dwBufferSize	The size of the buffer (in bytes), pointed to by ***pBuffer***.

Return:

Always returns* ERR_UNIMPLEMENTED_FUNCTION *when called in API V2.6.0.0, or later.

5.1.6 u2aSetReceiveTimeout

int **u2aSetReceiveTimeout** (int ***milliseconds***)

int **CU2AClass:: SetReceiveTimeout** (int ***milliseconds***)

This function sets the timeout value, in milliseconds, used during USB receive operations.

Parameters:

milliseconds	Timeout value, in milliseconds (minimum 2, maximum 200)

Comments:

The default receive timeout is 25 milliseconds. This value should be adequate for virtually all of the API functions. However, the values of certain parameters used when calling some API functions may require longer timeouts. For example, using slower baud rates (e.g. 300 baud) with the UART functions may cause timeouts to occur when sending more than a few bytes of data.

You can temporarily increase the timeout while calling a single API function (or a group of functions) as follows:

```
old_value = u2aSetReceiveTimeout(new_value);    // set new value, saving the old value
u2aUART_Write(handle, 10, data);                // call the API function
u2aSetReceiveTimeout(old_value);                // restore the saved timeout value
```

Note that leaving the receive timeout set to larger values may adversely impact the overall performance of other API functions.

Return:

Returns the previous timeout value on success. If an error occurs, the timeout value is *not* changed and one of the following error codes is returned:

- ERR_PARAM_OUT_OF_RANGE
- ERR_COM_PORT_NOT_OPEN
- ERR_OPERATION_FAILED

5.1.7 u2aEnableDeviceDetect

BOOL **u2aEnableDeviceDetect** (U2A_HANDLE ***handle***, BOOL ***enable***, void ****Callback***)

BOOL **CU2AClass:: EnableDeviceDetect** (BOOL ***enable***, void ****Callback***)

This function enables/disables the detection and automatic reporting of the controller device (USB2ANY or OneDemo) plug and unplug events.

Parameters:

handle	Valid handle (obtained from **U2A_Open**) for the USB2ANY device.

enable	Determines whether to detect and report changes of the device connection status.
	Disable = 0
	Enable = 1

Callback	The address of the callback function to be called when a controller board is connected or disconnected. If set to NULL, no function is called. Note that board connection/disconnection status can be determined "manually" by calling the **u2aStatus_IsUSB2ANYConnected**() function, regardless of whether the **u2aEnableDeviceDetect**() function has been called.

Comments:

If ***Callback*** is set to a non-Null value, the function at the specified address will be called whenever the controller board is connected or disconnected. The callback function must have the following prototype:

```
void __stdcall Callback(BOOL bConnected, const char *szSerialNumber)
```

There will be two parameters passed to the function. The first parameter is a Boolean value that denotes the connection status and will be zero (0) if a board was disconnected or one (1) if a board was connected. The second parameter is a pointer to char string containing the serial number of the board that was connected or disconnected.

If the use of a callback function is not desired, this function should not be used. Instead, use the **u2aStatus_IsUSB2ANYConnected**() function.

Return:

Returns a Boolean value – TRUE on success, FALSE on failure.

5.1.8 u2aEnableDebugLogging

int **u2aEnableDebugLogging** (BOOL ***enable***)

int **CU2AClass:: EnableDebugLogging** (BOOL ***enable***)

This function enables/disables the logging of debug messages for every API function that is called. The debug log is intended to help with development and debugging of USB2ANY and OneDemo applications that use the API functions provided by the SDK.

Parameters:

enable	Determines whether to log information for each function called.
	Disable = 0
	Enable = 1

Comments:

Logging of debug information is enabled when ***enable*** is set to a TRUE (i.e., non-zero) value. The log file shows the full path and version number of the currently loaded USB2ANY.DLL library. The debug information logged for each function call consists of a time/date stamp, the function name, any parameters passed to the function (with values), and the returned value. Additional information may also be logged.

The debug log files are typically located in the **USB2ANY\Logs** sub-folder of the Windows **Documents** folder (e.g., **C:\Users\YourUserName\Documents\USB2ANY\Logs**). The name of the debug log file will be **<AppName> API Debug.log**, where **<AppName>** is the name of the application program that loaded USB2ANY.DLL. For example, running the USB2ANY Explorer.exe program will create a debug log file named **USB2ANY Explorer API Debug.log**. The contents of the debug log file will look something like this:

```
2013-08-26 12:04:51.692 ######################## SESSION START   ###########################
2013-08-26 12:04:51.692 Loading C:\Windows\SysWOW64\USB2ANY.dll (v2.6.1.1)
2013-08-26 12:04:51.896
2013-08-26 12:04:51.897 u2aEnableAPIProfiling(enable=1) [3279, TI_USB2ANY.cpp]
2013-08-26 12:04:51.899   Returning value 0 [3285, TI_USB2ANY.cpp]
2013-08-26 12:04:51.922
2013-08-26 12:04:51.925 u2aEnableDeviceDetect(enable=1, pfnCallback=0x002E6680) [480, TI_USB2ANY.cpp]
2013-08-26 12:05:00.832
2013-08-26 12:05:00.834 u2aFindControllers() [813, TI_USB2ANY.cpp]
2013-08-26 12:05:00.917   Found controller S/N AF14904609001C00 [799, TI_USB2ANY.cpp]
2013-08-26 12:05:00.919   Returning value 1 [808, TI_USB2ANY.cpp]
2013-08-26 12:05:00.920   Elapsed time:   89.77163 ms.
2013-08-26 12:05:00.921   Returning value 1 [819, TI_USB2ANY.cpp]
2013-08-26 12:05:00.923   Return code set to 1 [493, TI_USB2ANY.cpp]
2013-08-26 12:05:00.925   Elapsed time:   94.56611 ms.
2013-08-26 12:05:00.926   Returning value 1 [498, TI_USB2ANY.cpp]
```

Return:

The **u2aEnableDebugLogging** function returns the previous enable state of debug logging (TRUE if it was enabled before executing this function, otherwise FALSE). If an error occurs, a negative error code is returned.

5.1.9 u2aSuppressSplash

BOOL **u2aSuppressSplash** (BOOL ***suppress***)

BOOL **CU2AClass:: SuppressSplash** (BOOL ***suppress***)

This function enables/disables the display of the splash screen, which occurs the first time that the **u2aOpen** function is called.

Parameters:

suppress	Determines whether the splash screen is displayed. Set TRUE to suppress the splash screen or FALSE to allow it to be displayed.

Comments:

The splash screen is displayed only once per session, when the **u2aOpen** function is called. In order for this function to have an effect, it must be called ***before*** the first call to **u2aOpen.**

Return:

The **u2aSuppressSplash** function returns the previous splash suppression flag (TRUE if suppression was enabled before executing this function, otherwise FALSE). If an error occurs, a negative error code is returned.

5.1.10 u2aSetAsyncIOCallback

int **u2aSetAsyncIOCallback** (U2A_HANDLE ***handle***, void ****Callback,*** int ***nFunctionID***)

int **CU2AClass:: SetAsyncIOCallback** (void ****Callback,*** int ***nFunctionID***)

This function sets the address of the user function to be called when the specified function's I/O completes.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Callback	The address of the callback function to be called when the I/O operation completes. If set to NULL, no function is called. See **Comments** section for more information.

nFunctionID	The ID of the function for which the callback will be set. Valid values are:
	Cmd_I2C_ReadInternal = 101 (0x65)
	Cmd_I2C_WriteInternal = 102 (0x66)

Comments:

The **u2aSetAsyncIOCallback** function allows certain I/O functions to have their results returned asynchronously. Identifiers for the API functions that allow asynchronous operation are shown in the description of the ***nFunctionID*** parameter. Note that the callback will remain in effect until cancelled, which is done by calling this function with the Callback parameter set to NULL.

When ***Callback*** is set to a non-null value, a function at the specified address will be called when I/O for the specified function completes. The callback function must have the following prototype:

```
void _stdcall CallBack(int result, BYTE *data)
```

There will be two parameters passed to the callback function. The first parameter, `result`, will be set to either the number of bytes transferred (zero or positive) or an error code (always negative).

The second parameter, `data`, is a pointer to the returned data on a successful read operation. The `data` parameter is always set to NULL on write operations, as there is no data returned.

To disable the callback, call this function with the ***Callback*** parameter set to NULL. In this case, the ***nFunctionID*** parameter must also be set to the appropriate function ID.

Return:

u2aSetAsyncIOCallback returns zero on success, or a negative error code if an error occurs.

5.2 I2C Functions

5.2.1 u2aI2C_Control

int **u2aI2C_Control** (U2A_HANDLE ***handle***, I2C_Speed ***Speed***,
I2C_AddressLength ***AddressLength***, I2C_PullUps ***PullUps***)

int **CU2AClass:: I2C_Control** (I2C_Speed ***Speed***, I2C_AddressLength ***AddressLength***,
I2C_PullUps ***PullUps***)

This function sets the communications parameters (speed, address length, and pullup state) for the I^{2}C interface.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Speed	The bitrate for I^{2}C communications. Valid values are:
	I2C_100kHz = 0
	I2C_400kHz = 1
	I2C_10kHz = 2

AddressLength	Size of the I^{2}C slave device address. May be 7 or 10 bits. Valid values are:
	I2C_7Bits = 0
	I2C_10Bits = 1

PullUps	Sets the state of the I^{2}C pullups. May be 0 (off) or 1 (on). Valid values are:
	I2C_PullUps_OFF = 0
	I2C_PullUps_ON = 1

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.2.2 u2aI2C_RegisterRead

int **u2aI2C_RegisterRead** (U2A_HANDLE ***handle***, UInt16 ***I2C_Address***,
Byte ***RegisterAddress***)

int **CU2AClass:: I2C_RegisterRead** (UInt16 ***I2C_Address***, Byte ***RegisterAddress***)

This function reads a single byte from a register of a device on the I^{2}C bus.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Address	The address of the I^{2}C device. May be 7 or 10 bits. See **u2aI2C_Control**.
RegisterAddress	Address of the register in the I^{2}C slave device. Must be a single byte value.

Return:

Returns the value of the byte read on success. If an error occurs, a negative error code is returned.

5.2.3 u2aI2C_RegisterWrite

int **u2aI2C_RegisterWrite**(U2A_HANDLE ***handle***, UInt16 ***I2C_Address***,
Byte ***RegisterAddress***, Byte ***Value***)

int **CU2AClass:: I2C_RegisterWrite**(UInt16 ***I2C_Address***, Byte ***RegisterAddress***,
Byte ***Value***)

This function writes a single byte to a register of a device on the I^{2}C bus.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Address	The address of the I^{2}C device. May be 7 or 10 bits. See **u2aI2C_Control**.
RegisterAddress	Address of the register in the I^{2}C slave device. Must be a single byte value.
Value	The single-byte value to be written to the specified register.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.2.4 u2aI2C_MultiRegisterRead

int **u2aI2C_MultiRegisterRead** (U2A_HANDLE ***handle***, UInt16 ***I2C_Address***,
Byte ***StartingRegisterAddress***, Byte ***nBytes***,
Byte *****Data***)

int ***CU2AClass::*** **I2C_MultiRegisterRead** (UInt16 ***I2C_Address***,
Byte ***StartingRegisterAddress***, Byte ***nBytes***,
Byte *****Data***)

This function reads data from multiple registers of a device on the I^{2}C bus.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Address	The address of the I^2C device. May be 7 or 10 bits. See **u2aI2C_Control**.
StartingRegisterAddress	Address of the first register to read. Must be a single byte value.
nBytes	The number of bytes (i.e., registers) to be read.
Data	Pointer to array of bytes to receive the data read.

Return:

Returns number of bytes read on success. If an error occurs, a negative error code is returned.

5.2.5 u2aI2C_MultiRegisterWrite

int **u2aI2C_MultiRegisterWrite** (U2A_HANDLE ***handle***, UInt16 ***I2C_Address***, Byte ***StartingRegisterAddress***, Byte ***nBytes***, Byte *****Data***)

int ***CU2AClass::*****I2C_MultiRegisterWrite** (UInt16 ***I2C_Address***, Byte ***StartingRegisterAddress***, Byte ***nBytes***, Byte *****Data***)

This function writes data to multiple registers of a device on the I^2C bus.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Address	The address of the I^2C device. May be 7 or 10 bits. See **u2aI2C_Control**.
StartingRegisterAddress	Address of the first register to write. Must be a single byte value.
nBytes	The number of bytes to be written.
Data	Pointer to array of bytes to be written.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.2.6 u2aI2C_InternalRead

int **u2aI2C_InternalRead** (U2A_HANDLE ***nHandle***, UInt16 I2C_***Slave_Address***, UInt16 ***InternalAddress***, Byte ***IntAddrSize***, UInt16 ***nBytes***, Byte *****Data***)

int ***CU2AClass::*****I2C_InternalRead** (UInt16 I2C_***Slave_Address***, UInt16 ***InternalAddress***, Byte ***IntAddrSize***, UInt16 ***nBytes***, Byte *****Data***)

This function reads data from a device on the I^{2}C bus, using an internal address.

Parameters:

nHandle	Valid handle (obtained from **u2aOpen**) for the USB2ANY device.
I2C_Slave_Address	The address of the I^{2}C device. May be 7 or 10 bits. See **u2aI2C_Control**.
InternalAddress	Internal address of the data to read.
IntAddrSize	Number of internal address bytes. Must be 0, 1, or 2.
nBytes	The number of bytes to be read.
Data	Pointer to array of bytes to receive the data read.

Return:

If a positive number, the number of bytes actually read. On failure, an error code is returned (always negative).

5.2.7 u2aI2C_InternalWrite

int **u2aI2C_InternalWrite** (U2A_HANDLE nHandle, UInt16 I2C_***Slave_Address***,
UInt16 ***InternalAddress***, Byte ***IntAddrSize***, UInt16 ***nBytes***,
Byte ****Data***)

int ***CU2AClass::*****I2C_InternalWrite** (UInt16 I2C_***Slave_Address***,
UInt16 ***InternalAddress***, Byte ***IntAddrSize***, UInt16 ***nBytes***,
Byte ****Data***)

This function writes data to an internal address of a device on the I^{2}C bus.

Parameters:

nHandle	Valid handle (obtained from **u2aOpen**) for the USB2ANY device.
I2C_Slave_Address	The address of the I^{2}C device. May be 7 or 10 bits. See **u2aI2C_Control**.
InternalAddress	Internal target address for the data to be written.
IntAddrSize	Number of internal address bytes. Must be 0, 1, or 2.
nBytes	The number of bytes to be written.
Data	Pointer to array of bytes to be written.

Return:

If a positive number, the number of bytes actually written. On failure, an error code is returned (always negative).

5.2.8 u2aI2C_RawRead

int **u2aI2C_RawRead** (U2A_HANDLE ***handle***, UInt16 ***I2C_Address***, Byte ***nBytes***, Byte ****Data***)

int ***CU2AClass::***I2C_RawRead (UInt16 ***I2C_Address***, Byte ***nBytes***, Byte ****Data***)

This function reads raw data from a device on the I^2C bus.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Address	The address of the I2C device. May be 7 or 10 bits. See **u2aI2C_Control**.
nBytes	The number of bytes to be read.
Data	Pointer to a buffer to receive the data read.

Return:

On success, returns number of bytes read. If an error occurs, a negative error code is returned.

5.2.9 u2aI2C_RawWrite

int **u2aI2C_RawWrite** (U2A_HANDLE ***handle***, UInt16 ***I2C_Address***, Byte ***nBytes***, Byte ****Data***)

int ***CU2AClass::*I2C_RawWrite** (UInt16 ***I2C_Address***, Byte ***nBytes***, Byte ****Data***)

This function writes raw data to a device on the I^2C bus.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Address	The address of the I^2C device. May be 7 or 10 bits. See **u2aI2C_Control**.
nBytes	The number of bytes to be written.
Data	Pointer to array of bytes to be written.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.2.10 u2aI2C_BlockWriteBlockRead

int **u2aI2C_BlockWriteBlockRead** (U2A_HANDLE ***handle***, UInt16 ***I2C_Address***,
Byte ***nWriteBytes***, Byte ****WriteData***, Byte ***nReadBytes***,
Byte ****ReadData***)

int ***CU2AClass::*** **u2aI2C_BlockWriteBlockRead** (UInt16 ***I2C_Address***,
Byte ***nWriteBytes***, Byte ****WriteData***, Byte ***nReadBytes***,
Byte ****ReadData***)

This function writes a specified number of data bytes to a device on the I^{2}C bus, then, after a repeated START (i.e., no STOP), reads a specified number of data bytes from the device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Address	The address of the I^{2}C device. May be 7 or 10 bits. See **u2aI2C_Control**.
nWriteBytes	The number of bytes to be written.
WriteData	Pointer to array of bytes to be written.
nReadBytes	The number of bytes to be read.
ReadData	Pointer to a buffer to receive the data read.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.3 SPI Functions

5.3.1 u2aSPI_Control

int **u2aSPI_Control** (U2A_HANDLE ***handle***, SPI_ClockPhase ***SPI_ClockPhase***,
SPI_ClockPolarity ***SPI_ClockPolarity***,
SPI_BitDirection ***SPI_BitDirection***,
SPI_CharacterLength ***SPI_CharacterLength***,
SPI_ChipSelectType ***SPI_CSType***,
SPI_ChipSelectPolarity ***SPI_CSPolarity***,
Byte ***DividerHigh***, Byte ***DividerLow***)

int ***CU2AClass::SPI_Control*** (SPI_ClockPhase ***SPI_ClockPhase***,
SPI_ClockPolarity ***SPI_ClockPolarity***,
SPI_BitDirection ***SPI_BitDirection***,
SPI_CharacterLength ***SPI_CharacterLength***,
SPI_ChipSelectType ***SPI_CSType***,
SPI_ChipSelectPolarity ***SPI_CSPolarity***,
Byte ***DividerHigh***, Byte ***DividerLow***)

This function sets the parameters for SPI transactions.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

SPI_ClockPhase	Defines the edge of the clock on which data is captured:
	SPI_Capture_On_Trailing_Edge = 0
	SPI_Capture_On_Leading_Edge = 1

SPI_ClockPolarity	Defines which clock state is considered inactive:
	SPI_Inactive_State_Low = 0
	SPI_Inactive_State_High = 1

SPI_BitDirection	Defines which bit of each byte is sent first:
	SPI_LSB_First = 0
	SPI_MSB_First = 1

SPI_CharacterLength	Defines the number of data bits in each byte transferred:
	SPI__8_Bit = 0
	SPI__7_Bit = 1

SPI_CSType	Defines how the CS (chip select) signal is generated:
	SPI_ With_Every_Byte = 0
	SPI_With_Every_Packet = 1
	SPI_With_Every_Word = 2
	SPI_No_CS = 3
	SPI_Packets_With_MOSI_High = 4
	SPI_Pulse_After_Packet = 255

SPI_CSPolarity	Defines the state in which the CS (chip select) signal is active:
	SPI_Active_High = 0
	SPI_Active_Low = 1

DividerHigh	The high byte of the clock divider that sets the SPI bit rate.

DividerLow	The low byte of the clock divider that sets the SPI bit rate.

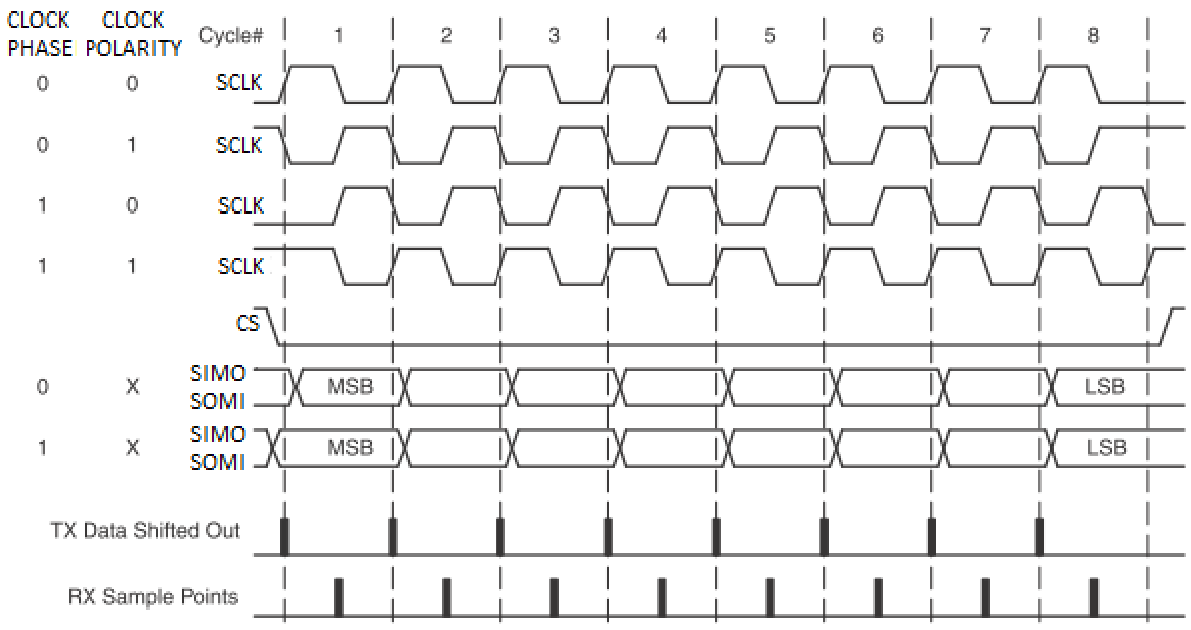
Comments:

The bit rate divisor can be calculated as follows:

```
divider = 24000000 / bit_rate_bps
_DividerHigh = (divider >> 8) & 0xFF
_DividerLow = divider & 0xFF
```

The table below shows divider constants for common bit rates:

Bit Rate (kbps)	Divider	_DividerHigh	_DividerLow
10	2400	9	96
25	960	3	192
50	480	1	224
100	240	0	240
125	192	0	192
200	120	0	120
250	96	0	96
400	60	0	60
500	48	0	48
800	30	0	30
1000	24	0	24
2000	12	0	12
4000	6	0	6
8000	3	0	3



Timing with various clock phase and polarity settings

When is ***SPI_CSType*** is set to SPI_Packets_With_MOSI_High (type 4), the chip select behavior is the same as when ***SPI_CSType*** is set to SPI_With_Every_Packet (type 1), but the MOSI signal will be held high between packets.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.3.2 u2aSPI_WriteAndRead

int **u2aSPI_WriteAndRead** (U2A_HANDLE ***handle***, Byte ***nBytes***, Byte ****Data***)

int ***CU2AClass::*SPI_WriteAndRead** (Byte ***nBytes***, Byte ****Data***)

This function simultaneously writes and reads data from an SPI device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nBytes	The number of bytes to be written/read.
Data	Pointer to array of bytes to be written/read.

Comments:

The ***Data*** buffer has a dual purpose and is used for both data to be sent and data received. When **u2aSPI_WriteAndRead()** is called, the buffer contains the data to be sent. On return, the buffer contains the data received.

Return:

Returns number of bytes read on success. If an error occurs, a negative error code is returned.

5.3.3 u2aSPI_WriteAndReadEx

int **u2aSPI_WriteAndReadEx** (U2A_HANDLE ***handle***, Byte nSS, Byte ***nBytes***, Byte ****Data***)

int ***CU2AClass::*SPI_WriteAndReadEx** (Byte nCS, Byte ***nBytes***, Byte ****Data***)

This function simultaneously writes and reads data from an SPI device, using a user-specified Slave Select (SS) signal.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nSS	The number of the GPIO to use for the SS signal.
nBytes	The number of bytes to be written/read.
Data	Pointer to array of bytes to be written/read.

Comments:

The nSS parameter specifies the number of the GPIO to be used for the Slave Select (SS) signal. The specified GPIO must be set as an output before calling this function. This needs to be done only once, unless the GPIO function is subsequently changed to something other than output. For example, to use **GPIO8** as the SS signal, you could do the following:

```
Byte nSS = 8;

u2aGPIO_SetPort(handle, nSS, GPIO_Output);
u2aSPI_WriteAndReadEx(handle, nSS, dataSize, dataBuf);
```

Note that **GPIO2**, **GPIO4**, and **GPIO5** are used for the SPI interface signals and, therefore, cannot be used for the SS signal. Also, the default SS signal (used by the **u2aSPI_WriteAndRead** function) uses **GPIO6**.

The ***Data*** buffer has a dual purpose and is used for both data to be sent and data received. When **u2aSPI_WriteAndRead()** is called, the buffer contains the data to be sent. On return, the buffer contains the data received.

Return:

Returns number of bytes read on success. If an error occurs, a negative error code is returned.

5.4 SMBus Functions

5.4.1 u2aSMBUS_Control

int **u2aSMBUS_Control** (U2A_HANDLE ***handle***, SMBUS_FLAGS ***Flags***)

int ***CU2AClass::*SMBUS_Control** (SMBUS_FLAGS ***Flags***)

This function sets the default value to be used when **SMBUS_PEC_DEFAULT** is used as the ***Flags*** parameter for other SMBus read/write functions.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Flags	Must be one of the following flag values:
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

Returns zero on success. On failure, an error code is returned (always negative).

5.4.2 u2aSMBUS_SendByte

int **u2aSMBUS_SendByte** (U2A_HANDLE ***handle***, UInt16 ***I2C_Slave_Address***, Byte ***CommandCode***, SMBUS_FLAGS ***Flags***)

int ***CU2AClass::*SMBUS_SendByte** (UInt16 ***I2C_Slave_Address***, Byte ***CommandCode***, SMBUS_FLAGS ***Flags***)

This function sends a single byte command to an SMBus device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.
CommandCode	The SMBus command to be sent.

Flags	Must be one of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

Returns zero on success. On failure, an error code is returned (always negative).

Comments:

If the SMBUS_PEC_CRC8 flag is used, the PEC is automatically calculated and appended to the end of the SMBus packet.

5.4.3 u2aSMBUS_WriteByte

int **u2aSMBUS_WriteByte** (U2A_HANDLE ***handle***, UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, Byte ***Data***, SMBUS_FLAGS ***Flags***)

int **CU2AClass::SMBUS_WriteByte** (UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, Byte ***Data***, SMBUS_FLAGS ***Flags***)

This function sends a single byte to an SMBus device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.
CommandCode	The SMBus command to be sent.
Data	The 8-bit data byte to be sent.

Flags	Must be one of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

Returns zero on success. On failure, an error code is returned (always negative).

Comments:

If the SMBUS_PEC_CRC8 flag is used, the PEC is automatically calculated and appended to the end of the SMBus packet.

5.4.4 u2aSMBUS_WriteWord

int **u2aSMBUS_WriteWord** (U2A_HANDLE ***handle***, UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, UInt16 ***Data***, SMBUS_FLAGS ***Flags***)

int **CU2AClass::SMBUS_WriteWord** (UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, UInt16 ***Data***, SMBUS_FLAGS ***Flags***)

This function sends a single byte to an SMBus device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.
CommandCode	The SMBus command to be sent.
Data	The 16-bit data word to be sent.

Flags	Must be one of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

Returns zero on success. On failure, an error code is returned (always negative).

Comments:

If the SMBUS_PEC_CRC8 flag is used, the PEC is automatically calculated and appended to the end of the SMBus packet.

5.4.5 u2aSMBUS_WriteBlock

int **u2aSMBUS_WriteBlock** (U2A_HANDLE ***handle***, UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, Byte ***Length***, Byte ****Data***,
SMBUS_FLAGS ***Flags***)

int **CU2AClass::SMBUS_WriteBlock** (UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, Byte ***Length***, Byte ****Data***,
SMBUS_FLAGS ***Flags***)

This function sends a single byte to an SMBus device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.
CommandCode	The SMBus command to be sent.
Length	The number of bytes of data to be sent.
Data	Pointer to array of data bytes to be sent.

Flags	Must be one of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

Returns zero on success. On failure, an error code is returned (always negative).

Comments:

If the SMBUS_PEC_CRC8 flag is used, the PEC is automatically calculated and appended to the end of the SMBus packet.

5.4.6 u2aSMBUS_ReceiveByte

int **u2aSMBUS_ReceiveByte** (U2A_HANDLE ***handle***, UInt16 ***I2C_Slave_Address***,
SMBUS_FLAGS ***Flags***)

int **CU2AClass::SMBUS_ReceiveByte** (UInt16 ***I2C_Slave_Address***, SMBUS_FLAGS
Flags)

This function receives a single byte from an SMBus device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.

Flags	Must be one of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

Returns the value of the byte read. On failure, an error code is returned (always negative).

Comments:

If the SMBUS_PEC_CRC8 flag is used, the PEC for the received data is automatically calculated and verified. If the calculated PEC does not match the received PEC, the error code ERR_DATA_READ_ERROR is returned.

5.4.7 u2aSMBUS_ReadByte

int **u2aSMBUS_ReadByte** (U2A_HANDLE ***handle***, UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, SMBUS_FLAGS ***Flags***)

int **CU2AClass::SMBUS_ReadByte** (UInt16 ***I2C_Slave_Address***, Byte ***CommandCode***,
SMBUS_FLAGS ***Flags***)

This function sends the specified command, and then receives a single byte from an SMBus device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.
CommandCode	The SMBus command to be sent.

Flags	Must be one of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

Returns the value of the byte read. On failure, an error code is returned (always negative).

Comments:

If the SMBUS_PEC_CRC8 flag is used, the PEC for the received data is automatically calculated and verified. If the calculated PEC does not match the received PEC, the error code ERR_DATA_READ_ERROR is returned.

5.4.8 u2aSMBUS_ReadWord

int **u2aSMBUS_ReadWord** (U2A_HANDLE ***handle***, UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, SMBUS_FLAGS ***Flags***)

int **CU2AClass::SMBUS_ReadWord** (UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, SMBUS_FLAGS ***Flags***)

This function sends the specified command, and then receives a 16-bit word from an SMBus device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.
CommandCode	The SMBus command to be sent.

Flags	Must be one of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

Returns the 16-bit value of the data read. On failure, an error code is returned (always negative).

Comments:

If the SMBUS_PEC_CRC8 flag is used, the PEC for the received data is automatically calculated and verified. If the calculated PEC does not match the received PEC, the error code ERR_DATA_READ_ERROR is returned.

5.4.9 u2aSMBUS_ReadBlock

int **u2aSMBUS_ReadBlock** (U2A_HANDLE ***handle***, UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, Byte ***Length***, Byte ****Data***,
SMBUS_FLAGS ***Flags***)

int **CU2AClass::SMBUS_ReadBlock** (UInt16 ***I2C_Slave_Address***,
Byte ***CommandCode***, Byte ***Length***, Byte ****Data***,
SMBUS_FLAGS ***Flags***)

This function sends the specified command, and then receives multiple bytes of data from an SMBus device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.
CommandCode	The SMBus command to be sent.
Length	The number of bytes of data to be received.
Data	Pointer to an array of bytes to store the data received.

Flags	Must be one or more of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2
	SMBUS_VAR_LEN = 8

Return:

Returns the number of bytes received on success. On failure, an error code is returned (always negative).

Comments:

If the SMBUS_PEC_CRC8 flag is used, the PEC for the received data is automatically calculated and verified. If the calculated PEC does not match the received PEC, the error code ERR_DATA_READ_ERROR is returned. When PEC is on, the SMBUS_VAR_LEN flag may be used to allow less than the requested number of bytes to be received.

5.4.10 u2aSMBUS_BlockWriteBlockRead

int **u2aSMBUS_BlockWriteBlockRead** (U2A_HANDLE nHandle, Byte I2C_***Slave_Address***, Byte ***CommandCode***, Byte ***WriteCount***, Byte ****WriteData,*** Byte ***ReadCount***, Byte ****ReadData,*** SMBUS_FLAGS ***Flags***)

int ***CU2AClass::*SMBUS_BlockWriteBlockRead** (Byte I2C_***Slave_Address***, Byte ***CommandCode***, Byte ***WriteCount***, Byte ****WriteData,*** Byte ***ReadCount***, Byte ****ReadData,*** SMBUS_FLAGS ***Flags***)

This function writes a command code, followed by a number of data bytes, to the slave address of a SMBus device on the I^{2}C bus. Then, after a repeated start (no STOP), it reads a number of data bytes.

Parameters:

nHandle	Valid handle (obtained from **u2aOpen**) for the USB2ANY device.
I2C_Slave_Address	The address of the SMBus device. Must be 7 bits.
CommandCode	The SMBus command to be sent.
WriteCount	The number of data bytes to be written.
WriteData	Pointer to array of the data bytes to be written.
ReadCount	The number of data bytes of data to be received.
ReadData	Pointer to an array of bytes to store the data received.

Flags	Must be one of the following flag values:
	SMBUS_PEC_DEFAULT = 0
	SMBUS_PEC_OFF = 1
	SMBUS_PEC_CRC8 = 2

Return:

If a positive number, the number of bytes read. On failure, an error code is returned (always negative).

5.4.11 u2aSMBUS_GetEchoBuffer

int **u2aSMBUS_GetEchoBuffer** (U2A_HANDLE handle, Byte ***nBufferSize***, Byte ****Buffer***)

int **CU2AClass::SMBUS_GetEchoBuffer** (Byte ***nBufferSize***, Byte ****Buffer***)

This function retrieves echoed data from a previously called SMBUS function.

Note: This function is currently unavailable.

Parameters:

nHandle	A valid handle, obtained by a call to the **u2aOpen** function.
nBufferSize	The size, in bytes, of the buffer pointed to by ***Buffer***.
Buffer	Pointer to a buffer to receive the echoed read/write data.

Return:

Returns zero on success. On failure, an error code is returned (always negative).

5.5 RFFE Functions

5.5.1 u2aRFFE_RegisterZeroWrite

int **u2aRFFE_RegisterZeroWrite** (U2A_HANDLE handle, Byte ***SlaveAddress***, Byte ***Data***)

int **CU2AClass::RFFE_RegisterZeroWrite** (Byte ***SlaveAddress***, Byte ***Data***)

This function writes seven bits of a single byte of data to register zero of a device.

Parameters:

nHandle	A valid handle, obtained by a call to the **u2aOpen** function.
SlaveAddress	Address of the slave device. Valid address is in the range 0x00 to 0x0F.
Data	Data byte to be written. *Note that bit 7 is not written*.

Return:

Returns zero or a positive number on success. On failure, an error code is returned (always negative).

5.5.2 u2aRFFE_RegisterWrite

int **u2aRFFE_RegisterWrite** (U2A_HANDLE handle, Byte ***SlaveAddress***, Byte ***RegisterAddress***, Byte ***Data***)

int **CU2AClass::RFFE_RegisterWrite** (Byte ***SlaveAddress***, Byte ***RegisterAddress***, Byte ***Data***)

This function writes a single byte of data to a register of a device.

Parameters:

nHandle	A valid handle, obtained by a call to the **u2aOpen** function.
SlaveAddress	Address of the slave device. Valid address is in the range 0x00 to 0x0F.
RegisterAddress	Address of register to which data will be written. Valid address is in the range 0x00 to 0x0F.
Data	Data to be written.

Return:

Returns zero or a positive number on success. On failure, an error code is returned (always negative).

5.5.3 u2aRFFE_ExtRegisterWrite

int **u2aRFFE_ExtRegisterWrite** (U2A_HANDLE handle, Byte ***SlaveAddress***,
Byte ***RegisterAddress***, Byte ***nLength***, Byte *****Data***)

int **CU2AClass::RFFE_ExtRegisterWrite** (Byte ***SlaveAddress***, Byte ***RegisterAddress***,
Byte ***nLength***, Byte *****Data***)

This function writes up to 16 bytes of data to registers of a device.

Parameters:

nHandle	A valid handle, obtained by a call to the **u2aOpen** function.
SlaveAddress	Address of the slave device. Valid address is in the range 0x00 to 0x0F.
RegisterAddress	Address of the first register to which data will be written. Valid address is in the range 0x00 to 0xFF.
nLength	Length, in bytes, of data to be written. Valid length is 1 to 16 bytes.
Data	Pointer to buffer containing data to be written.

Return:

Returns zero or a positive number on success. On failure, an error code is returned (always negative).

5.5.4 u2aRFFE_ExtRegisterWriteLong

int **u2aRFFE_ExtRegisterWriteLong** (U2A_HANDLE handle, Byte ***SlaveAddress***,
UInt16 ***RegisterAddress***, Byte ***nLength***,
Byte *****Data***)

int **CU2AClass::RFFE_ExtRegisterWriteLong** (Byte ***SlaveAddress***,
UInt16 ***RegisterAddress***, Byte ***nLength***,
Byte *****Data***)

This function writes up to 8 bytes of data to registers of a device.

Parameters:

nHandle	A valid handle, obtained by a call to the **u2aOpen** function.
SlaveAddress	Address of the slave device. Valid address is in the range 0x00 to 0x0F.
RegisterAddress	Address of the first register to which data will be written. Valid address is in the range 0x0000 to 0xFFFF.
nLength	Length, in bytes, of data to be written. Valid length is 1 to 8 bytes.
Data	Pointer to buffer containing data to be written.

Return:

Returns zero or a positive number on success. On failure, an error code is returned (always negative).

5.5.5 u2aRFFE_RegisterRead

int **u2aRFFE_RegisterRead** (U2A_HANDLE handle, Byte ***SlaveAddress***, Byte ***RegisterAddress***)

int **CU2AClass::RFFE_RegisterRead** (Byte ***SlaveAddress***, Byte ***RegisterAddress***)

This function reads a single byte of data from a register of a device.

Parameters:

nHandle	A valid handle, obtained by a call to the **u2aOpen** function.
SlaveAddress	Address of the slave device.
RegisterAddress	Address of register from which data will be read.

Return:

Returns the value of the unsigned data read (always positive) on success. On failure, an error code is returned (always negative).

5.5.6 u2aRFFE_ExtRegisterRead

int **u2aRFFE_ExtRegisterRead** (U2A_HANDLE handle, Byte ***SlaveAddress***, Byte ***RegisterAddress***, Byte ***nLength***, Byte ****Data***)

int **CU2AClass::RFFE_ExtRegisterRead** (Byte ***SlaveAddress***, Byte ***RegisterAddress***, Byte ***nLength***, Byte ****Data***)

This function reads up to 16 bytes of data from registers of a device.

Parameters:

nHandle	A valid handle, obtained by a call to the **u2aOpen** function.
SlaveAddress	Address of the slave device. Valid address is in the range 0x00 to 0x0F.
RegisterAddress	Address of the first register from which data will be read. Valid address is in the range 0x00 to 0xFF.
nLength	Length, in bytes, of data to be read. Valid length is 1 to 16 bytes.
Data	Pointer to buffer that will receive the data read. The buffer must be large enough to receive at least ***nLength*** bytes.

Return:

Returns zero or a positive number on success. On failure, an error code is returned (always negative).

5.5.7 u2aRFFE_ExtRegisterReadLong

int **u2aRFFE_ExtRegisterReadLong** (U2A_HANDLE handle, Byte ***SlaveAddress***,
UInt16 ***RegisterAddress***, Byte ***nLength***,
Byte ****Data***)

int **CU2AClass::RFFE_ExtRegisterReadLong** (Byte ***SlaveAddress***,
UInt16 ***RegisterAddress***, Byte ***nLength***,
Byte ****Data***)

This function reads up to 8 bytes of data from registers of a device.

Parameters:

nHandle	A valid handle, obtained by a call to the **u2aOpen** function.
SlaveAddress	Address of the slave device. Valid address is in the range 0x00 to 0x0F.
RegisterAddress	Address of the first register from which data will be read. Valid address is in the range 0x0000 to 0xFFFF.
nLength	Length, in bytes, of data to be read. Valid length is 1 to 8 bytes.
Data	Pointer to buffer that will receive the data read. The buffer must be large enough to receive at least ***nLength*** bytes.

Return:

Returns zero or a positive number on success. On failure, an error code is returned (always negative).

5.6 ADC Functions

5.6.1 u2aADC_Control

int **u2aADC_Control** (U2A_HANDLE ***handle***, ADC_PinFunction ***ADC0***,
ADC_PinFunction ***ADC1***, ADC_PinFunction ***ADC2***,
ADC_PinFunction ***ADC3***, ADC_VREF ***VREF***)

int **CU2AClass::ADC_Control** (ADC_PinFunction ***ADC0***, ADC_PinFunction ***ADC1***,
ADC_PinFunction ***ADC2***, ADC_PinFunction ***ADC3***,
ADC_VREF ***VREF***)

This function sets the parameters for the ADC pins.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

ADC0	Sets the function of the ADC0 pin:
	ADC_No_Change = 0
	ADC_Analog_In = 1

ADC1	Sets the function of the ADC1 pin:
	ADC_No_Change = 0
	ADC_Analog_In = 1

ADC2	Sets the function of the ADC2 pin:
	ADC_No_Change = 0
	ADC_Analog_In = 1

ADC3	Sets the function of the ADC3 pin:
	ADC_No_Change = 0
	ADC_Analog_In = 1

VREF	Sets the reference voltage for the ADC:
	ADC_VREF_1V5 = 0
	ADC_VREF_2V5 = 1
	ADC_VREF_3V3 = 2
	ADC_VREF_External = 3

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.6.2 u2aADC_ConvertAndRead

int **u2aADC_ConvertAndRead** (U2A_HANDLE ***handle***, Byte ***nBytes***, Byte *__*Data*__)

int **CU2AClass::ADC_ConvertAndRead** (Byte ***nBytes***, Byte *__*Data*__)

This function triggers an ADC conversion and reads the results.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nBytes	The size of the buffer for ADC result data – MUST be at least 12 bytes.
Data	Pointer to a buffer for the ADC result data.

Notes:

The USB2ANY has four ADC channels. The inputs for channels zero (ADC0) and one (ADC1) have pre-amplifiers with a gain of two. So, the ADC result for signals on these pins will be a value that represents double the actual voltage of the signal on the pins. This makes these inputs compatible with smaller signals (maximum of V_{REF} / 2).

The inputs for channels two (ADC2) and three (ADC3) *do not* have pre-amplifies, so the ADC result for signals on these pins represents the actual voltage of the signals.

The OneDemo does not have pre-amplifiers on any of the ADC inputs.

Return:

Returns a positive value on success. If an error occurs, a negative error code is returned. The results of the ADC conversion(s) are returned in the buffer pointed to by the ***Data*** parameter. The data in the buffer is formatted as follows:

Byte	Value
0	Always equal to 26
1	ADC0 result (High byte)
2	ADC0 result (Low byte)
3	ADC1 result (High byte)
4	ADC1 result (Low byte)
5	ADC2 result (High byte)
6	ADC2 result (Low byte)
7	ADC3 result (High byte)
8	ADC3 result (Low byte)
9	Reference voltage index (see ***VREF*** parameter of **u2aADC_Control** function)

5.6.3 u2aADC_Enable

int **u2aADC_Enable** (U2A_HANDLE ***handle***, int ***nChannel***, int ***nMode***)

int **CU2AClass::ADC_Enable** (int ***nChannel***, int ***nMode***)

This function enables or disables a specified ADC channel or range of channels.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nChannel	Specifies the ADC channel(s) to be enabled or disabled. When specifying a single channel, valid values are: 0 to 3 for USB2ANY, or 0 to 7 for OneDemo.
nMode	Defines whether the ADC channel is enabled or disabled: ADC_Disable = 0 Disables the specified single channel. ADC_Enable = 1 Enables the specified single channel. ADC_EnableMask = 2 Channels are enabled or disabled according to the bit mask specified by ***nChannel***. For each channel, the ADC is enabled if the corresponding bit is set (1) or disabled if the bit is clear (0).

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.6.4 u2aADC_SetReference

int **u2aADC_SetReference** (U2A_HANDLE ***handle***, ADC_VREF ***VREF***)

int **CU2AClass::ADC_SetReference** (ADC_VREF ***VREF***)

This function sets the voltage of the reference used for ADC conversion.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

VREF	Sets the reference voltage for the ADC:
	ADC_VREF_1V5 = 0
	ADC_VREF_2V5 = 1
	ADC_VREF_3V3 = 2
	ADC_VREF_External = 3

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.6.5 u2aADC_Acquire

int **u2aADC_Acquire** (U2A_HANDLE ***handle***, int ***nInterval***, int ***nSamples***)

int **CU2AClass::ADC_Acquire** (int ***nInterval***, int ***nSamples***)

This function immediately triggers ADC conversions on the specified channel(s). A total of ***nSamples*** samples are acquired, with an interval of ***nInterval*** microseconds between samples.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nInterval	The sampling interval, in microseconds. The interval must be in the range of 5 to 32767 microseconds. ***May not be less than 5 microseconds.***
nSamples	Number of data points to acquire for each specified channel. If ***nSamples*** is set to one, the ***nInterval*** parameter is ignored.

Notes:

Data is acquired for enabled channels (i.e., disabled channels are ignored) at each sample interval. See **u2aADC_Enable** for more about enabling/disabling channels.

There is a block of 4096 bytes of memory available to store the acquired data. Each data point requires two bytes of memory. The maximum number of samples per channel, per acquisition, is as follows:

Channels enabled	Max. Samples per Channel
1	2048
2	1024
3	682
4	512

Channels enabled	Max. Samples per Channel
5	409
6	341
7	292
8	256

Return:

On success, returns the number of data points (*not* samples) acquired. If an error occurs, a negative error code is returned. Use the **u2aADC_GetData** function to retrieve the actual data resulting from the conversion.

5.6.6 u2aADC_AcquireTriggered

int **u2aADC_AcquireTriggered**(U2A_HANDLE ***handle***, int ***nInterval***, int ***nSamples***,
int ***nTrigger***)

int **CU2AClass::ADC_AcquireTriggered**(int ***nInterval***, int ***nSamples***, int ***nTrigger***)

This function triggers ADC conversions on the specified channel(s) after the specified trigger event occurs. A total of ***nSamples*** samples are acquired, with an interval of ***nInterval*** microseconds between samples.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nInterval	The sampling interval, in microseconds. *May not be less than 5 microseconds.*
nSamples	Number of data points to acquire for each specified channel. If ***nSamples*** is set to one, the ***nInterval*** parameter is ignored.
nTrigger	The event that will start the ADC conversion. Must be one of the following values: ADC_EVENT_I2C = 1 ADC starts after any I2C command is received, but before it is executed.

Notes:

See **u2aADC_Acquire** Notes section for more information.

Return:

On success, returns the number of data points (*not* samples) acquired. If an error occurs, a negative error code is returned.

5.6.7 u2aADC_GetData

int **u2aADC_GetData** (U2A_HANDLE ***handle***, UInt16 ***nOffset***, UInt16 ***nDataPoints***,
UInt16 ****buffer***)

int **CU2AClass::ADC_GetData** (UInt16 ***nOffset***, UInt16 ***nDataPoints***, UInt16 ****buffer***)

This function retrieves the data acquired by the **u2aADC_Acquire** function.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nOffset	Starting offset, in 16-bit words, of data to be transferred.
nDataPoints	Number of data points (16-bit words) to be transferred. If this number is larger than the available data, or is larger than the maximum allowable transfer size, the number of data points transferred may be less. In that case, the return value will reflect the actual number of data points that were transferred.
buffer	Address of a buffer to receive the transferred data.

Notes:

The USB2ANY has four ADC channels. The inputs for channels zero (ADC0) and one (ADC1) have pre-amplifiers with a gain of two. So, the ADC result for signals on these pins will be a value that represents double the actual voltage of the signal on the pins. This makes these inputs compatible with smaller signals (maximum of V_{REF} / 2).

The inputs for channels two (ADC2) and three (ADC3) *do not* have pre-amplifies, so the ADC result for signals on these pins represents the actual voltage of the signals.

The OneDemo does not have pre-amplifiers on any of the ADC inputs.

Comments:

The returned data is structured as ***nDataPoints*** records, with each record consisting of two bytes that represent a single 16-bit data point. The records are arranged as a repeating list of values for each enabled channel, in channel order. The **u2aADC_Acquire** function returns the number of converted data points, which can be used as the ***nDataPoints*** parameter to this function. The available number of data points can also be obtained by calling the **u2aADC_GetStatus** function.

For example, if only channel 2 is enabled, the returned data values would be in the following format:

Value 1	Channel 2 data
Value 2	Channel 2 data
Value 3	Channel 2 data
Value 4	Channel 2 data
…	…
Value ***nDataPoints***	Channel 2 data

If channels 1, 3, and 5 are enabled, the returned data would be in the following format:

Value 1	Channel 1 data
Value 2	Channel 3 data
Value 3	Channel 5 data
Value 4	Channel 1 data
Value 5	Channel 3 data
Value 6	Channel 5 data
…	…
	Channel 1 data
	Channel 3 data
Value ***nDataPoints***	Channel 5 data

Return:

On success, returns the total number of data points copied to ***buffer***. If an error occurs, a negative error code is returned.

5.6.8 u2aADC_GetStatus

int **using** (U2A_HANDLE ***handle***, BYTE ****buffer***)

int **CU2AClass::ADC_GetStatus** (BYTE ****buffer***)

This function retrieves the status of the ADC interface.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
buffer	Address of a buffer to receive the status information. ***The buffer must be large enough to hold at least 9 bytes of data.***

Return:

On success, returns the status of the ADC interface. If an error occurs, a negative error code is returned.

The status is returned as follows:

Byte	Meaning
0	ADC channels enabled: Bit | Meaning 0 | ADC channel 0 is enabled 1 | ADC channel 1 is enabled 2 | ADC channel 2 is enabled 3 | ADC channel 3 is enabled 4 | ADC channel 4 is enabled 5 | ADC channel 5 is enabled 6 | ADC channel 6 is enabled 7 | ADC channel 7 is enabled

1	Reference source index:
	ADC_VREF_1V5 = 0
	ADC_VREF_2V5 = 1
	ADC_VREF_3V3 = 2
	ADC_VREF_External = 3

2	Non-zero if capture is in progress
3	Non-zero if new data is available for transfer
4	Number of channels captured.
5	Low byte of data size (in 16-bit samples)
6	High byte of data size
7	Reserved (always zero)
8	Reserved (always zero)

5.7 DAC Functions

5.7.1 u2aDACs_Write

int **u2aDACs_Write** (U2A_HANDLE ***handle***, DACs_WhichDAC ***DACs_WhichDAC***,
DACs_OperatingMode ***DACs_OperatingMode***, Byte ***Value***)

int **CU2AClass::DACs_Write** (DACs_WhichDAC ***DACs_WhichDAC***,
DACs_OperatingMode ***DACs_OperatingMode***, Byte ***Value***)

This function writes data to the specified DAC.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

DACs_WhichDAC	Which DAC to write to:
	DAC0 = 0
	DAC1 = 1

DACs_OperatingMode	Sets the operating mode (ignored by OneDemo):
	DACs_Normal = 0
	DACs_PWD_1k = 1
	DACs_PWD_100k = 2
	DACs_PWD_HiZ = 3

Value	Value to write to the DAC.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.7.2 u2aDAC_SetValue

int **u2aDAC_SetValue** (U2A_HANDLE ***handle***, int ***WhichDAC***, Byte ***Value***)

int **CU2AClass::DAC_SetValue** (int ***WhichDAC***, Byte ***Value***)

This function writes a value (to be converted to an output voltage) to the specified DAC.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

WhichDAC	Which DAC to write to:
	DAC0 = 0
	DAC1 = 1

Value	Value to write to the DAC.

Comments:

The table below shows the correlation between ***Value*** and DAC output voltage. The voltages shown are approximate and depend on many factors (e.g., load current, 3.3V reference accuracy, etc.). The DAC outputs are intended to be used for very low current loads, such as voltage references. The outputs are capable of sourcing up to about 20ma, but deviation from the output voltage shown in the table may increase as the current load increases.

Value	Volts	Value	Volts	Value	Volts	Value	Volts	Value	Volts	Value	Volts	Value	Volts	Value	Volts
00	0.000	20	0.414	40	0.828	60	1.242	80	1.656	A0	2.071	C0	2.485	E0	2.899
01	0.013	21	0.427	41	0.841	61	1.255	81	1.669	A1	2.084	C1	2.498	E1	2.912
02	0.026	22	0.440	42	0.854	62	1.268	82	1.682	A2	2.096	C2	2.511	E2	2.925
03	0.039	23	0.453	43	0.867	63	1.281	83	1.695	A3	2.109	C3	2.524	E3	2.938
04	0.052	24	0.466	44	0.880	64	1.294	84	1.708	A4	2.122	C4	2.536	E4	2.951
05	0.065	25	0.479	45	0.893	65	1.307	85	1.721	A5	2.135	C5	2.549	E5	2.964
06	0.078	26	0.492	46	0.906	66	1.320	86	1.734	A6	2.148	C6	2.562	E6	2.976
07	0.091	27	0.505	47	0.919	67	1.333	87	1.747	A7	2.161	C7	2.575	E7	2.989
08	0.104	28	0.518	48	0.932	68	1.346	88	1.760	A8	2.174	C8	2.588	E8	3.002
09	0.116	29	0.531	49	0.945	69	1.359	89	1.773	A9	2.187	C9	2.601	E9	3.015
0A	0.129	2A	0.544	4A	0.958	6A	1.372	8A	1.786	AA	2.200	CA	2.614	EA	3.028
0B	0.142	2B	0.556	4B	0.971	6B	1.385	8B	1.799	AB	2.213	CB	2.627	EB	3.041
0C	0.155	2C	0.569	4C	0.984	6C	1.398	8C	1.812	AC	2.226	CC	2.640	EC	3.054
0D	0.168	2D	0.582	4D	0.996	6D	1.411	8D	1.825	AD	2.239	CD	2.653	ED	3.067
0E	0.181	2E	0.595	4E	1.009	6E	1.424	8E	1.838	AE	2.252	CE	2.666	EE	3.080
0F	0.194	2F	0.608	4F	1.022	6F	1.436	8F	1.851	AF	2.265	CF	2.679	EF	3.093
10	0.207	30	0.621	50	1.035	70	1.449	90	1.864	B0	2.278	D0	2.692	F0	3.106
11	0.220	31	0.634	51	1.048	71	1.462	91	1.876	B1	2.291	D1	2.705	F1	3.119
12	0.233	32	0.647	52	1.061	72	1.475	92	1.889	B2	2.304	D2	2.718	F2	3.132
13	0.246	33	0.660	53	1.074	73	1.488	93	1.902	B3	2.316	D3	2.731	F3	3.145
14	0.259	34	0.673	54	1.087	74	1.501	94	1.915	B4	2.329	D4	2.744	F4	3.158
15	0.272	35	0.686	55	1.100	75	1.514	95	1.928	B5	2.342	D5	2.756	F5	3.171
16	0.285	36	0.699	56	1.113	76	1.527	96	1.941	B6	2.355	D6	2.769	F6	3.184
17	0.298	37	0.712	57	1.126	77	1.540	97	1.954	B7	2.368	D7	2.782	F7	3.196
18	0.311	38	0.725	58	1.139	78	1.553	98	1.967	B8	2.381	D8	2.795	F8	3.209
19	0.324	39	0.738	59	1.152	79	1.566	99	1.980	B9	2.394	D9	2.808	F9	3.222
1A	0.336	3A	0.751	5A	1.165	7A	1.579	9A	1.993	BA	2.407	DA	2.821	FA	3.235
1B	0.349	3B	0.764	5B	1.178	7B	1.592	9B	2.006	BB	2.420	DB	2.834	FB	3.248
1C	0.362	3C	0.776	5C	1.191	7C	1.605	9C	2.019	BC	2.433	DC	2.847	FC	3.261
1D	0.375	3D	0.789	5D	1.204	7D	1.618	9D	2.032	BD	2.446	DD	2.860	FD	3.274
1E	0.388	3E	0.802	5E	1.216	7E	1.631	9E	2.045	BE	2.459	DE	2.873	FE	3.287
1F	0.401	3F	0.815	5F	1.229	7F	1.644	9F	2.058	BF	2.472	DF	2.886	FF	3.300

Value vs. DAC output voltage

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.8 PWM Functions

5.8.1 u2aPWM_Control

int **u2aPWM_Control** (U2A_HANDLE ***handle***, PWM_ModeControl ***ModeControl***,
PWM_WhichPWM ***PWM_WhichPWM*****,**
PWM_InputDivider ***InputDivider*****,** UInt16 ***CompareRegister0***,
PWM_OutputMode ***OutputMode1*****,** UInt16 ***CompareRegister1***,
PWM_InputDividerEX ***InputDividerEX*****)**

int **CU2AClass::PWM_Control** (PWM_ModeControl ***ModeControl***,
PWM_WhichPWM ***PWM_WhichPWM*****,**
PWM_InputDivider ***InputDivider*****,** UInt16 ***CompareRegister0***,
PWM_OutputMode ***OutputMode1*****,** UInt16 ***CompareRegister1***,
PWM_InputDividerEX ***InputDividerEX*****)**

This function sets the parameters for the PWM pins.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

ModeControl	Sets the mode of the selected PWM channel:
	PWM_Stop = 0
	PWM_Up = 1
	PWM_Continuous = 2
	PWM_UP_Down = 3

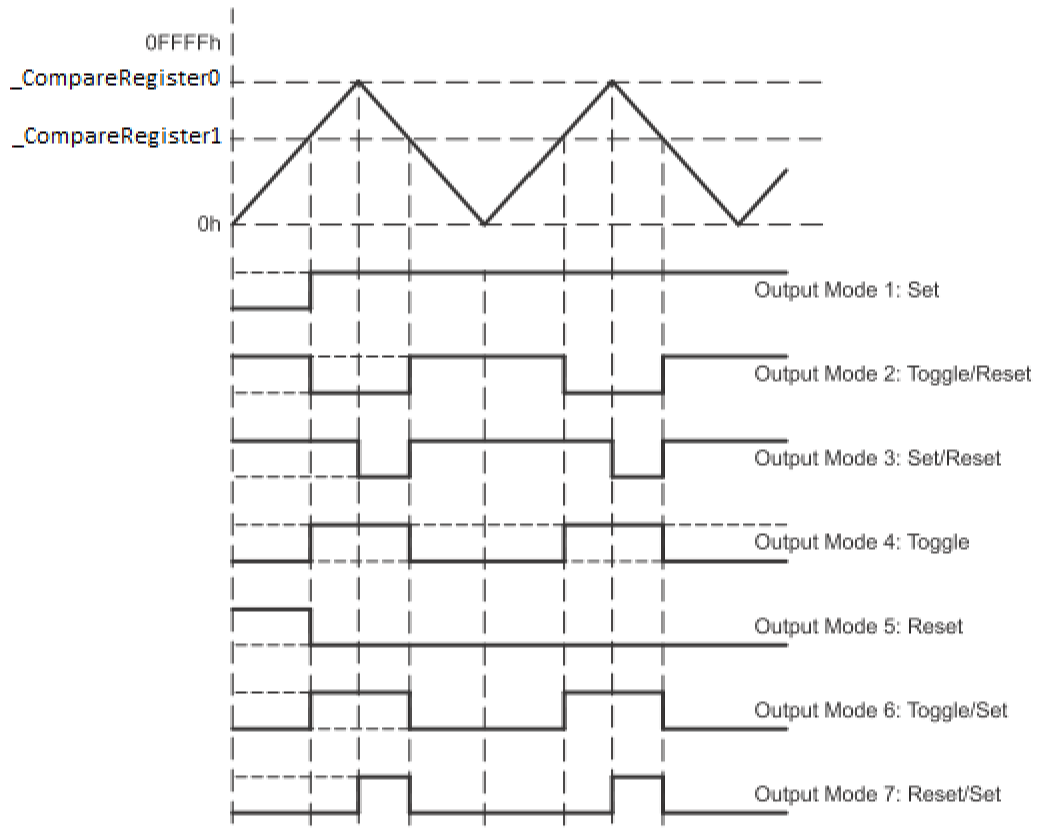
PWM_WhichPWM	Selects the PWM channel:
	PWM0 = 0
	PWM1 = 1
	PWM2 = 2
	PWM3 = 3

InputDivider	Sets the divider for the selected PWM channel:
	PWM_Div_1 = 0 (24 MHz)
	PWM_Div_2 = 1 (12 MHz)
	PWM_Div_4 = 2 (6 MHz)
	PWM_Div_8 = 3 (3 MHz)
	
	This determines the frequency at which the timer counts.

CompareRegister0	Frequency control (sets frequency). Timer counts up to this value, then counts down to 0.

OutputMode1	Sets the output mode for the selected PWM channel:
	PWM_Out_bit_value = 0
	PWM_Out_Set = 1
	PWM_Out_Toggle_Reset = 2
	PWM_Out_Set_Reset = 3
	PWM_Out_Toggle = 4
	PWM_Out_Reset = 5
	PWM_Out_Toggle_Set = 6
	PWM_Out_Reset_Set = 7

CompareRegister1	PWM control (sets duty cycle). This is the value of the timer count at which the output toggles.
InputDividerEX	Further divides the input clock. This must be one of the following values: 0 = /1 1 = /2 2 = /3 3 = /4 4 = /5 5 = /6 6 = /7 7 = /8 If set to a non-zero value, the actual input divider is the ***InputDivider*** divider times the ***InputDividerEX*** divider. For example, if you set ***InputDivider*** to 2 and ***InputDividerEX*** to 5, the input frequency would be calculated as follows: $FREQ_{IN}$ = 24 MHz / (4 * 6) = 24 MHz / 24 = 1MHz



Output Example – Timer in Up/Down Mode

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.9 UART Functions

5.9.1 u2aUART_Control

int **u2aUART_Control** (U2A_HANDLE ***handle***, UInt16 ***BaudRate***, UInt16 ***Parity***, UInt16 ***BitDirection***, UInt16 ***CharacterLength***, UInt16 ***StopBits***)

int **CU2AClass::UART_Control** (UInt16 ***BaudRate***, UInt16 ***Parity***, UInt16 ***BitDirection***, UInt16 ***CharacterLength***, UInt16 ***StopBits***)

This function sets the USART parameters for serial communication.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

BaudRate	Sets the baud rate for the USART transmitter and receiver:
	UART_9600_bps = 0
	UART_19200_bps = 1
	UART_38400_bps = 2
	UART_57600_bps = 3
	UART_115200_bps = 4
	UART_230400_bps = 5
	UART_300_bps = 6
	UART_320_bps = 7
	UART_600_bps = 8
	UART_1200_bps = 9
	UART_2400_bps = 10
	UART_4800_bps = 11

Parity	Sets the parity encoding/decoding:
	UART_None = 0
	UART_Even = 1
	UART_Odd = 2

BitDirection	Defines which bit of each byte is sent first:
	UART_LSB_First = 0 (recommended)
	UART_MSB_First = 1 (non-standard)

CharacterLength	Sets the number of data bits in each character:
	UART_8_Bit = 0
	UART_7_Bit = 1

StopBits	Sets the number of stop bits:
	UART_One_Stop = 0
	UART_Two_Stop = 1

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.9.2 u2aUART_Write

int **u2aUART_Write** (U2A_HANDLE ***handle***, Byte ***nBytes***, Byte ****Data***)

int **CU2AClass::UART_Write** (Byte ***nBytes***, Byte ****Data***)

This function writes serial data via the USART.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nBytes	The number of bytes to be written.
Data	Pointer to array of bytes to be written.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.9.3 u2aUART_Read

int **u2aUART_Read** (U2A_HANDLE ***handle***, Byte ***nBytes***, Byte ****Data***)

int **CU2AClass::UART_Read** (Byte ***nBytes***, Byte ****Data***)

This function reads serial data via the USART.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nBytes	The number of bytes to be read.
Data	Pointer to array of bytes to receive the data read.

Return:

On success, returns the number of bytes read. If an error occurs, a negative error code is returned.

5.9.4 u2aUART_DisableReceiver

int **u2aUART_DisableReceiver** (U2A_HANDLE ***handle***)

int **CU2AClass::UART_DisableReceiver** ()

This function disables the reception of serial data via the USART. Once this function is executed, the serial data receiver will remain disabled until the **u2aUART_Control** function is called.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.9.5 u2aUART_GetRxCount

int **u2aUART_GetRxCount** (U2A_HANDLE ***handle***)

int **CU2AClass::UART_GetRxCount** ()

This function checks the receive queue for data received by the UART and returns the number of data bytes that are currently available to be read.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Return:

On success, returns the number of bytes available to be read (possibly zero). If an error occurs, a negative error code is returned.

5.9.6 u2aUART_SetMode

int **u2aUART_SetMode** (U2A_HANDLE ***handle***, UInt16 ***mode***)

int **CU2AClass::UART_SetMode** (UInt16 ***mode***)

This is used to allow the UART functions to operate in special modes.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

mode	One of the follow mode constants:
	UART_Normal = 0
	UART_ReceiverOff = 1
	UART_RecvAfterXmit = 2

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.10 GPIO Functions

5.10.1 u2aGPIO_WriteControl

int **u2aGPIO_WriteControl** (U2A_HANDLE ***handle***, GPIO_PinFunction ***GPIO0***,
GPIO_PinFunction ***GPIO1***, GPIO_PinFunction ***GPIO2***, GPIO_PinFunction ***GPIO3***,
GPIO_PinFunction ***GPIO4***, GPIO_PinFunction ***GPIO5***, GPIO_PinFunction ***GPIO6***,
GPIO_PinFunction ***GPIO7***, GPIO_PinFunction ***GPIO8***, GPIO_PinFunction ***GPIO9***,
GPIO_PinFunction ***GPIO10***, GPIO_PinFunction ***GPIO11***, GPIO_PinFunction ***GPIO12***)

int **CU2AClass::GPIO_WriteControl** (GPIO_PinFunction ***GPIO0***, GPIO_PinFunction ***GPIO1***,
GPIO_PinFunction ***GPIO2***, GPIO_PinFunction ***GPIO3***, GPIO_PinFunction ***GPIO4***,
GPIO_PinFunction ***GPIO5***, GPIO_PinFunction ***GPIO6***, GPIO_PinFunction ***GPIO7***,
GPIO_PinFunction ***GPIO8***, GPIO_PinFunction ***GPIO9***, GPIO_PinFunction ***GPIO10***,
GPIO_PinFunction ***GPIO11***, GPIO_PinFunction ***GPIO12***)

This function writes control data to all of the GPIO pins, simultaneously.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
GPIO0	Function of GPIO pin 0
GPIO1	Function of GPIO pin 1
GPIO2	Function of GPIO pin 2
GPIO3	Function of GPIO pin 3
GPIO4	Function of GPIO pin 4
GPIO5	Function of GPIO pin 5
GPIO6	Function of GPIO pin 6
GPIO7	Function of GPIO pin 7
GPIO8	Function of GPIO pin 8
GPIO9	Function of GPIO pin 9
GPIO10	Function of GPIO pin 10
GPIO11	Function of GPIO pin 11
GPIO12	Function of GPIO pin 12

Comments:

The following constant values are used to define the function of each GPIO pin:

Constant	**Value**	**GPIO Pin Function**
GPIO_No_Change	0	The pin’s function is not changed.
GPIO_Output	1	Sets pin as an output.
GPIO_Input_No_Resistor	2	Sets pin as a floating input with no resistor.
GPIO_Input_Pull_Up	3	Sets pin as an input with a pull-up resistor.
GPIO_Input_Pull_Down	4	Sets pin as an input with a pull-down resistor.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.10.2 u2aGPIO_WriteState

int **u2aGPIO_WriteState** (U2A_HANDLE ***handle***, GPIO_OutPinState ***GPIO0***, GPIO_OutPinState ***GPIO1***, GPIO_OutPinState ***GPIO2***, GPIO_OutPinState ***GPIO3***, GPIO_OutPinState ***GPIO4***, GPIO_OutPinState ***GPIO5***, GPIO_OutPinState ***GPIO6***, GPIO_OutPinState ***GPIO7***, GPIO_OutPinState ***GPIO8***, GPIO_OutPinState ***GPIO9***, GPIO_OutPinState ***GPIO10***, GPIO_OutPinState ***GPIO11***, GPIO_OutPinState ***GPIO12***)

int **CU2AClass::GPIO_WriteState** (GPIO_OutPinState ***GPIO0***, GPIO_OutPinState ***GPIO1***, GPIO_OutPinState ***GPIO2***, GPIO_OutPinState ***GPIO3***, GPIO_OutPinState ***GPIO4***, GPIO_OutPinState ***GPIO5***, GPIO_OutPinState ***GPIO6***, GPIO_OutPinState ***GPIO7***, GPIO_OutPinState ***GPIO8***, GPIO_OutPinState ***GPIO9***, GPIO_OutPinState ***GPIO10***, GPIO_OutPinState ***GPIO11***, GPIO_OutPinState ***GPIO12***)

This function sets the output state of all GPIO pins, simultaneously.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
GPIO0	Output state of GPIO pin 0
GPIO1	Output state of GPIO pin 1
GPIO2	Output state of GPIO pin 2
GPIO3	Output state of GPIO pin 3
GPIO4	Output state of GPIO pin 4
GPIO5	Output state of GPIO pin 5
GPIO6	Output state of GPIO pin 6
GPIO7	Output state of GPIO pin 7
GPIO8	Output state of GPIO pin 8
GPIO9	Output state of GPIO pin 9
GPIO10	Output state of GPIO pin 10
GPIO11	Output state of GPIO pin 11
GPIO12	Output state of GPIO pin 12

Comments:

The following constant values are used to define the output state of each GPIO pin:

Constant	Value	GPIO Pin Function
GPIO_Out_No_Change	0	The pin's output is not changed.
GPIO_Out_Low	1	Sets pin's output to a low state.
GPIO_Out_High	2	Sets pin's output to a high state.

Important Note: This function affects ***only*** those pins that are configured as outputs.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.10.3 u2aGPIO_ReadState

int **u2aGPIO_ReadState** (U2A_HANDLE ***handle***, Byte ***nBytes***, Byte ****Data***)

int **CU2AClass::GPIO_ReadState** (Byte ***nBytes***, Byte ****Data***)

This function reads the input state of all of the GPIO pins, simultaneously.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nBytes	The size of the buffer to receive the data. Size must be at least 13 bytes to receive data for all GPIO pins.
Data	The address of a buffer to receive the data.

Important Note: This function returns valid values ***only*** for those pins that are configured as inputs. Data is received for pins set as outputs, but it is meaningless.

Return:

Returns number of bytes copied to the ***Data*** buffer on success. If an error occurs, a negative error code is returned.

Data is returned as an array of bytes. The first byte is the value read from GPIO0, the second byte is read from GPIO1, and so on. Data bytes are returned for all GPIO pins, regardless of whether they are configured as inputs. See the **Important Note** above.

5.10.4 u2aGPIO_SetPort

int **u2aGPIO_SetPort** (U2A_HANDLE ***handle***, Byte ***GPIO_Port***, Byte ***function***)

int **CU2AClass::GPIO_SetPort** (Byte ***GPIO_Port***, Byte ***function***)

This function configures a single GPIO pin as an output or input (with resistor options).

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
GPIO_Port	The GPIO port to be configured. Must be in the range 0 – 12.
function	A function code from the table below.

The ***function*** parameter must be set to one of the following constant values:

Constant	Value	GPIO Pin Function
GPIO_No_Change	0	The pin's function is not changed.
GPIO_Output	1	Sets pin as an output.
GPIO_Input_No_Resistor	2	Sets pin as a floating input with no resistor.
GPIO_Input_Pull_Up	3	Sets pin as an input with a pull-up resistor.
GPIO_Input_Pull_Down	4	Sets pin as an input with a pull-down resistor.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.10.5 u2aGPIO_WritePort

int **u2aGPIO_WritePort** (U2A_HANDLE ***handle***, Byte ***GPIO_Port***, Byte ***state***)

int **CU2AClass::GPIO_WritePort** (Byte ***GPIO_Port***, Byte ***state***)

This function sets the state of a single GPIO output pin.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
GPIO_Port	The GPIO port to be configured. Must be in the range 0 – 12.
state	A state constant from the table below.

The ***state*** parameter must be set to one of the following constant values:

Constant	Value	GPIO Pin Function
No_Change	0	The pin's output is not changed.
Low	1	Sets pin's output to a low state.
High	2	Sets pin's output to a high state.

Important Note: This function has no effect unless the specified pin is configured as an output.

Return:

Returns zero on success, or a negative error code on failure. Returns **ERR_INVALID_CONFIGURATION** if the specified pin is not configured as an output.

5.10.6 u2aGPIO_ReadPort

int **u2aGPIO_ReadPort** (U2A_HANDLE ***handle***, Byte ***GPIO_Port***)

int **CU2AClass::GPIO_ReadPort** (Byte ***GPIO_Port***)

This function reads the state of a single GPIO input pin.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
GPIO_Port	The GPIO port to be configured. Must be in the range 0 – 12.

Important Note: This function has no effect unless the specified pin is configured as an output.

Return:

On success, returns zero if the input pin is low, one if it is high. On error, a negative error code is returned. Returns **ERR_INVALID_CONFIGURATION** if the specified pin is not configured as an input.

5.10.7 u2aGPIO_WritePulse

int **u2aGPIO_WritePulse** (U2A_HANDLE ***handle***, Byte ***GPIO_Port***, Byte ***polarity***,
UInt16 ***duration***)

int **CU2AClass::GPIO_WritePulse** (Byte ***GPIO_Port***, Byte ***polarity***, UInt16 ***duration***)

This function outputs a single pulse on a specified GPIO output pin. The pulse can be either “high” or “low” with a width of 5 to 65535 microseconds

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
GPIO_Port	The GPIO port to be used for output. Must be in the range 0 – 12.
polarity	The polarity of the pulse: 0 = low pulse, 1 = high pulse.
duration	The desired pulse width, in microseconds. Must be at least 5 microseconds. May be set to zero to initialize the output state, without producing a pulse.

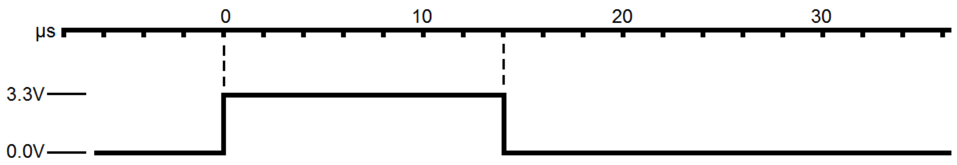
Comments:

This function will automatically configure the specified port as an output and set the initial state to the opposite of the polarity parameter.

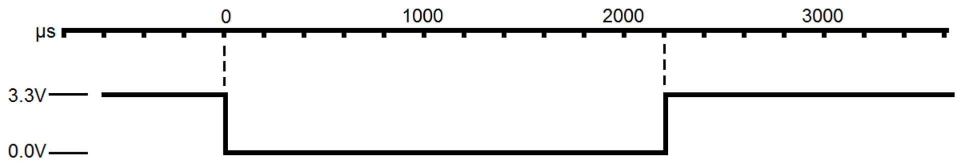
For best results, the pin specified by the ***GPIO_Port*** parameter should be preset to the correct initial state by calling this function with the ***polarity*** parameter set to the desired value and the ***duration*** parameter set to zero. Otherwise, if the initial state is incorrect, an unexpected extra edge may be produced.

The timing of the pulse is uncalibrated, but is typically accurate to within 0.5%.

Example A: ***polarity*** = 1, ***duration*** = 14



Example B: ***polarity*** = 0, ***duration*** = 2200



Return:

Returns zero on success, or a negative error code on failure. Returns the error code **ERR_INVALID_CONFIGURATION** if the specified pin is not valid, or the specified pulse width is less than 5 microseconds.

5.11 Memory Functions

5.11.1 u2aMSP430_ByteRead

int **u2aMSP430_ByteRead** (U2A_HANDLE ***handle***, UInt16 ***Address***)

int **CU2AClass::MSP430_ByteRead** (UInt16 ***Address***)

This function reads a single byte from the MSP430 memory.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Address	The 16-bit address of the byte to read.

Return:

On success, returns the unsigned value of the byte read. If an error occurs, a negative error code is returned.

5.11.2 u2aMSP430_ByteWrite

int **u2aMSP430_ByteWrite** (U2A_HANDLE ***handle***, UInt16 ***Address***, Byte ***Value***)

int **CU2AClass::MSP430_ByteWrite** (UInt16 ***Address***, Byte ***Value***)

This function writes a single byte to the MSP430 memory.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Address	The 16-bit address of the byte to write.
Value	The 8-bit value to write.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.11.3 u2aMSP430_WordRead

int **u2aMSP430_WordRead** (U2A_HANDLE ***handle***, UInt16 ***Address***)

int **CU2AClass::MSP430_WordRead** (UInt16 ***Address***)

This function reads a single word from the MSP430 memory.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Address	The 16-bit address of the word to read.

Return:

On success, returns the unsigned value of the word read. If an error occurs, a negative error code is returned.

5.11.4 u2aMSP430_WordWrite

int **u2aMSP430_WordWrite** (U2A_HANDLE ***handle***, UInt16 ***Address***, UInt16 ***Value***)

int **CU2AClass::MSP430_WordWrite** (UInt16 ***Address***, UInt16 ***Value***)

This function writes a single word to the MSP430 memory.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Address	The 16-bit address of the word to write.
Value	The 16-bit value to write.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.11.5 u2aMSP430_MemoryRead

int **u2aMSP430_MemoryRead** (U2A_HANDLE ***handle***, UInt16 ***Address***, UInt16 ***nBytes***,
Byte ****Data***)

int **CU2AClass::MSP430_MemoryRead** (UInt16 ***Address***, UInt16 ***nBytes***, Byte ****Data***)

This function reads a user-defined number of bytes from the MSP430 memory.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Address	The 16-bit address of the first byte of MSP430 memory to read.
nBytes	The number of bytes to be read. Must be in the range of 1-54 bytes.
Data	Pointer to a buffer to receive the bytes read.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.11.6 u2aMSP430_MemoryWrite

int **u2aMSP430_MemoryWrite** (U2A_HANDLE ***handle***, UInt16 ***Address***, UInt16 ***nBytes***, Byte ****Data***)

int **CU2AClass::MSP430_MemoryWrite** (UInt16 ***Address***, UInt16 ***nBytes***, Byte ****Data***)

This function writes a user-defined number of bytes to the MSP430 memory.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Address	The 16-bit address of the first byte of MSP430 memory to write.
nBytes	The number of bytes to be written. Must be in the range of 1-54 bytes.
Data	Pointer to a buffer of bytes to be written.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.12 FEC Functions

5.12.1 u2aFEC_Control (Deprecated)

int **u2aFEC_Control** (U2A_HANDLE ***handle***, UInt16 ***Interval***, UInt16 ***Enable***,
UInt16 ***Divider0***, UInt16 ***Divider1***)

Warning: This function is deprecated. Use **u2aFEC_Configure** for all new development.

This function configures the Frequency and Event Counter (FEC).

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Interval	A number representing the desired sample interval. See the table below.
Enable	Set to one (1) to enable FEC, zero (0) to disable it.
Divider0	Value for ID (input divider) register. See MSP430F5529 User's Guide.
Divider1	Value for TAIDEX register. See MSP430F5529 User's Guide.

Comments:

The ***Interval*** parameter must be a value from the following table:

Interval	**Sample period**
0	1 millisecond
1	2 milliseconds
2	5 milliseconds
3	10 milliseconds
4	20 milliseconds
5	50 milliseconds
6	100 milliseconds
7	200 milliseconds
8	500 milliseconds
9	1 second
10	2 seconds
11	5 seconds
12	10 seconds

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.12.2 u2aFEC_Configure

int **u2aFEC_Configure** (U2A_HANDLE ***handle***, UInt16 ***Enable***, UInt16 ***Interval***,
UInt16 ***Divider***)

int **CU2AClass::FEC_Configure** (UInt16 ***Enable***, UInt16 ***Interval***, UInt16 ***Divider***)

This function configures the Frequency and Event Counter (FEC).

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Enable	Set to one (1) to enable FEC, zero (0) to disable it.
Interval	A number representing the desired sample interval. See the table below.
Divider	The desired input frequency divider. See the table below.

Comments:

The ***Interval*** parameter must be a value from the following table:

Interval	Sample period	*Interval*	Sample period
0	1 millisecond	7	200 milliseconds
1	2 milliseconds	8	500 milliseconds
2	5 milliseconds	9	1 second
3	10 milliseconds	10	2 seconds
4	20 milliseconds	11	5 seconds
5	50 milliseconds	12	10 seconds
6	100 milliseconds		

The ***Divider*** parameter must be one of the values from the following table:

1	6	14	32
2	7	16	40
3	8	20	48
4	10	24	56
5	12	28	64

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.12.3 u2aFEC_CountAndRead

int **u2aFEC_CountAndRead** (U2A_HANDLE ***handle***)

int **CU2AClass::FEC_CountAndRead** ()

This function starts the Frequency and Event Counter (FEC) and then attempts to retrieve the result.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Comments:

About 20 milliseconds after starting the FEC, an attempt is made to retrieve the result. If the attempt is unsuccessful, a second attempt is made after a delay of 25 milliseconds. If both attempts fail, an error code of **ERR_NO_DATA** is returned.

The **u2aFEC_CountAndRead** function will generally return the **ERR_NO_DATA** code when the ***Interval*** parameter passed to the **u2aFEC_Configure** function specifies an interval greater than 20 milliseconds. In that case, you can retrieve the result by calling the **u2aFEC_GetResult** function after the time period specified by the ***Interval*** parameter has passed. The **u2aFEC_CountAndRead** function was designed in this way to allow longer intervals to be handled asynchronously.

Return:

On success, returns the number of events (may be zero) counted during the specified time interval. If an error occurs, a negative error code is returned. See Comments section for an explanation of the **ERR_NO_DATA** error code.

If the input to the FEC is a periodic waveform, you can calculate the frequency (in Hz), using the following formula:

$$\text{freq}_{\text{Hz}} = (1.0 / \text{Interval}_{\text{secs}}) * \text{ReturnValue}$$

5.12.4 u2aFEC_PurgeBuffer

int **u2aFEC_PurgeBuffer** (U2A_HANDLE ***handle***)

int **CU2AClass::FEC_PurgeBuffer** ()

This function purges any un-retrieved results from the FEC results buffer.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.12.5 u2aFEC_GetResult

int **u2aFEC_GetResult** (U2A_HANDLE ***handle***)

int **CU2AClass::FEC_GetResult** ()

This function retrieves the result a call to the **u2aFEC_CountAndRead** function.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Return:

On success, returns the result (event count) of the **u2aFEC_CountAndRead** function call. If an error occurs, a negative error code is returned.

Comments:

This function should be called *only* if a call to the **u2aFEC_CountAndRead** function returned an error code of **ERR_NO_DATA.** If the **u2aFEC_CountAndRead** function returns a valid event count, this function will *always* return the **ERR_NO_DATA** code.

5.13 Interrupt Functions

5.13.1 u2aInterrupt_Control

int **u2aInterrupt_Control** (U2A_HANDLE ***handle***, UInt16 ***Interrupt_PinFunction0***,
UInt16 ***Interrupt_PinFunction1***,
UInt16 ***Interrupt_PinFunction2***,
UInt16 ***Interrupt_PinFunction3***)

int **CU2AClass::Interrupt_Control** (UInt16 ***Interrupt_PinFunction0***,
UInt16 ***Interrupt_PinFunction1***,
UInt16 ***Interrupt_PinFunction2***,
UInt16 ***Interrupt_PinFunction3***)

This function is used to enable, disable, and set the parameters of the four interrupt pins on the USB2ANY.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Interrupt_PinFunction0	Sets the state of interrupt pin 0 (INT0). ***Not available on OneDemo.***
Interrupt_PinFunction1	Sets the state of interrupt pin 1 (INT1).
Interrupt_PinFunction2	Sets the state of interrupt pin 2 (INT2).
Interrupt_PinFunction3	Sets the state of interrupt pin 3 (INT3).

Comments:

For each interrupt pin, the state is set according to the following table:

Parameter	Value	Description
Intr_No_Change	0	No change
Intr_Falling_Edge	1	Interrupt enabled on falling edge
Intr_Rising_Edge	2	Interrupt enabled on rising edge
Intr_Disabled	3	Interrupt disabled

Note: INT0 is not available on OneDemo because the pin is used for EVM Detect.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.13.2 u2aInterrupt_CheckReceived

int **u2aInterrupt_CheckReceived** (U2A_HANDLE ***handle***, int ***Channel***, BOOL ***Reset***)

int **CU2AClass::Interrupt_CheckReceived** (int ***Channel***, BOOL ***Reset***)

This function is used to check whether an interrupt has occurred on the specified channel.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Channel	The interrupt channel to check (0=INT0, 1 = INT1, 2=INT2, 3=INT3)
Reset	If set TRUE, resets the count for the specified channel to zero, after the current count is read. If set FALSE, the count will continue to increment for each interrupt received.

Return:

Returns the number of interrupts received for the specified channel. If an error occurs, a negative error code is returned.

5.13.3 u2aInterrupt_SetHandler

int **u2aInterrupt_SetHandler** (U2A_HANDLE ***handle***, int ***Channel***, void ****Callback***, HWND ***Hwnd***, int ***MsgNum***)

int **CU2AClass::Interrupt_SetHandler** (int ***Channel***, void ****Callback***, HWND ***Hwnd***, int ***MsgNum***)

This function is used to configure the notification method used by the interrupt handler.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Channel	The interrupt channel to configure (0=INT0, 1 = INT1, 2=INT2, 3=INT3)
Callback	The address of the callback function to be called when the specified interrupt occurs. If set to NULL, no function is called. See **Comments** section for more information.
Hwnd	The handle of the window to receive a notification message when the specified interrupt occurs. If set to NULL, no message is sent. See **Comments** section for more information.
MsgNum	The message number to be sent to the window specified by ***Hwnd***.

Comments:

When ***Callback*** is set to a non-Null value, a function at the specified address will be called when an interrupt occurs on the specified ***Channel***. That function must have the following prototype:

```
void _stdcall CallBack(DWORD channel, DWORD count)
```

There will be two parameters passed to the function. The first parameter, `channel`, will be set to the channel number of the interrupt (i.e., 0 – 3). This allows multiple interrupts to share the same callback function.

The second parameter, `count`, is set to the number of interrupts that occurred since the last call to the callback function. Normally, this will be set to one. However, if additional interrupts occur before the callback function returns, they will be accumulated and reported by the next call to the callback function. To avoid accumulation of interrupts, the callback function should be written such that it processes each call in the minimum amount of time, with minimum overhead.

When ***Hwnd*** is set to a non-Null value, a message will be posted to the specified window's queue. The message will be in the standard Windows message format, with two parameters: ***wParam*** and ***lParam***. The ***wParam*** parameter will be set to the interrupt channel number and the ***lParam*** parameter will be set to the interrupt count, as with the callback function.

If both ***Callback*** and ***Hwnd*** are set to NULL, the **u2aInterrupt_CheckReceived** function may be used to determine whether interrupts have occurred, using a polling method. Note that, if desired, both ***Callback*** and ***Hwnd*** methods can be used simultaneously.

To turn off either (or both) of the ***Callback*** and ***Hwnd*** methods, call this function with the appropriate parameter(s) set to NULL, always making sure that *both* parameters are set to the desired values.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.14 Digital Capture Functions

5.14.1 u2aDigital_Capture

int **u2aDigital_Capture** (U2A_HANDLE ***handle***, UInt32 ***frequency***,
UInt16 ***samples***, UInt16 ***timeframe***)

int **CU2AClass::Digital_Capture** (UInt32 ***frequency***, UInt16 ***samples***,
UInt16 ***timeframe***)

This function is used to initiate capture of digital data via the GPIO7 input pin.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
frequency	The sampling frequency in Hertz. Must be in the range of 16 Hz to 262143 Hz.
timeframe	A number representing the length of the time window, during which the digital signal is sampled. Must be one of the following values:

timeframe	**Capture time (milliseconds)**
0	1
1	2
2	5
3	10
4	20
5	50
6	100
7	200
8	500
9	1000
10	2000
11	5000
12	10000

Comments:

The actual sampling frequency is derived by dividing the USB2ANY's master clock by an integer value. This limits the number of actual frequencies available. The actual frequency used for the sampling interval is calculated by an algorithm that chooses a frequency at least as high as the requested frequency (possibly much higher). The difference between the requested and actual frequencies is much more obvious at higher frequencies, where the clock divider is a small number. For example, requesting any frequency over 250,000 Hz will result in the maximum sampling frequency of 333,333 Hz.

Return:

On success, returns the actual sampling frequency (see **Comments** section for more information). If an error occurs, a negative error code is returned.

5.15 EasyScale™ Functions

5.15.1 u2aEasyScale_Control

int **u2aEasyScale_Control** (U2A_HANDLE ***handle***, UInt16 ***UpperThreshold***,
UInt16 ***LowerThreshold***)

int **CU2AClass::EasyScale_Control** (UInt16 ***UpperThreshold***, UInt16 ***LowerThreshold***)

This function is used to set the voltage thresholds for EasyScale™ communications.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
UpperThreshold	The upper voltage threshold used for determining data states.
LowerThreshold	The lower voltage threshold used for determining data states.

Comments:

The table below shows the values available for the ***UpperThreshold*** and ***LowerThreshold*** parameters and the approximate voltage they represent:

Value	Volts	Value	Volts	Value	Volts	Value	Volts	Value	Volts	Value	Volts	Value	Volts	Value	Volts
00	0.000	20	0.414	40	0.828	60	1.242	80	1.656	A0	2.071	C0	2.485	E0	2.899
01	0.013	21	0.427	41	0.841	61	1.255	81	1.669	A1	2.084	C1	2.498	E1	2.912
02	0.026	22	0.440	42	0.854	62	1.268	82	1.682	A2	2.096	C2	2.511	E2	2.925
03	0.039	23	0.453	43	0.867	63	1.281	83	1.695	A3	2.109	C3	2.524	E3	2.938
04	0.052	24	0.466	44	0.880	64	1.294	84	1.708	A4	2.122	C4	2.536	E4	2.951
05	0.065	25	0.479	45	0.893	65	1.307	85	1.721	A5	2.135	C5	2.549	E5	2.964
06	0.078	26	0.492	46	0.906	66	1.320	86	1.734	A6	2.148	C6	2.562	E6	2.976
07	0.091	27	0.505	47	0.919	67	1.333	87	1.747	A7	2.161	C7	2.575	E7	2.989
08	0.104	28	0.518	48	0.932	68	1.346	88	1.760	A8	2.174	C8	2.588	E8	3.002
09	0.116	29	0.531	49	0.945	69	1.359	89	1.773	A9	2.187	C9	2.601	E9	3.015
0A	0.129	2A	0.544	4A	0.958	6A	1.372	8A	1.786	AA	2.200	CA	2.614	EA	3.028
0B	0.142	2B	0.556	4B	0.971	6B	1.385	8B	1.799	AB	2.213	CB	2.627	EB	3.041
0C	0.155	2C	0.569	4C	0.984	6C	1.398	8C	1.812	AC	2.226	CC	2.640	EC	3.054
0D	0.168	2D	0.582	4D	0.996	6D	1.411	8D	1.825	AD	2.239	CD	2.653	ED	3.067
0E	0.181	2E	0.595	4E	1.009	6E	1.424	8E	1.838	AE	2.252	CE	2.666	EE	3.080
0F	0.194	2F	0.608	4F	1.022	6F	1.436	8F	1.851	AF	2.265	CF	2.679	EF	3.093
10	0.207	30	0.621	50	1.035	70	1.449	90	1.864	B0	2.278	D0	2.692	F0	3.106
11	0.220	31	0.634	51	1.048	71	1.462	91	1.876	B1	2.291	D1	2.705	F1	3.119
12	0.233	32	0.647	52	1.061	72	1.475	92	1.889	B2	2.304	D2	2.718	F2	3.132
13	0.246	33	0.660	53	1.074	73	1.488	93	1.902	B3	2.316	D3	2.731	F3	3.145
14	0.259	34	0.673	54	1.087	74	1.501	94	1.915	B4	2.329	D4	2.744	F4	3.158
15	0.272	35	0.686	55	1.100	75	1.514	95	1.928	B5	2.342	D5	2.756	F5	3.171
16	0.285	36	0.699	56	1.113	76	1.527	96	1.941	B6	2.355	D6	2.769	F6	3.184
17	0.298	37	0.712	57	1.126	77	1.540	97	1.954	B7	2.368	D7	2.782	F7	3.196
18	0.311	38	0.725	58	1.139	78	1.553	98	1.967	B8	2.381	D8	2.795	F8	3.209
19	0.324	39	0.738	59	1.152	79	1.566	99	1.980	B9	2.394	D9	2.808	F9	3.222
1A	0.336	3A	0.751	5A	1.165	7A	1.579	9A	1.993	BA	2.407	DA	2.821	FA	3.235
1B	0.349	3B	0.764	5B	1.178	7B	1.592	9B	2.006	BB	2.420	DB	2.834	FB	3.248
1C	0.362	3C	0.776	5C	1.191	7C	1.605	9C	2.019	BC	2.433	DC	2.847	FC	3.261
1D	0.375	3D	0.789	5D	1.204	7D	1.618	9D	2.032	BD	2.446	DD	2.860	FD	3.274
1E	0.388	3E	0.802	5E	1.216	7E	1.631	9E	2.045	BE	2.459	DE	2.873	FE	3.287
1F	0.401	3F	0.815	5F	1.229	7F	1.644	9F	2.058	BF	2.472	DF	2.886	FF	3.300

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.15.2 u2aEasyScale_Write

int **u2aEasyScale_Write** (U2A_HANDLE ***handle***, BYTE ***DataBytes[ ]***, UInt16 ***nBytes***, BYTE ****DataBits***, UInt16 ***nBits***, UInt16 ***WriteSpeed***, UInt16 ***WriteACK***)

int **CU2AClass::EasyScale_Write** (BYTE ***DataBytes[ ]***, UInt16 ***nBytes***, BYTE ****DataBits***, UInt16 ***nBits***, UInt16 ***WriteSpeed***, UInt16 ***WriteACK***)

This function is used to write data to an EasyScale™ device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
DataBytes	An array of data bytes to be sent.
nBytes	The number of bytes in the ***DataBytes*** array.
DataBits	An array of additional bits to send.
nBits	The number of additional bits in the ***DataBits*** array.
WriteSpeed	The bit rate at which to send the data. See table in the **Comments** section.
WriteACK	Set to 1 to check for an ACK from the device, else set to 0.

Comments:

The ***DataBytes*** parameter is a pointer to an array of bytes, with each element containing eight (8) bits of data to be sent. If the total number of bits to be sent is not an even multiple of eight, the remaining bits are in an array of bytes pointed to by the ***DataBits*** parameter.

The ***DataBits*** array may contain up to seven (7) elements, with each element representing one bit. If an element contains a zero (0) value, the bit value is interpreted to be zero (0). Any non-zero value (1 – 255) is construed to represent a bit value of one (1).

The total number of data bits sent to the device is (***DataBytes* * 8) + *DataBits***. Because of a packet size limitation and packet header overhead, the maximum number of bits that can be written is 408.

The ***WriteSpeed*** parameter must be one of the values from the following table:

WriteSpeed	**Bit rate**
0	10 kbps
1	50 kbps
2	100 kbps
3	200 kbps
4	400 kbps

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.15.3 u2aEasyScale_Read

int **u2aEasyScale_Read** (U2A_HANDLE ***handle***, UInt16 ***ReadSpeed***, BYTE ***nBytes***, BYTE ***nBits***, BYTE ***DataBuffer[ ])***

int **CU2AClass::EasyScale_Read** (UInt16 ***ReadSpeed***, BYTE ***nBytes***, BYTE ***nBits***, BYTE ***DataBuffer[ ])***

This function is used to read data from an EasyScale™ device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
ReadSpeed	The bit rate at which to read the data. See table in the **Comments** section.
nBytes	The number of full 8-bit bytes to be read.
nBits	The number of additional bits to be read.
DataBuffer	An array to receive the data read. See **Comments** section for size requirement.

Comments:

The ***DataBuffer*** parameter is a pointer to an array of bytes, which must be large enough to receive all of the data, plus some overhead data. The size of the array must be at least ***nBytes* + 3** bytes.

Because of a packet size limitation and packet header overhead, the maximum number of bits that can be read is 408.

The ***ReadSpeed*** parameter must be one of the values from the following table:

ReadSpeed	**Bit rate**
0	10 kbps
1	50 kbps
2	100 kbps
3	200 kbps
4	400 kbps

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.15.4 u2aEasyScale_WriteAndRead

int **u2aEasyScale_WriteAndRead** (U2A_HANDLE ***handle***, UInt16 ***nBits***,
BYTE ***WriteData[ ]***, BYTE ***ReadData[ ]***, BYTE ***WriteSpeed***,
BYTE ***WriteACK***)

int **CU2AClass::EasyScale_WriteAndRead** (UInt16 ***nBits***, BYTE ***WriteData[ ]***,
BYTE ***ReadData[]***, BYTE ***WriteSpeed***, BYTE ***WriteACK***)

This function is used to simultaneously write data to, and read data from, an EasyScale™ device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nBits	The total number of bits to be written and read. (See **Comments** section)
WriteData	An array of the data bytes to be sent.
ReadData	An array to receive the data bytes that are read.
WriteSpeed	The bit rate at which to send the data. See table in the **Comments** section.
WriteACK	Set to 1 to check for an ACK from the device, else set to 0.

Comments:

The ***WriteData*** parameter is a pointer to an array of bytes, with each byte containing eight (8) bits of data to be sent. If the total number of bits to be sent (***nBits***) is not an even multiple of eight, the upper bits of the last byte will contain the remaining bits (any unnecessary lower bits are ignored). For example, if 12 bits of data (010101100111) are to be sent, two bytes would be required (*X* represents ignored values):

56 7*X* (hex) or 01010110 0111*XXXX* (binary)

The size of the ***WriteData*** and ***ReadData*** arrays must be large enough to hold the data to be written and read, respectively. The size (in bytes) can be calculated as follows:

array size = (int)((nBits + 7) / 8)

Because of a packet size limitation and packet header overhead, the maximum allowable value of the ***nBits*** parameter is 408.

The ***WriteSpeed*** parameter must be one of the values from the following table:

WriteSpeed	**Bit rate**
0	10 kbps
1	50 kbps
2	100 kbps

Return:

Returns the number of bits read on success. If an error occurs, a negative error code is returned.

5.16 DisplayScale™ Functions

5.16.1 u2aDisplayScale_Setup

int **u2aDisplayScale_Setup** (U2A_HANDLE ***handle***, UInt16 ***options***, UInt16 ***speed***)

int **CU2AClass:: DisplayScale_Setup** (UInt16 ***options***, UInt16 ***speed***)

This function is used to set the parameters for DisplayScale™ communications when using the **u2aDisplayScale_*xxxx*** I/O functions.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
options	Set to zero (0) to use the default behavior: write on GPIO0 and read on the rising edge of GPIO1. The behavior may be modified by adding one or more of the following values: 1 = Read on falling edge, instead of the rising edge 2 = Use GPIO7 for writing and GPIO6 for reading
speed	The bit rate used for communications, in kHz. Valid values are 15, 50, and 100.

Comments:

There are currently no DisplayScale options, so the ***options*** parameter is ignored.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.16.2 u2aDisplayScale_WriteReg

int **u2aDisplayScale_WriteReg** (U2A_HANDLE ***handle***, BYTE ***address***, BYTE ***Data***)

int **CU2AClass:: DisplayScale_WriteReg** (BYTE ***address***, BYTE ***Data***)

This function is used to write a byte of data to a DisplayScale™ register.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
address	7-bit register address
Data	A byte of data to be written to the register.

Comments:

The **u2aDisplayScale_Setup** function must be called at least once, to set the communications parameters, before calling this function.

Data is sent with the bits transmitted from the most-significant to the least-significant, for each byte.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.16.3 u2aDisplayScale_ReadReg

int **u2aDisplayScale_ReadReg**(U2A_HANDLE ***handle***, BYTE ***address***)

int **CU2AClass:: DisplayScale_ReadReg**(BYTE ***address***)

This function is used to read a byte of data from a DisplayScale™ register.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
address	7-bit register address

Comments:

The **u2aDisplayScale_Setup** function must be called at least once, to set the communications parameters, before calling this function.

Data is read with the bits received from the most-significant to the least-significant, for each byte.

Return:

Returns the byte read on success. If an error occurs, a negative error code is returned.

5.16.4 u2aDisplayScale_WriteAndRead

int **u2aDisplayScale_WriteAndRead** (U2A_HANDLE ***handle***, UInt16 ***nBits***,
BYTE ***WriteData[ ]***, BYTE ***ReadData[ ]***)

int **CU2AClass::DisplayScale_WriteAndRead** (UInt16 ***nBits***, BYTE ***WriteData[ ]***,
BYTE ***ReadData[ ]***)

This function is used to simultaneously write data to, and read data from, a DisplayScale™ device.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
nBits	The total number of bits to be written and read. (See **Comments** section)
WriteData	An array of the data bytes to be sent.
ReadData	An array to receive the data bytes that are read.

Comments:

The ***WriteData*** parameter is a pointer to an array of bytes, with each byte containing eight (8) bits of data to be sent. If the total number of bits to be sent (***nBits***) is not an even multiple of eight, the upper bits of the last byte will contain the remaining bits (any unnecessary lower bits are ignored). For example, if 12 bits of data (010101100111) are to be sent, two bytes would be required (*X* represents ignored values):

56 7*X* (hex) or 01010110 0111*XXXX* (binary)

The size of the ***WriteData*** and ***ReadData*** arrays must be large enough to hold the data to be written and read, respectively. The size (in bytes) can be calculated as follows:

array size = (int)((nBits + 7) / 8)

Because of a packet size limitation and packet header overhead, the maximum allowable value of the ***nBits*** parameter is 408.

Return:

Returns the number of bits read on success. If an error occurs, a negative error code is returned.

5.17 OneWire Functions

5.17.1 u2aOneWire_SetMode

int **u2aOneWire_SetMode** (U2A_HANDLE ***handle***, UInt16 ***mode***)

int **CU2AClass::OneWire_SetMode (UInt16 *mode*)**

This function sets the OneWire interface protocol mode or disables it.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
mode	Sets the OneWire interface mode.

Comments:

The following table shows the valid values for the ***mode*** parameter:

Constant Name	Value	Protocol Description	Output Pin
OW_Disable	0	Disables the OneWire interface.	N/A
OW_Mode1	1	Enables Pulse Mode.	OW1
OW_Mode2	2	Enables Pulse Mode.	OW2
OW_Mode3	3	Enables Pulse Mode with Address.	OW3
OW_Mode4	4	Enables Pulse Mode.	OW4
OW_Mode5	5	Enables Pulse Mode with Address and extended (x10) pulse time.	OW5

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.17.2 u2aOneWire_PulseSetup

int **u2aOneWire_PulseSetup** (U2A_HANDLE ***handle***, UInt16 ***timeSetup***,
UInt16 ***timeLow***, UInt16 ***timeHigh***, UInt16 ***timeStore,***
int flags)

int **CU2AClass::OneWire_PulseSetup (UInt16 *timeSetup*, UInt16 *timeLow*,
UInt16 *timeHigh*, UInt16 *timeStore*, int flags)**

This function sets the timing constants used by OneWire Pulse Mode on the currently enabled OW*x* pin.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
timeSetup	Defines the setup time, in microseconds. The valid range is 100 µs to 65535 µs. The default ***timeSetup*** value is 0 microseconds*****.
timeLow	Defines the low pulse period, in microseconds. The valid range is 5µs to 65535µs. The default ***timeLow*** value is 10 µs*****. If the OneWire interface is currently enabled in Mode 3, this value also sets the low pulse period for the address pulses.
timeHigh	Defines the high pulse period, in microseconds. The valid range is 5µs to 65535µs. The default ***timeHigh*** value is 10 µs*****. If the OneWire interface is currently enabled in Mode 3, this value also sets the high pulse period for the address pulses.
timeStore	Defines the store time, in microseconds. The valid range is 100 µs to 65535 µs. The default ***timeStore*** value is 0 microseconds*****.
flags	May be one or more (combined with logical OR) of the following values:

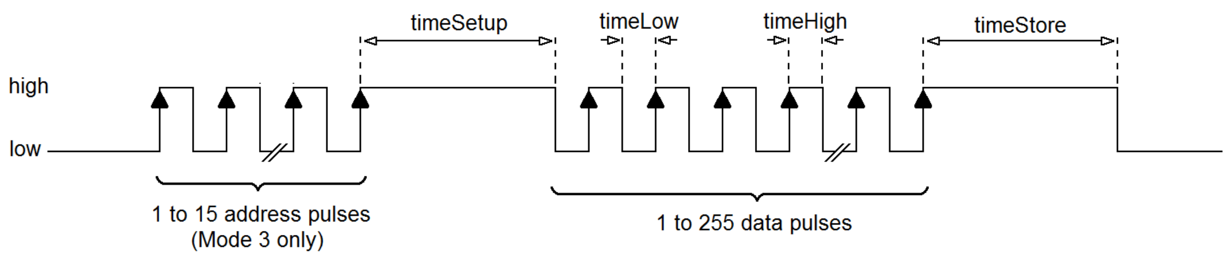
Flag Name	**Value**	**Description**
OW_FLAG_NONE*	0	Standard signal, as shown in **Comments** section.
OW_FLAG_INVERT	1	Invert all signal states.
OW_FLAG_NO_INIT	2	Suppress automatic setting of the signal's initial and final states.

*The default is OW_FLAG_NONE, a non-inverted signal with automatic setting of the signal's initial and final states.

***** The default setting will be in effect on the currently enabled OW*x* pin after the USB2ANY is reset and until ***u2aOneWire_PulseSetup*** is called to change the parameters.

Comments:

The following diagram illustrates how the timing constants affect the output signal.



Note: The resolution of all timers used for OneWire is 5µs. I.e., the time value specified for each parameter will be rounded down to the nearest multiple of 5µs. For example, whether you specify 10, 11, or 14 for the ***timeLow*** parameter, it will end up being set to 10µs.

The timeSetup and timeStore intervals of the output signal are optional and will be present only if the ***timeSetup*** and/or ***timeStore*** parameters are set to a value greater than zero.

In Mode 5, the actual time for all timing variables is the time specified multiplied by 10.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.17.3 u2aOneWire_PulseWrite

int **u2aOneWire_PulseWrite** (U2A_HANDLE ***handle***, Byte ***address***, Byte ***pulses***)

int **CU2AClass::OneWire_PulseWrite (**Byte ***address,*** Byte ***pulses*)**

This function outputs the specified number of pulses on the currently selected OneWire interface output pin.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
address	Specifies the address for Mode 3 protocol. Ignored by other protocols. The valid range is 1 to 15.
pulses	Specifies the number of pulses to output. The valid range is 1 to 255.

Comments:

This function is very similar to the **u2aOneWire_PulseWriteEx** function, except that the range of the ***address*** and ***pulses*** parameters is limited. If a larger value is needed for the ***address*** or ***pulses*** parameters, use the **u2aOneWire_PulseWriteEx** function instead of this one.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.17.4 u2aOneWire_PulseWriteEx

int **u2aOneWire_PulseWriteEx** (U2A_HANDLE ***handle***, Byte ***address***, Uint16 ***pulses***)

int **CU2AClass::OneWire_PulseWriteEx (**Byte ***address,*** UInt16 ***pulses*)**

This function outputs the specified number of pulses on the currently selected OneWire interface output pin.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
address	Specifies the address for Mode 3 protocol. Ignored by other protocols. The valid range is 1 to 255.
pulses	Specifies the number of pulses to output. The valid range is 1 to 511.

Comments:

This function is very similar to the **u2aOneWire_PulseWrite** function, except that the range of the ***address*** and ***pulses*** parameters is extended.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.17.5 u2aOneWire_SetOutput

int **u2aOneWire_SetOutput** (U2A_HANDLE ***handle***, Byte ***state***)

int **CU2AClass::OneWire_SetOutput (Byte *state*)**

This function sets the state of the output signal of the currently selected OneWire interface output pin.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
state	The desired state of the output. May be 0 (low) or 1 (high).

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.18 Power Functions

5.18.1 u2aPower_WriteControl (Deprecated)

int **u2aPower_WriteControl** (U2A_HANDLE ***handle***, Power_3V3 ***Power_3V3***,
Power_5V0 ***Power_5V0***)

Warning: This function is deprecated. Use **u2aPower_Enable** for all new development.

This function enables/disables the 3.3V and 5.0V power outputs.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Power_3V3	Sets the state of the **+3.3V_EXT** power output pin. Valid values are:
	Power_3V3_OFF = 0
	Power_3V3_ON = 1

Power_5V0	Sets the state of the **+5V_EXT** power output pin. Valid values are:
	Power_5V0_OFF = 0
	Power_5V0_ON = 1

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.18.2 u2aPower_Enable

int **u2aPower_Enable** (U2A_HANDLE ***handle***, int ***Enable3V3***, int ***Enable5v0***,
int ***EnableAdj***)

int **CU2AClass::Power_Enable** (int ***Enable3V3***, int ***Enable5v0***, int ***EnableAdj***)

This function enables/disables the 3.3V, 5.0V, and Adjustable power outputs.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Enable3V3	Sets the state of the **+3.3V_EXT** power output pin. Valid values are:
	Power_3V3_Disable = 0
	Power_3V3_Enable = 1
	Power_3V3_Ignore = 2

Enable5v0	Sets the state of the **+5V_EXT** power output pin. Valid values are:
	Power_5V0_Disable = 0
	Power_5V0_Enable = 1
	Power_5V0_Ignore = 2

EnableAdj	**ONEDEMO** Sets the state of the **ADJ_2.5-5V_DUT** power output pin. Valid values are:
	Power_ADJ_Disable = 0
	Power_ADJ_Enable = 1
	Power_ADJ_Ignore = 2

Comments:

For each output, Power_xxx_Disable turns the power OFF, Power_xxx_Enable turns it ON, and Power_xxx_Ignore leaves it in its current state.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.18.3 u2aPower_SetVoltageRef

int **u2aPower_SetVoltageRef** (U2A_HANDLE ***handle***, int ***Value***)

int **CU2AClass::Power_SetVoltageRef** (int ***Value***)

This function sets the output voltage of the adjustable power supply.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
Value	The 8-bit DAC reference value used to set the output voltage.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.18.4 u2aPower_ReadStatus

int **u2aPower_ReadStatus** (U2A_HANDLE ***handle***)

int **CU2AClass::Power_ReadStatus** ()

This function reads the status of the 3.3V and 5.0V power outputs.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Return:

On success, returns one of the values from the table below. If an error occurs, a negative error code is returned.

Return values for USB2ANY:

Return Value	3.3V Power	5.0V Power
0	OK	OK
1	FAULT	OK
2	OK	FAULT
3	FAULT	FAULT

Return values for OneDemo:

Return Value	3.3V Power	5.0V Power	Adjustable
0	OK	OK	OK
1	FAULT	OK	OK
2	OK	FAULT	OK
3	FAULT	FAULT	OK
4	OK	OK	FAULT
5	FAULT	OK	FAULT
6	OK	FAULT	FAULT
7	FAULT	FAULT	FAULT

5.19 Miscellaneous Functions

5.19.1 u2aFirmwareVersion_Read

int **u2aFirmwareVersion_Read**(U2A_HANDLE ***handle***, BYTE ****pBuffer***, int ***nBufferSize***)

int **CU2AClass::FirmwareVersion_Read**(BYTE ****pBuffer***, int ***nBufferSize***)

This function reads the firmware version number of the USB2ANY controller associated with the specified handle.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.
pBuffer	Pointer to a buffer, which will receive the version data. The buffer must be large enough to receive at least four (4) bytes of data.
dwBufferSize	The size of the buffer (in bytes), pointed to by pBuffer.

Return:

Returns the number of bytes received and copied to the provided buffer. If an error occurs, a negative error code is returned.

The format of the data returned in the buffer pointed to by ***pBuffer*** is as follows:

Byte Offset	Description
0	Major version
1	Minor version
2	Major revision
3	Minor revision

5.19.2 u2aLED_WriteControl (USB2ANY)

int **u2aLED_WriteControl** (U2A_HANDLE ***handle***, LED ***LEDState***)

int **CU2AClass::LED_WriteControl** (LED ***LEDState***)

This function is used to control the LED on the USB2ANY.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

LEDState	Sets the state of the on-board LED. Valid values are:
	LED_OFF = 0
	LED_ON = 1
	LED_TOGGLE = 2

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.19.3 u2aLED_SetState (OneDemo)

int **u2aLED_SetState** (U2A_HANDLE ***handle***, int ***LEDState***, int ***BlinkCode***)

int **CU2AClass::LED_SetState** (int ***LEDState***, int ***BlinkCode***)

This function is used to control the green LED on the USB2ANY or the bi-color (red/green) LED on the OneDemo.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

LEDState	Sets the state of the on-board LEDs. Valid value is one or more of:
	LEDS_OFF = 0
	LEDS_RED_ON = 1 (ignored by USB2ANY)
	LEDS_GREEN_ON = 2
	LEDS_TOGGLE = 4

BlinkCode	Sets the blink pattern of the green LED or bi-color LED. See the table in the **Comments** section for descriptions of the available blink patterns.

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

Comments:

This function can used to set the visible color of the LED to red, green, or yellow (both LEDs on).

If the LEDS_TOGGLE bit is set, the current state of the LED(s) is toggled and the other bits of the parameter are ignored.

Note that, when ***BlinkCode*** is set non-zero on the OneDemo, the LEDs blink simultaneously, in the color defined by ***LEDState*** (except for LEDS_BLINK_ALT – see description below). The blink patterns are defined by the descriptions in the following table:

Symbolic Name	Value	Description of repeating blink pattern
LEDS_NON_BLINK	0	When on, LEDs will not blink.

Symbolic Name	**Value**	**Description of repeating blink pattern**
LEDS_BLINK_1	1	BLINK pause
LEDS_BLINK_2	2	BLINK BLINK pause
LEDS_BLINK_3	3	BLINK BLINK BLINK pause
LEDS_BLINK_4	4	BLINK BLINK BLINK BLINK pause
LEDS_BLINK_FAST	5	Fast blinking, with no pause
LEDS_BLINK_ALT	6	OneDemo: Red and green LEDs blink alternately USB2ANY: Slow blinking, with no pause

When a non-zero ***BlinkCode*** is specified, the LED(s) will blink in the color specified by the ***LEDState*** parameter. For LEDS_BLINK_ALT to work properly on the OneDemo, both red and green LEDs must be turned on.

5.19.4 u2aClock_Control

int **u2aClock_Control** (U2A_HANDLE ***handle***, ClockDivider1 ***ClockDivider1***,
ClockDivider2 ***ClockDivider2*****)**

int **CU2AClass::Clock_Control** (ClockDivider1 ***ClockDivider1***, ClockDivider2
ClockDivider2**)**

This function sets the clock divider.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

ClockDivider1	Sets the first clock divider. Must be one of the following values:
	ClockDiv1_1 = 0
	ClockDiv1_2 = 1
	ClockDiv1_4 = 2
	ClockDiv1_8 = 3
	ClockDiv1_16 = 4
	ClockDiv1_32 = 5

ClockDivider2	Sets the second clock divider. Must be one of the following values:
	ClockDiv2_1 = 0
	ClockDiv2_2 = 1
	ClockDiv2_4 = 2
	ClockDiv2_8 = 3
	ClockDiv2_16 = 4
	ClockDiv2_32 = 5

Return:

Returns zero on success. If an error occurs, a negative error code is returned.

5.20 Status Functions

5.20.1 u2aStatus_IsUSB2ANYConnected

int **u2aStatus_IsUSB2ANYConnected** ()

int **CU2AClass::Status_IsUSB2ANYConnected** ()

This function reports whether the USB2ANY (or OneDemo) is connected.

Parameters:

None

Comments:

The return value of this function is meaningful only after a successful call to the **u2aEnableDeviceDetect** function. Otherwise, the connection status cannot be determined and the return value is always FALSE.

Return:

Returns TRUE (1) if a USB2ANY or OneDemo device is connected, otherwise FALSE (0).

5.20.2 u2aStatus_GetErrorCode

int **u2aStatus_GetErrorCode** (U2A_HANDLE ***handle***)

int **CU2AClass::Status_GetErrorCode** ()

This function retrieves the most recent status/error code for the device specified by ***handle***.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Comments:

The **u2aStatus_GetErrorCode** function returns error status for the last API function executed. Therefore, it is recommended that the function be called immediately after the API function for which the error status is desired. The previous error status is cleared before every API function is executed and new error status is stored only until the next API function is executed.

This is a “one-shot” function that returns error codes only once for the most recently executed function. Any reported error code is cleared immediately after this function executes. The returned error status must be stored locally if it will be used more than once.

If this function is called more than once between API commands, only the first call will return an error code and all subsequent calls will return zero (i.e., no error).

Return:

Returns the most recent status/error code for last API function called on the specified device.

5.20.3 u2aStatus_GetText

char * **u2aStatus_GetText** (int ***Code***, char ****TextBuffer***, UInt16 ***nBufferSize***)

char * **CU2AClass::Status_GetText** (int ***Code***, char ****TextBuffer***, UInt16 ***nBufferSize***)

This function retrieves readable English text for a given status/error code.

Parameters:

Code	The status/error code for which to obtain readable text.
TextBuffer	Pointer to a buffer to receive the null-terminated text.
nBufferSize	The size of the buffer to receive the text.

Return:

Returns a pointer to ***TextBuffer***, which will contain the text for the specified code. If ***TextBuffer*** is too small to hold the entire text, the text will be truncated. A buffer size of at least 40 bytes (i.e., ***nBufferSize*** >= 40) is recommended.

5.20.4 u2aStatus_GetControllerType

int **u2aStatus_GetControllerType** (U2A_HANDLE ***handle***)

int **CU2AClass::Status_GetControllerType** ()

This function retrieves the controller type opened on ***handle***.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Return:

Returns a value that designates the controller type:

Returned Value	Controller Type
0	Unknown
1	USB2ANY
2	OneDemo

5.20.5 u2aStatus_EVMDetect

int **u2aStatus_EVMDetect** (U2A_HANDLE ***handle***, int ***nFunction***, void ****Callback***)

int **CU2AClass::Status_EVMDetect** (int ***nFunction***, void ****Callback***)

This function controls the operation of the **EVM DETECT** pin.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

nFunction	Defines the function to be performed:
	EVM_DetectDisable = 0
	EVM_DetectEnable = 1
	EVM_DetectQuery = 2

Callback	The address of the callback function to be called when an EVM board is connected or disconnected. If set to NULL, no function is called, but board connection/disconnection can be detected use a polled mode. See **Comments** section for more information. This parameter is ignored if ***nFunction*** is not set to 1 (EVM_DetectEnable).

Comments:

The EVM DETECT signal shares a pin with INT0 signal. When EVM detection is enabled, INT0 will not be available for other purposes.

If ***Callback*** is set to a non-Null value, the function at the specified address will be called when an EVM board is connected or disconnected. The callback function must have the following prototype:

```
void _stdcall CallBack(int channel, DWORD flag)
```

There will be two parameters passed to the function. The value of the first parameter is the interrupt identifier (always equal to 4). The value of the second parameter will be zero (0) if an EVM board was disconnected or one (1) if a board was connected.

If the use of a callback function is not desired, EVM board connection/disconnection can be detected using a polling method. To use the polling method, do the following:

1) Enable EVM board detection in polled mode:

```
u2aStatus_EVMDetect(handle, 1, NULL);
```

2) Call **u2aStatus_EVMDetect** with ***nFunction*** set to 2 and ***Callback*** set to NULL:

```
Ret = u2aStatus_EVMDetect(handle, 2, NULL);
```

3) The function will return one (1) if a board is connected or zero (0) if not.

Return:

The **EVM_DetectQuery** function returns zero if the **EVM DETECT** pin is low, one if it is high. On success, the other functions return zero. If an error occurs, a negative error code is returned.

5.20.6 u2aStatus_GetBoardRevision

int **u2aStatus_GetBoardRevision** (U2A_HANDLE ***handle***)

int **CU2AClass::Status_GetBoardRevision** ()

This function returns the hardware revision level of the OneDemo board that is currently open.

Parameters:

handle	A valid handle, obtained by a call to the **u2aOpen** function.

Return:

On success, the PCB Assembly revision number is returned. The revision number is always a positive value, representing a revision letter (0=A, 1=B, 2=C, etc.). If an error occurs, a negative error code is returned.

The USB2ANY is not able to detect the board revision and always returns zero (0) in response to this function.

Appendix A: Error Codes

Virtually all of the API functions return an integer value. In general, the return value represents an error code if it is a negative value. In most cases, a return value of zero means “Success” or “No error”. Some functions return the result of the function as a positive number.

See the **Return:** section of the each individual function description for specific details regarding the return value.

Error codes and descriptions by value

Value	Mnemonic	Description
0	ERR_OK	No error
-1	ERR_COM_RX_OVERFLOW	Receiver overflowed
-2	ERR_COM_RX_BUF_EMPTY	Receive buffer is empty
-3	ERR_COM_TX_BUF_FULL	Transmit buffer is full
-4	ERR_COM_TX_STALLED	Transmit is stalled
-5	ERR_COM_TX_FAILED	Transmit failed
-6	ERR_COM_OPEN_FAILED	Failed to open communications port
-7	ERR_COM_PORT_NOT_OPEN	Communications port is not open
-8	ERR_COM_PORT_IS_OPEN	Communications port is open
-9	ERR_COM_READ_TIMEOUT	Receive timeout
-10	ERR_COM_READ_ERROR	Communications port read error
-11	ERR_COM_WRITE_ERROR	Communications port write error
-12	ERR_DEVICE_NOT_FOUND	Communications device not found
-13	ERR_COM_CRC_FAILED	Communications CRC failed
-20	ERR_INVALID_PORT	Invalid port
-21	ERR_ADDRESS_OUT_OF_RANGE	Address is out of accepted range
-22	ERR_INVALID_FUNCTION_CODE	Invalid function code
-23	ERR_BAD_PACKET_SIZE	Invalid packet size
-24	ERR_INVALID_HANDLE	Invalid handle
-25	ERR_OPERATION_FAILED	Operation failed
-26	ERR_PARAM_OUT_OF_RANGE	Parameter is out of range
-27	ERR_PACKET_OUT_OF_SEQUENCE	Packet is out of sequence
-28	ERR_INVALID_PACKET_HEADER	Invalid packet header
-29	ERR_UNIMPLEMENTED_FUNCTION	Function not implemented
-30	ERR_TOO_MUCH_DATA	Too much data
-31	ERR_INVALID_DEVICE	Invalid device
-32	ERR_UNSUPPORTED_FIRMWARE	Unsupported firmware version
-33	ERR_BUFFER_TOO_SMALL	Buffer is too small
-34	ERR_NO_DATA	No data available
-35	ERR_RESOURCE_CONFLICT	Resource conflict
-36	ERR_NO_EVM	EVM is required for external power
-37	ERR_COMMAND_BUSY	Command is busy
-38	ERR_ADJ_POWER_FAIL	Adjustable power supply failure
-39	ERR_NOT_ENABLED	Interface or mode is not enabled
-40	ERR_I2C_INIT_ERROR	I2C initialization failed
-41	ERR_I2C_READ_ERROR	I2C read error
-42	ERR_I2C_WRITE_ERROR	I2C write error
-43	ERR_I2C_BUSY	I2C busy (transfer is pending)

Value	Mnemonic	Description
-44	ERR_I2C_ADDR_NAK	Address not acknowledged (NAK)
-45	ERR_I2C_DATA_NAK	Data not acknowledged (NAK)
-46	ERR_I2C_READ_TIMEOUT	Read timeout
-47	ERR_I2C_READ_DATA_TIMEOUT	Read data timeout
-48	ERR_I2C_READ_COMP_TIMEOUT	Timeout waiting for read complete
-49	ERR_I2C_WRITE_TIMEOUT	Write timeout
-50	ERR_I2C_WRITE_DATA_TIMEOUT	Write data timeout
-51	ERR_I2C_WRITE_COMP_TIMEOUT	Timeout waiting for write complete
-52	ERR_I2C_NOT_MASTER	I2C not in Master mode
-53	ERR_I2C_ARBITRATION_LOST	I2C arbitration lost
-54	ERR_I2C_NO_PULLUP_POWER	I2C pullups require the 3.3V EXT power to be on
-60	ERR_SPI_INIT_ERROR	SPI initialization failed
-61	ERR_SPI_WRITE_READ_ERROR	SPI write/read error
-70	ERR_DATA_WRITE_ERROR	Data write error
-71	ERR_DATA_READ_ERROR	Data read error
-72	ERR_TIMEOUT	Operation timeout
-73	ERR_DATA_CRC_FAILED	Data CRC failed

Appendix B: Exported Symbols

USB2ANY.DLL is a 32-bit dynamic-linked library with all functions exported using the *__stdcall* calling convention. The 32-bit versions of Visual C++ define *PASCAL* as *__stdcall*, which may cause some confusion.

In older 16-bit versions of Visual C++, *PASCAL* names were simply undecorated function names in all uppercase letters. However, *__stdcall* in 32-bit versions of Visual C++ allow mixed-case names and decorate the name by prefixing it with an underscore (_) and appending an at-sign (@) followed by the required stack space (in bytes). For example, a function declared as:

```
int  __stdcall MyFunc (int x, int y)
```

would be decorated as:

```
_MyFunc@8
```

Note that *__stdcall* is like *PASCAL* in the sense that the stack is cleaned-up by the called function (in this case, by the USB2ANY function).

Exported Symbols (by API Function Name value)

API Function Name	Ordinal*	Exported Name
u2aADC_Control	19	_u2aADC_Control@24
u2aADC_ConvertAndRead	20	_u2aADC_ConvertAndRead@12
u2aClock_Control	21	_u2aClock_Control@12
u2aClose	22	_u2aClose@4
u2aDACs_Write	23	_u2aDACs_Write@16
u2aEnableDebugLogging	24	_u2aEnableDebugLogging@4
u2aEnableDeviceDetect	25	_u2aEnableDeviceDetect@8
u2aFEC_Control	26	_u2aFEC_Control@56
u2aFEC_CountAndRead	27	_u2aFEC_CountAndRead@4
u2aFindControllers	28	_u2aFindControllers@0
u2aFirmwareVersion_Read	29	_u2aFirmwareVersion_Read@12
u2aGetErrorList	36	_u2aGetErrorList@12
u2aGetSerialNumber	37	_u2aGetSerialNumber@8
u2aGetSerialNumberW	38	_u2aGetSerialNumberW@8
u2aGetStatusText	39	_u2aGetStatusText@12
u2aGPIO_ReadPort	30	_u2aGPIO_ReadPort@8
u2aGPIO_ReadState	31	_u2aGPIO_ReadState@12
u2aGPIO_SetPort	32	_u2aGPIO_SetPort@12
u2aGPIO_WriteControl	33	_u2aGPIO_WriteControl@56
u2aGPIO_WritePort	34	_u2aGPIO_WritePort@12
u2aGPIO_WriteState	35	_u2aGPIO_WriteState@56
u2aI2C_Control	40	_u2aI2C_Control@16
u2aI2C_InternalRead	41	_u2aI2C_InternalRead@24
u2aI2C_InternalWrite	42	_u2aI2C_InternalWrite@24
u2aI2C_MultiRegisterRead	43	_u2aI2C_MultiRegisterRead@20
u2aI2C_MultiRegisterWrite	44	_u2aI2C_MultiRegisterWrite@20
u2aI2C_RawRead	45	_u2aI2C_RawRead@16
u2aI2C_RawWrite	46	_u2aI2C_RawWrite@16

API Function Name	Ordinal*	Exported Name
u2aI2C_RegisterRead	47	_u2aI2C_RegisterRead@12
u2aI2C_RegisterWrite	48	_u2aI2C_RegisterWrite@16
u2aInterrupt_Control	49	_u2aInterrupt_Control@20
u2aIsUSB2ANYConnected	50	_u2aIsUSB2ANYConnected@0
u2aLED_WriteControl	51	_u2aLED_WriteControl@8
u2aMSP430_ByteRead	52	_u2aMSP430_ByteRead@8
u2aMSP430_ByteWrite	53	_u2aMSP430_ByteWrite@12
u2aMSP430_WordRead	54	_u2aMSP430_WordRead@8
u2aMSP430_WordWrite	55	_u2aMSP430_WordWrite@12
u2aOpen	56	_u2aOpen@4
u2aOpenW	57	_u2aOpenW@4
u2aPower_ReadStatus	59	_u2aPower_ReadStatus@4
u2aPower_WriteControl	60	_u2aPower_WriteControl@12
u2aPWM_Control	58	_u2aPWM_Control@32
u2aReadResponse	61	_u2aReadResponse@12
u2aReadResponseB64	62	_u2aReadResponseB64@12
u2aSetReceiveTimeout	66	_u2aSetReceiveTimeout@4
u2aSetResponseMode	67	_u2aSetResponseMode@4
u2aSPI_Control	63	_u2aSPI_Control@36
u2aSPI_WriteAndRead	64	_u2aSPI_WriteAndRead@12
u2aSPI_WriteAndRead_P	65	_u2aSPI_WriteAndRead_P@12
u2aStatus_GetErrorCode	68	_u2aStatus_GetErrorCode@4
u2aStatus_GetText	69	_u2aStatus_GetText@12
u2aStatus_GetTextW	70	_u2aStatus_GetTextW@12
u2aUART_Control	71	_u2aUART_Control@24
u2aUART_Read	72	_u2aUART_Read@12
u2aUART_Write	73	_u2aUART_Write@12

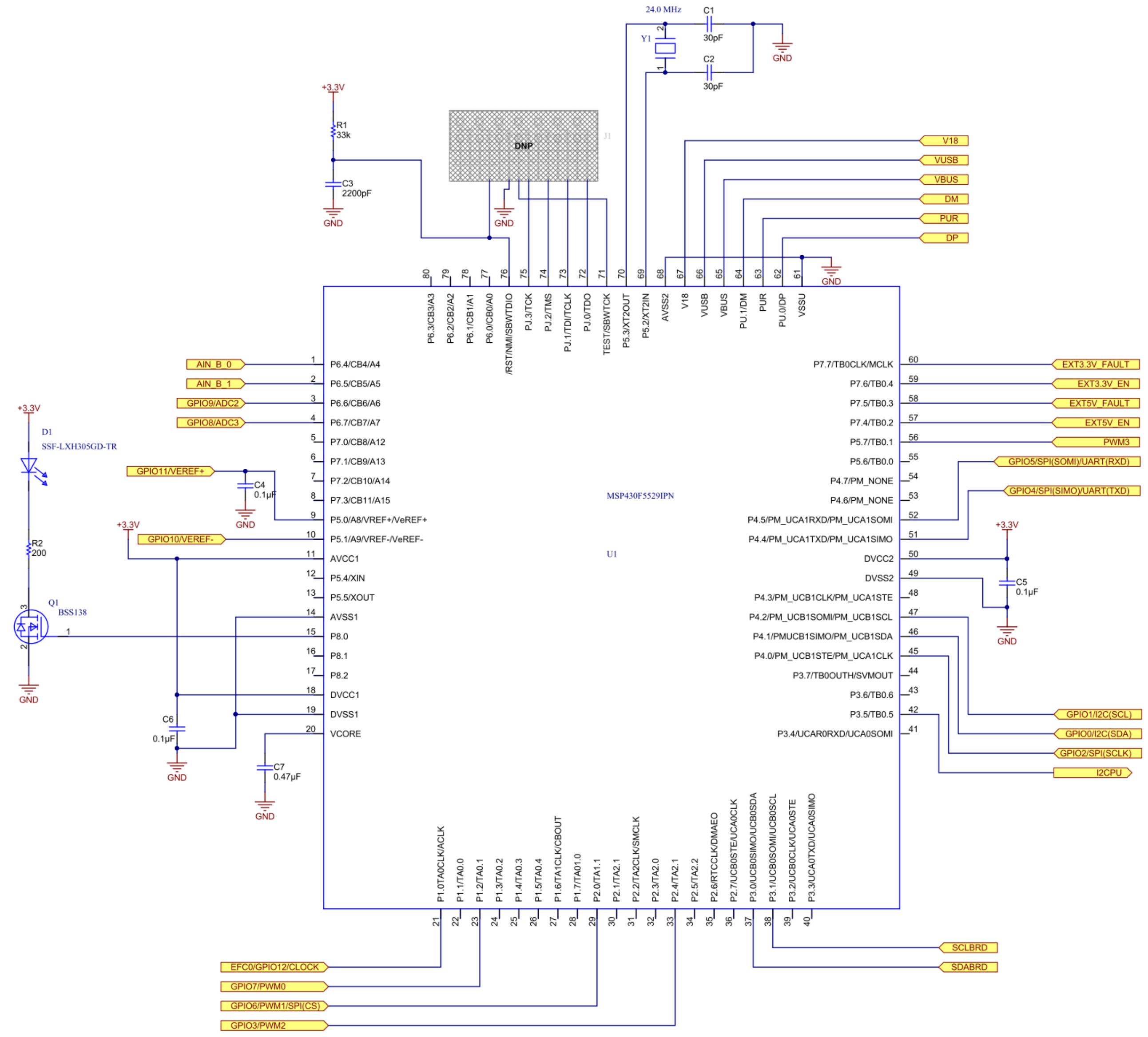
*** Note:** It is recommended that you use the exported name, instead of the ordinal, because the ordinals may change in subsequent releases of the API library.

Appendix C: Visual Basic Interface (VB6)

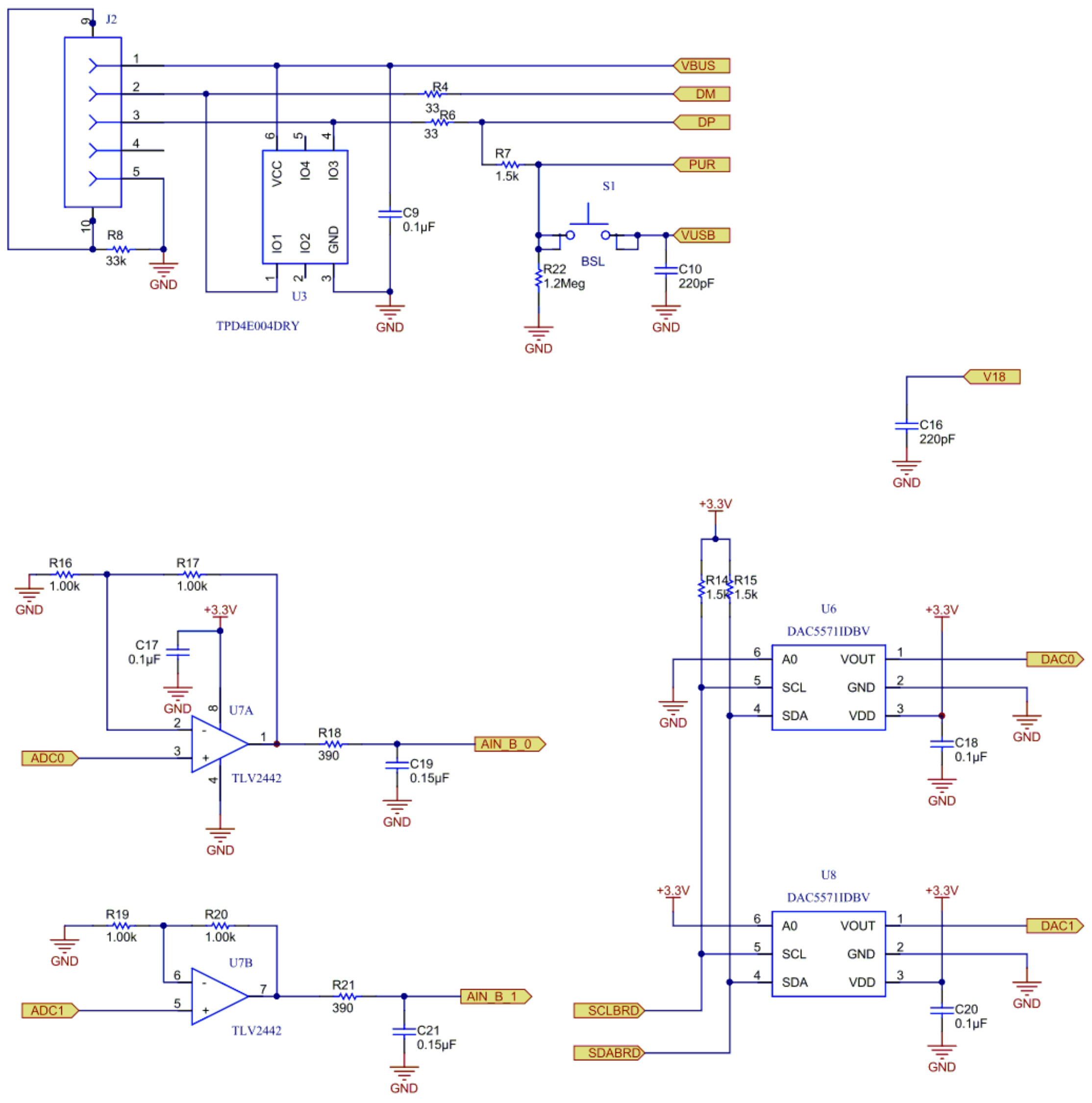
```
'
' Declarations for USB2ANY.DLL subroutines and functions
'
Public Declare Function u2aFindControllers Lib "USB2ANY" Alias "_u2aFindControllers@0" () As Long
Public Declare Function u2aGetSerialNumber Lib "USB2ANY" Alias "_u2aGetSerialNumber@8" (ByVal index As Long, _
    ByVal SerialNumber As String) As Long
Public Declare Function u2aOpen Lib "USB2ANY" Alias "_u2aOpen@4" (ByVal SerialNumber As String) As Long
Public Declare Function u2aGetStatusText Lib "USB2ANY" Alias "_u2aGetStatusText@12" (ByVal code As Long, _
    ByVal buffer As String, ByVal bufsize As Long) As String
 Public Declare Function u2aSetReceiveTimeout Lib "USB2ANY" Alias "_u2aSetReceiveTimeout@4" (ByVal milliseconds As Long) As Long

'//////////////////////////////////////////////////////////////////////////////////////////////////////
'//The following API functions require a handle parameter. They MUST be called with a valid
'//handle, obtained from a successful call to the u2aOpen() function.
'//////////////////////////////////////////////////////////////////////////////////////////////////////

'//
'// Controller Functions
'//

Public Declare Function u2aClose Lib "USB2ANY" Alias "_u2aClose @4" (ByVal handle As Long) As Long
Public Declare Function u2aReadResponse Lib "USB2ANY" Alias "_u2aReadResponse@12" (ByVal handle As Long, ByVal pBuffer As String, _
    ByVal dwBufferSize As Long) As Long

'//
'// I2C Functions
'//

Public Declare Function u2aI2C_Control Lib "USB2ANY" Alias "_u2aI2C_Control@20" (ByVal handle As Long, ByVal Speed As Long, _
    ByVal AddressLength As Long, ByVal PullUps As Long) As Long
Public Declare Function u2aI2C_RegisterRead Lib "USB2ANY" Alias "_u2aI2C_RegisterRead@12" (ByVal handle As Long, _
    ByVal I2C_Address As Long, ByVal RegisterAddress As Long) As Long
Public Declare Function u2aI2C_RegisterWrite Lib "USB2ANY" Alias "_u2aI2C_RegisterWrite@16" (ByVal handle As Long, _
    ByVal I2C_Address As Long, ByVal RegisterAddress As Long, ByVal Value As Long) As Long
Public Declare Function u2aI2C_MultiRegisterRead Lib "USB2ANY" Alias "_u2aI2C_MultiRegisterRead@20" (ByVal handle As Long, _
    ByVal I2C_Address As Long, ByVal StartingRegisterAddress As Long, ByVal nBytes As Long, ByVal Data[] As Long) As Long
Public Declare Function u2aI2C_MultiRegisterWrite Lib "USB2ANY" Alias "_u2aI2C_MultiRegisterWrite@20" (ByVal handle As Long, _
    ByVal I2C_Address As Long, ByVal StartingRegisterAddress As Long, ByVal nBytes As Long, ByVal Data[] As Long) As Long
Public Declare Function u2aI2C_InternalRead Lib "USB2ANY" Alias "_u2aI2C_InternalRead@24" (ByVal handle As Long, _
    ByVal I2C_Slave_Address As Long, ByVal InternalAddress As Long, ByVal nIntBytes As Long, ByVal nBytes As Long, _
    ByVal Data[] As Long) As Long
Public Declare Function u2aI2C_InternalWrite Lib "USB2ANY" Alias "_u2aI2C_InternalWrite@24" (ByVal handle As Long, _
    ByVal I2C_Slave_Address As Long, ByVal InternalAddress As Long, ByVal nIntBytes As Long, ByVal nBytes As Long, _
    ByVal Data[] As Long) As Long
Public Declare Function u2aI2C_RawRead Lib "USB2ANY" Alias "_u2aI2C_RawRead@16" (ByVal handle As Long, _
    ByVal I2C_Address As Long, ByVal nBytes As Long, ByVal Data[] As Long) As Long
Public Declare Function u2aI2C_RawWrite Lib "USB2ANY" Alias "_u2aI2C_RawWrite@16" (ByVal handle As Long, ByVal I2C_Address As Long, _
    ByVal nBytes As Long, ByVal Data[] As Long) As Long

'//
'// SPI Functions
'//

Public Declare Function u2aSPI_Control Lib "USB2ANY" Alias "_u2aSPI_Control@36" (ByVal handle As Long, _
    ByVal _SPI_ClockPhase As Long, _
    ByVal _SPI_ClockPolarity As Long, _
    ByVal _SPI_BitDirection As Long, _
    ByVal _SPI_CharacterLength As Long, _
    ByVal _SPI_LatchType As Long, _
    ByVal _SPI_LatchPolarity As Long, _
    ByVal _DividerHigh As Long, _
    ByVal _DividerLow As Long RPAREN As Long
Public Declare Function u2aSPI_WriteAndRead Lib "USB2ANY" Alias "_u2aSPI_WriteAndRead@12" (ByVal handle As Long, _
        ByVal nBytes As Long,  ByVal Data[] As Long) As Long

'//
'// ADC, DAC, and PWM Functions
'//

Public Declare Function u2aADC_Control Lib "USB2ANY" Alias "_u2aADC_Control@24" (ByVal handle As Long, _
        ByVal ADC0 As Long, _
        ByVal ADC1 As Long, _
        ByVal ADC2 As Long, _
        ByVal ADC3 As Long, _
        ByVal VREF As Long) As Long
Public Declare Function u2aADC_ConvertAndRead Lib "USB2ANY" Alias "_u2aADC_ConvertAndRead@12" (ByVal handle As Long, _
        ByVal nBytes As Long, ByVal Data[] As Long) As Long
Public Declare Function u2aDACs_Write Lib "USB2ANY" Alias "_u2aDACs_Write@20" (ByVal handle As Long, ByVal _DACs_WhichDAC As Long, _
        ByVal _DACs_OperatingMode As Long, ByVal _Value As Long) As Long
Public Declare Function u2aPWM_Control Lib "USB2ANY" Alias "_u2aPWM_Control@32" (ByVal handle As Long, _
        ByVal _ModeControl As Long, _
        ByVal _PWM_WhichPWM As Long, _
        ByVal _InputDivider As Long, _
        ByVal _CompareRegister0 As Long, _
        ByVal _OutputMode1 As Long, _
        ByVal _CompareRegister1 As Long, _
        ByVal _InputDividerEX As Long) As Long

'//
'// UART Functions
'//

Public Declare Function u2aUART_Control Lib "USB2ANY" Alias "_u2aUART_Control@24" (ByVal handle As Long, _
        ByVal _UART_BaudRate As Long, _
        ByVal _UART_Parity As Long, _
        ByVal _UART_BitDirection As Long, _
        ByVal _UART_CharacterLength As Long, _
        ByVal _UART_StopBits As Long) As Long
Public Declare Function u2aUART_Write Lib "USB2ANY" Alias "_u2aUART_Write@12" (ByVal handle As Long , _
        ByVal nBytes As Long, ByVal Data[] As Long) As Long
Public Declare Function u2aUART_Read Lib "USB2ANY" Alias "_u2aUART_Read@12" (ByVal handle As Long, _
        ByVal nBytes As Long, ByVal Data[] As Long) As Long

'//
'// GPIO Functions
'//

Public Declare Function u2aGPIO_WriteControl Lib "USB2ANY" Alias "_u2aGPIO_WriteControl@56" (ByVal handle As Long, _
        ByVal GPIO0 As Long, _
        ByVal GPIO1 As Long, _
        ByVal GPIO2 As Long, _
        ByVal GPIO3 As Long, _
        ByVal GPIO4 As Long, _
        ByVal GPIO5 As Long, _
        ByVal GPIO6 As Long, _
        ByVal GPIO7 As Long, _
        ByVal GPIO8 As Long, _
        ByVal GPIO9 As Long, _
        ByVal GPIO10 As Long, _
        ByVal GPIO11 As Long, _
        ByVal GPIO12 As Long) As Long
Public Declare Function u2aGPIO_WriteState Lib "USB2ANY" Alias "_u2aGPIO_WriteState@56" (ByVal handle As Long, _
        ByVal GPIO0 As Long, _
        ByVal GPIO1 As Long, _
        ByVal GPIO2 As Long, _
        ByVal GPIO3 As Long, _
        ByVal GPIO4 As Long, _
        ByVal GPIO5 As Long, _
        ByVal GPIO6 As Long, _
        ByVal GPIO7 As Long, _
        ByVal GPIO8 As Long, _
        ByVal GPIO9 As Long, _
        ByVal GPIO10 As Long, _
        ByVal GPIO11 As Long, _
        ByVal GPIO12 As Long) As Long
Public Declare Function u2aGPIO_ReadState Lib "USB2ANY" Alias "_u2aGPIO_ReadState@12" (ByVal handle As Long , _
        ByVal nBytes As Long, ByVal Data[] As Long) As Long
Public Declare Function u2aGPIO_SetPort  Lib "USB2ANY" Alias "_u2aGPIO_SetPort@12"  (ByVal handle As Long, _
        ByVal GPIO_Port As Long, ByVal function As Long) As Long
Public Declare Function u2aGPIO_WritePort  Lib "USB2ANY" Alias "_u2aGPIO_WritePort@12"  (ByVal handle As Long, _
        ByVal GPIO_Port As Long, ByVal state As Long) As Long
Public Declare Function u2aGPIO_ReadPort  Lib "USB2ANY" Alias "_u2aGPIO_ReadPort@8"  (ByVal handle As Long, _
        ByVal GPIO_Port As Long) As Long

'//
'// Memory Functions
'//

Public Declare Function u2aMSP430_ByteRead Lib "USB2ANY" Alias "_u2aMSP430_ByteRead@8" (ByVal handle As Long , _
        ByVal Address As Long) As Long
Public Declare Function u2aMSP430_ByteWrite Lib "USB2ANY" Alias "_u2aMSP430_ByteWrite@12" (ByVal handle As Long, _
        ByVal Address As Long, ByVal Value As Long) As Long
Public Declare Function u2aMSP430_WordRead Lib "USB2ANY" Alias "_u2aMSP430_WordRead@8" (ByVal handle As Long, _
        ByVal Address As Long) As Long
Public Declare Function u2aMSP430_WordWrite Lib "USB2ANY" Alias "_u2aMSP430_WordWrite@12" (ByVal handle As Long, _
        ByVal Address As Long, ByVal Value As Long) As Long
Public Declare Function u2aMSP430_MemoryRead Lib "USB2ANY" Alias "_u2aMSP430_MemoryRead@16" (ByVal handle As Long, _
        ByVal Address As Long, ByVal nBytes As Long, ByVal *Data As Long) As Long
Public Declare Function u2aMSP430_MemoryWrite Lib "USB2ANY" Alias "_u2aMSP430_MemoryWrite@16" (ByVal handle As Long, _
        ByVal Address As Long, ByVal nBytes As Long, ByVal *Data As Long) As Long

'//
'// Miscellaneous Functions
'//

Public Declare Function u2aFirmwareVersion_Read Lib "USB2ANY" Alias "_u2aFirmwareVersion_Read@12" (ByVal handle As Long , _
        ByVal szVersion As String, ByVal bufsize As Long) As Long
Public Declare Function u2aPower_WriteControl Lib "USB2ANY" Alias "_u2aPower_WriteControl@12" (ByVal handle As Long, _
        ByVal _Power_3V3 As Long, ByVal _Power_5V0 As Long) As Long
Public Declare Function u2aPower_ReadStatus Lib "USB2ANY" Alias "_u2aPower_ReadStatus@4" (ByVal handle As Long) As Long
Public Declare Function u2aLED_WriteControl Lib "USB2ANY" Alias "_u2aLED_WriteControl@8" (ByVal handle As Long , _
        ByVal _LED As Long) As Long
Public Declare Function u2aClock_Control  Lib "USB2ANY" Alias "_u2aClock_Control@12" (ByVal handle As Long, _
        ByVal _ClockDivider1 As Long, ByVal _ClockDivider2 As Long) As Long
```

Appendix D: SMBus Details

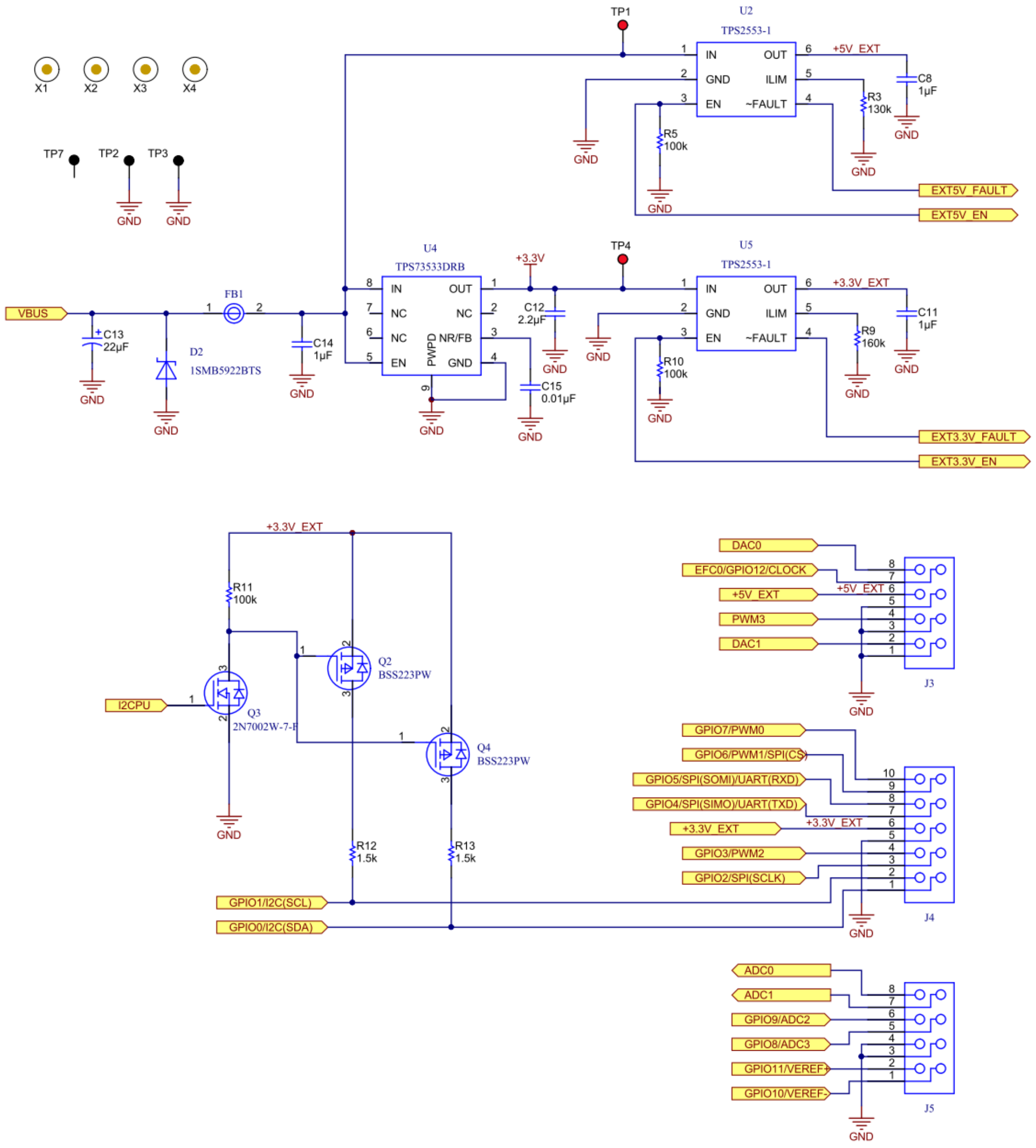
Appendix E: USB2ANY Schematic



USB2ANY Schematic – Page 1



USB2ANY Schematic – Page 2

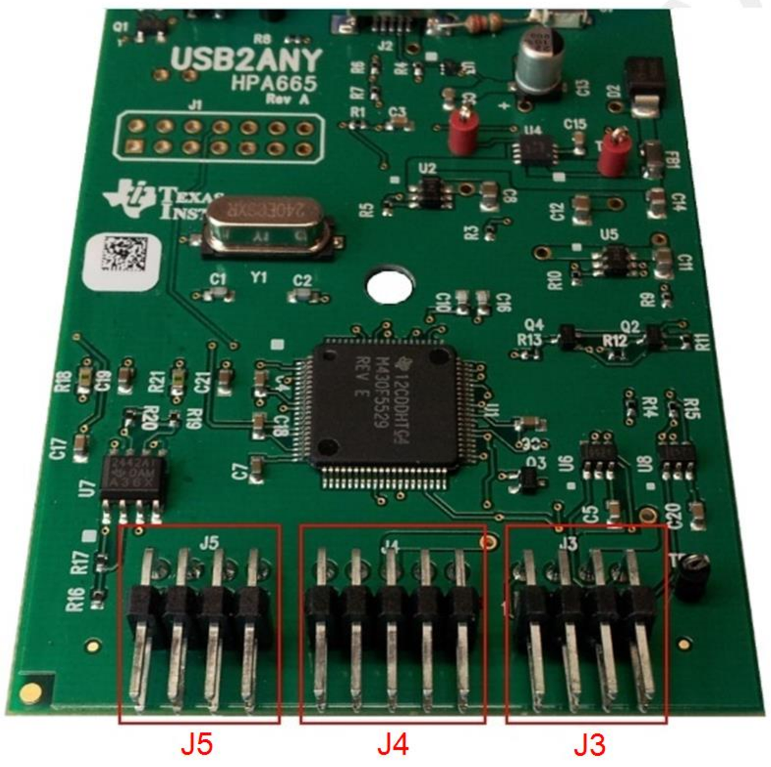


USB2ANY Schematic – Page 3

Appendix F: USB2ANY Cable Connections

The USB2ANY has four interface connectors: one USB 2.0 connector (J2) and three I/O connectors (J3, J4, and J5). The USB connector is a standard ‘A’ type mini USB receptacle. The I/O connectors are standard dual-row, .1” center, pin headers.

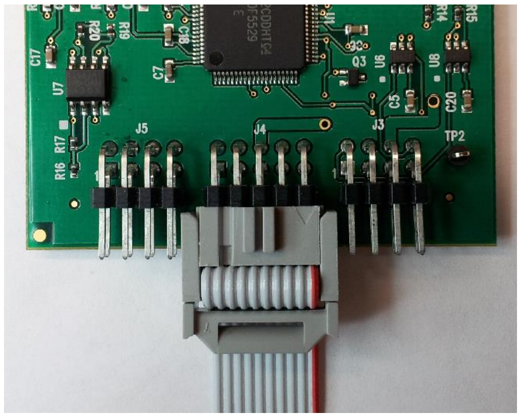
I/O connectors J3 and J5 are 8-pin type and J4 is a 10-pin type. They are configured such that they will accept either individual cable connections or a single 30-pin connection.



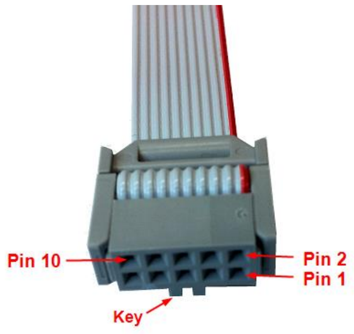
The standard USB2ANY Kit (HPA665-001) includes both a 10-pin cable and a 30-pin cable. The 10-pin cable is intended to be connected to J4. This single connection will supply the needs for many users as it provides access to the following interfaces and signals:

- **I^{2}C**: SDA and SCL
- **SPI**: SCLK, MOSI, MISO, and CS
- **UART**: RX and TX
- **Power**: +3.3V External
- **GPIO**: GPIO0, GPIO1, GPIO2, GPIO3, GPIO4, GPIO5, GPIO6, and GPIO7
- **PWM**: PWM0, PWM1, and PWM2
- **OneWire**: OW1, OW2, and OW3
- **µWire**: SCLK, MOSI, MISO, and CS
- **RFFE**: SCLK and SDATA
- **Interrupts**: INT0, INT1, and INT2

The 10-pin cable is about 6 inches in length and has a keyed female 10-pin IDC connector on each end. The cable should be connected to the USB2ANY board as shown in the following picture (note that the key must be facing up, away from the board):



The opposite end of the cable is intended to be connected to the EVM (Evaluation Module) or other target device. The red stripe on the cable indicates Pin 1.

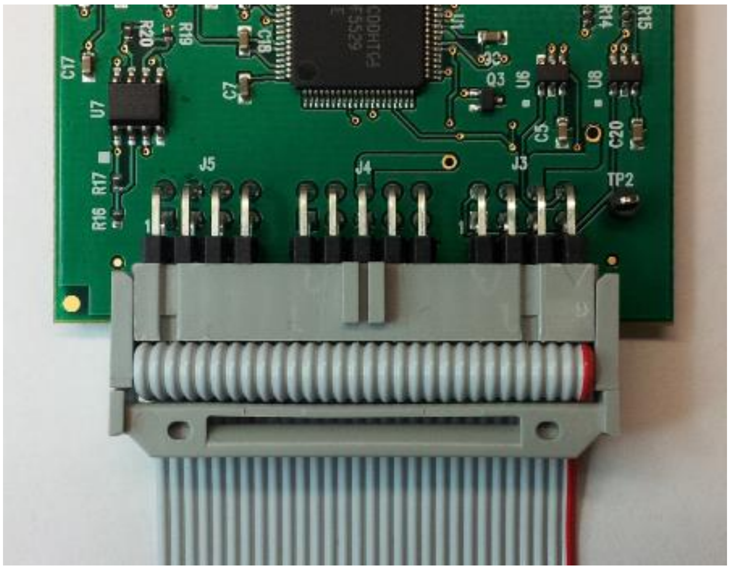


Schematic Pin #	Cable Pin #	Signals Available
J4-10	1	GPIO7, PWM0, INT2, OW2, OW5
J4-9	2	GPIO6, PWM1, RFFE:SCLK, SPI:CS, INT1, µWIRE:CS, OW1
J4-8	3	GPIO5, SPI:SOMI, UART:RXD, µWIRE:SOMI
J4-7	4	GPIO4, SPI:SIMO, UART:TXD, µWIRE:SIMO
J4-6	5	+3.3VEXT
J4-5	6	GND
J4-4	7	GPIO3, PWM2, RFFE:SDATA, INT0
J4-3	8	GPIO2, ES:DOUT, SPI:SCLK, µWIRE:SCLK
J4-2	9	GPIO1, I2C:SCL, OW3
J4-1	10	GPIO0, I2C:SDA, OW4

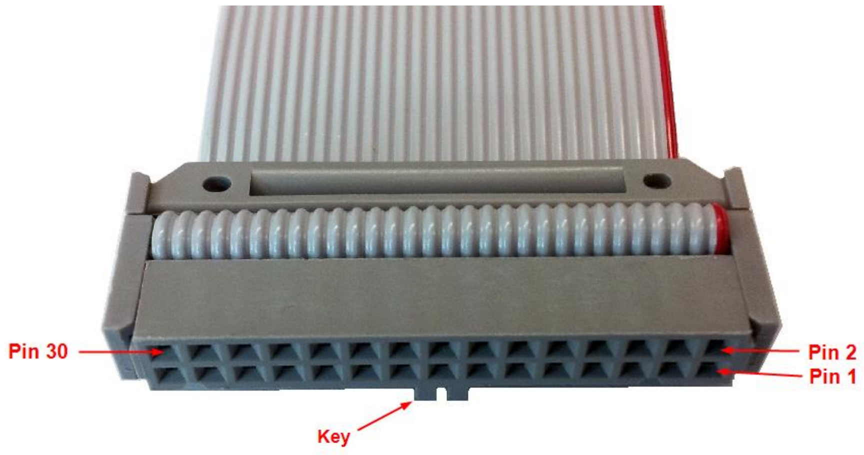
10-pin Cable Pinouts

IMPORTANT NOTE: The pin numbers for J4 on the schematics are for reference only and ***do not*** correspond to the pin numbers for cable connections.

The 30-pin cable is also about 6 inches in length and has a keyed female 30-pin IDC connector on each end. This cable provides access to all available signals. The cable should be connected to the USB2ANY board as shown in the following picture (note that the key must be facing up, away from the board):



The opposite end of the cable is intended to be connected to the EVM (Evaluation Module) or other target device. The red stripe on the cable indicates Pin 1.



Schematic Pin #	Cable Pin #	Signals Available
J3-8	1	DAC0
J3-7	2	PIO12, CLOCK, EFC0, INT3
J3-6	3	+5VEXT
J3-5	4	GND
J3-4	5	PWM3
J3-3	6	GND
J3-2	7	DAC1
J3-1	8	GND
	9	(unused)
	10	(unused)
J4-10	11	GPIO7, PWM0, INT2, OW2, OW5
J4-9	12	GPIO6, PWM1, RFFE:SCLK, SPI:CS, INT1, µWIRE:CS, OW1
J4-8	13	GPIO5, SPI:SOMI, UART:RXD, µWIRE:SOMI
J4-7	14	GPIO4, SPI:SIMO, UART:TXD, µWIRE:SIMO
J4-6	15	+3.3VEXT
J4-5	16	GND
J4-4	17	GPIO3, PWM2, RFFE:SDATA, INT0
J4-3	18	GPIO2, ES:DOUT, SPI:SCLK, µWIRE:SCLK
J4-2	19	GPIO1, I2C:SCL, OW3
J4-1	20	GPIO0, I2C:SDA, OW4
	21	(unused)
	22	(unused)
J5-8	23	ADC0
J5-7	24	ADC1
J5-6	25	GPIO9, ADC2, ES:AIN
J5-5	26	GPIO8, ADC3
J5-4	27	GND
J5-3	28	GND
J5-2	29	GPIO11, VREF+
J5-1	30	GPIO10, VREF-

30-pin Cable Pinouts

IMPORTANT NOTE: The pin numbers for J3, J4, and J5 on the schematics are for reference only and ***do not*** correspond to the pin numbers for cable connections.