Relevant Devices
This application note applies to the following devices:
C8051F32x, C8051F34x, C8051F38x, C8051T32x, C8051T62x, CP2101, CP2102, CP2103, CP2104, CP2105, CP2108, CP2109

1. Introduction
The Silicon Laboratories USBXpress® Development Kit provides a complete host and device software solution for interfacing Silicon Laboratories C8051F32x, C8051F34x, C8051F38x, C8051T32x, and C8051T62x USB MCUs and CP210x communication bridges to the Universal Serial Bus (USB). No USB protocol or host device driver expertise is required. Instead, a simple, high-level Application Program Interface (API) for both the host software and device firmware is used to provide complete USB connectivity.

The USBXpress Development Kit includes Windows device drivers, Windows device driver installer, host interface function library (host API) provided in the form of a Windows Dynamic Link Library (DLL), and device firmware interface function library (C8051F32x, C8051F34x, C8051F38x, C8051T32x, and C8051T62x devices only).

Figure 1. USBXpress Data Flow

---

**AN169**

**USBXpress® PROGRAMMER’S GUIDE**

**Relevant Devices**
This application note applies to the following devices:
C8051F32x, C8051F34x, C8051F38x, C8051T32x, C8051T62x, CP2101, CP2102, CP2103, CP2104, CP2105, CP2108, CP2109

**1. Introduction**
The Silicon Laboratories USBXpress® Development Kit provides a complete host and device software solution for interfacing Silicon Laboratories C8051F32x, C8051F34x, C8051F38x, C8051T32x, and C8051T62x USB MCUs and CP210x communication bridges to the Universal Serial Bus (USB). No USB protocol or host device driver expertise is required. Instead, a simple, high-level Application Program Interface (API) for both the host software and device firmware is used to provide complete USB connectivity.

The USBXpress Development Kit includes Windows device drivers, Windows device driver installer, host interface function library (host API) provided in the form of a Windows Dynamic Link Library (DLL), and device firmware interface function library (C8051F32x, C8051F34x, C8051F38x, C8051T32x, and C8051T62x devices only).

Figure 1. USBXpress Data Flow
2. USBXpress Versions

The USBXpress v4.0 and above driver incorporates Microsoft's WinUSB driver rather than the proprietary driver of v3.x, and both the underlying driver and the DLL have changed. To use USBXpress v4.0, install the latest driver from the website (www.silabs.com/interface-software) and upgrade the DLL.

The API to the DLL remains unchanged, but there is one new API call added to the library: `SI_CancelIo()`. This API call replaces `CancelIo()` and should be used if the software calls `SI_Read()` or `SI_Write()` in overlapped mode and the software needs to cancel I/O operations. The `SI_CancelIo()` function cancels read and write requests in both the DLL buffer and the driver buffer, whereas `CancelIo()` only cancels requests in the driver buffer, leaving outstanding requests in the DLL.

The AN220 Driver Customization software provides an option to customize either the v4.0 and above driver or the v3.x driver for legacy applications. Customizations of the v4.0 and above WinUSB driver will still require recertification through Microsoft. The AN220 software can be found here: www.silabs.com/interface-appnotes.
3. Host API Functions

The host API is provided in the form of a Windows Dynamic Link Library (DLL). The host interface DLL communicates with the USB device via the provided device driver and the operating system's USB stack. The following is a list of the host API functions available:

- **SI_GetNumDevices()** - Returns the number of devices connected
- **SI_GetProductString()** - Returns a descriptor for a device
- **SI_Open()** - Opens a device and returns a handle
- **SI_Close()** - Cancels pending IO and closes a device
- **SI_Read()** - Reads a block of data from a device
- **SI_Write()** - Writes a block of data to a device
- **SI_CancelIo()** - Cancels outstanding overlapped IO
- **SI_FlushBuffers()** - Flushes the TX and RX buffers for a device
- **SI_SetTimeouts()** - Sets read and write block timeouts
- **SI_GetTimeouts()** - Gets read and write block timeouts
- **SI_CheckRXQueue()** - Returns the number of bytes in a device's RX queue
- **SI_SetBaudRate()** - Sets the specified CP210x Baud Rate
- **SI_SetBaudDivisor()** - Sets the specified CP210x Baud Divisor Value
- **SI_SetLineControl()** - Sets the CP210x device Line Control
- **SI_SetFlowControl()** - Sets the CP210x device Flow Control
- **SI_GetModemStatus()** - Gets the CP210x device Modem Status
- **SI_SetBreak()** - Sets the Break State for CP210x device.
- **SI_ReadLatch()** - Gets the port latch value from a CP2103 device
- **SI_WriteLatch()** - Sets the port latch value to a CP2103 device
- **SI_GetPartNumber()** - Gets the CP210x device part number
- **SI_DeviceIOControl()** - Allows sending low-level commands to the device driver
- **SI_GetDLLVersion()** - Gets the version of the DLL currently in use
- **SI_GetDriverVersion()** - Gets the version of the USBXpress driver

In general, the user initiates communication with the target USB device(s) by making a call to **SI_GetNumDevices**. This call will return the number of target devices. This number is then used as a range when calling **SI_GetProductString** to build a list of device serial numbers or product description strings.

To access a device, it must first be opened by a call to **SI_Open** using an index determined from the call to **SI_GetNumDevices**. The **SI_Open** function will return a handle to the device that is used in all subsequent accesses. Data I/O is performed using the **SI_Write** and **SI_Read** functions. When I/O operations are complete, the device is closed by a call to **SI_Close**.

Additional functions are provided to flush the transmit and receive buffers (**SI_NFlushBuffers**), set receive and transmit timeouts (**SI_SetTimeouts**), check the receive buffer's status (**SI_CheckRXQueue**), and miscellaneous device control (**SI_DeviceIOControl**).

For CP210x devices, functions are available to set the baud rate (**SI_SetBaudRate**); set the baud divisor (**SI_SetBaudDivisor**); adjust the line control settings such as word length, stop bits, and parity (**SI_SetLineControl**); set hardware handshaking, software handshaking, and modem control signals (**SI_SetFlowControl**); and get modem status (**SI_GetModemStatus**). Additional functions are available for CP2103/4/5/8 devices to get (**SI_ReadLatch**) and set (**SI_WriteLatch**) the values of the additional GPIO pins available on the device. In order to differentiate between CP210x devices, a function (**SI_GetPartNumber**) has been provided to return the part number.

Each of these functions are described in detail in the following sections. Type definitions and constants are defined in "Appendix D—Definitions from C++ header file SiUSBXp.h".
3.1. SI_GetNumDevices

Description: This function returns the number of devices connected to the host.

Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3, CP2101/2/3/4/5/8/9/

Prototype: SI_STATUS SI_GetNumDevices (LPDWORD NumDevices)

Parameters: 1. NumDevices—Address of a DWORD variable that will contain the number of devices connected on return.

Return Value: SI_STATUS = SI_SUCCESS or SI_DEVICE_NOT_FOUND or SI_INVALID_PARAMETER

3.2. SI_GetProductString

Description: This function returns a null terminated serial number (S/N) string or product description string for the device specified by an index passed in DeviceNum. The index for the first device is 0 and the last device is the value returned by SI_GetNumDevices – 1.


Prototype: SI_STATUS SI_GetProductString (DWORD DeviceNum, LPVOID DeviceString, DWORD Options)

Parameters: 1. DeviceNum—Index of the device for which the product description string or serial number string is desired.
   2. DeviceString—Variable of type SI_DEVICE_STRING which will contain a NULL terminated device descriptor or serial number string on return.
   3. Options—DWORD containing flags to determine if DeviceString contains a serial number, product description, Vendor ID, or Product ID string. See "Appendix D—Definitions from C++ header file SiUSBXp.h" for flags.

Return Value: SI_STATUS = SI_SUCCESS or SI_DEVICE_NOT_FOUND or SI_INVALID_PARAMETER
3.3. **SI_Open**

**Description:** Opens a device (using device number as returned by `SI_GetNumDevices`) and returns a handle which will be used for subsequent accesses.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3, CP2101/2/3/4/5/8/9

**Prototype:**

```
SI_STATUS SI_Open (DWORD DeviceNum, HANDLE *Handle)
```

**Parameters:**

1. `DeviceNum`—Device index. 0 for first device, 1 for 2nd, etc.
2. `Handle`—Pointer to a variable where the handle to the device will be stored. This handle will be used by all subsequent accesses to the device.

**Return Value:** `SI_STATUS` = `SI_SUCCESS` or
- `SI_DEVICE_NOT_FOUND` or
- `SI_INVALID_PARAMETER` or
- `SI_GLOBAL_DATA_ERROR`

3.4. **SI_Close**

**Description:** Closes an open device using the handle provided by `SI_Open` and sets the handle to `INVALID_HANDLE_VALUE`.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3, CP2101/2/3/4/5/8/9

**Prototype:**

```
SI_STATUS SI_Close (HANDLE Handle)
```

**Parameters:**

1. `Handle`—Handle to the device to close as returned by `SI_Open`.

**Return Value:** `SI_STATUS` = `SI_SUCCESS` or
- `SI_INVALID_HANDLE` or
- `SI_SYSTEM_ERROR_CODE` or
- `SI_GLOBAL_DATA_ERROR`
3.5. **SI_Read**

**Description:** Reads the available number of bytes into the supplied buffer and retrieves the number of bytes that were read (this can be less than the number of bytes requested). This function returns synchronously if the overlapped object is set to NULL (this happens by default) but will not block system execution. If an initialized OVERLAPPED object is passed then the function returns immediately. If the read completed then the status will be SI_SUCCESS but if I/O is still pending then it will return STATUS_IO_PENDING. If STATUS_IO_PENDING is returned, the OVERLAPPED object can then be waited on using WaitForSingleObject(), and retrieve data or cancel using GetOverlappedResult() (as documented on MSDN by Microsoft) or SI_CancelIo() respectively. This functionality allows for multiple reads to be issued and waited on at a time. If any data is available when SI_Read is called it will return so check NumBytesReturned to determine if all requested data was returned. To make sure that SI_Read returns the requested number of bytes use SI_CheckRXQueue() described in Section "3.11. SI_CheckRXQueue" on page 9.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3, CP2101/2/3/4/5/8/9

**Prototype:**

```
SI_STATUS SI_Read (HANDLE Handle, LPVOID Buffer, DWORD NumBytesToRead, DWORD *NumBytesReturned, OVERLAPPED* o = NULL)
```

**Parameters:**

1. *Handle*—Handle to the device to read as returned by SI_Open.
2. *Buffer*—Address of a character buffer to be filled with read data.
3. *NumBytesToRead*—Number of bytes to read from the device into the buffer (0–64 kB).
4. *NumBytesReturned*—Address of a DWORD which will contain the number of bytes actually read into the buffer on return.
5. *(Optional)*—Address of an initialized OVERLAPPED object that can be used for asynchronous reads.

**Return Value:**

```
SI_STATUS = SI_SUCCESS or SI_READ_ERROR or SI_INVALID_PARAMETER or SI_INVALID_HANDLE or SI_SI_READ_TIMED_OUT or SI_IO_PENDING or SI_SYSTEM_ERROR_CODE or SI_INVALID_REQUEST_LENGTH or SIDEVICE_IO_FAILED
```
3.6. SI_Write

Description: Writes the specified number of bytes from the supplied buffer to the device. This function returns synchronously if the overlapped object is set to NULL (this happens by default) but will not block system execution. An initialized OVERLAPPED object may be specified and waited on just as described in the description for SI_Read(), Section "3.5. SI_Read" on page 6.


Prototype: `SI_STATUS SI_Write (HANDLE Handle, LPVOID Buffer, DWORD NumBytesToWrite, DWORD *NumBytesWritten, OVERLAPPED* o = NULL)`

Parameters:
1. **Handle**—Handle to the device to write as returned by `SI_Open`.
2. **Buffer**—Address of a character buffer of data to be sent to the device.
3. **NumBytesToWrite**—Number of bytes to write to the device (0–4096 bytes).
4. **NumBytesWritten**—Address of a DWORD which will contain the number of bytes actually written to the device.
5. (Optional) —Address of an initialized OVERLAPPED object that can be used for asynchronous writes.

Return Value: `SI_STATUS = SI_SUCCESS or SI_WRITE_ERROR or SI_INVALID_REQUEST_LENGTH or SI_INVALID_PARAMETER or SI_INVALID_HANDLE or SI_WRITE_TIMED_OUT or SI_IO_PENDING or SI_SYSTEM_ERROR_CODE or SI_DEVICE_IO_FAILED`  

3.7. SI_CancelIo

Description: Cancels any pending IO on a device opened with an OVERLAPPED object.


Prototype: `SI_STATUS SI_CancelIo (HANDLE Handle)`

Parameters: 1. **Handle**—Handle to the device to write as returned by `SI_Open`.

Return Value: `SI_STATUS = SI_SUCCESS or SI_INVALID_HANDLE or SI_SYSTEM_ERROR_CODE or SI_GLOBAL_DATA_ERROR`
3.8. SI_FlushBuffers

Description: On USB MCU devices, this function flushes both the receive buffer in the USBXpress device driver and the transmit buffer in the device. Note: Parameter 2 and 3 have no effect and any value can be passed when used with USB MCU devices.

On CP210x devices, this function operates in accordance with parameters 2 and 3. If parameter 2 (FlushTransmit) is non-zero, the CP210x device’s UART transmit buffer is flushed. If parameter 3 (FlushReceive) is non-zero, the CP210x device’s UART receive buffer is flushed. If parameters 2 and 3 are both non-zero, then both the CP210x device UART transmit buffer and UART receive buffer are flushed.


Prototype: SI_STATUS SI_FlushBuffers (HANDLE Handle, BYTE FlushTransmit, BYTE FlushReceive)

Parameters: 1. Handle—Handle to the device as returned by SI_Open.
2. FlushTransmit—Set to a non-zero value to flush the CP210x UART transmit buffer.
3. FlushReceive—Set to a non-zero value to flush the receive buffer.

Return Value: SI_STATUS = SI_SUCCESS or SI_INVALID_HANDLE or SI_SYSTEM_ERROR_CODE

3.9. SI_SetTimeouts

Description: Sets the read and write timeouts. Timeouts are used for SI_Read and SI_Write when called synchronously (OVERLAPPED* o is set to NULL). The default value for timeouts is INFINITE (0xFFFFFFFF), but they can be set to wait for any number of milliseconds between 0x0 and 0xFFFFFFFF.


Prototype: SI_STATUS SI_SetTimeouts (DWORD ReadTimeout, DWORD WriteTimeout)

Parameters: 1. ReadTimeout—SI_Read operation timeout (in milliseconds).
2. WriteTimeout—SI_Write operation timeout (in milliseconds).

Return Value: SI_STATUS = SI_SUCCESS or SI_DEVICE_IO_FAILED
3.10. SI_GetTimeouts

Description: Returns the current read and write timeouts. If a timeout value is 0xFFFFFFFF (INFINITE) it has been set to wait infinitely; otherwise the timeouts are specified in milliseconds.

Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3/2/4/5/8/9

Prototype: SI_STATUS SI_GetTimeouts (LPDWORD ReadTimeout, LPDWORD WriteTimeout)

Parameters: 1. ReadTimeout—SI_Read operation timeout (in milliseconds).
2. WriteTimeout—SI_Write operation timeout (in milliseconds).

Return Value: SI_STATUS = SI_SUCCESS or
SI_INVALID_PARAMETER or
SI_DEVICE_IO_FAILED

3.11. SI_CheckRXQueue

Description: Returns the number of bytes in the receive queue and a status value that indicates if an overrun (SI_QUEUE_OVERRUN) has occurred and if the RX queue is ready (SI_QUEUE_READY) for reading. Upon indication of an Overrun condition it is recommended that data transfer be stopped and all buffers be flushed using the SI_FlushBuffers command.

Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3/2/4/5/8/9

Prototype: SI_STATUS SI_CheckRXQueue (HANDLE Handle, LPDWORD NumBytesInQueue, LPDWORD QueueStatus)

Parameters: 1. Handle—Handle to the device as returned by SI_Open.
2. NumBytesInQueue—Address of a DWORD variable that contains the number of bytes currently in the receive queue on return.
3. QueueStatus—Address of a DWORD variable that contains the SI_RX_EMPTY (also SI_RX_NO_OVERRUN), SI_RX_OVERRUN, or SI_RX_READY flag.

Return Value: SI_STATUS = SI_SUCCESS or
SI_DEVICE_IO_FAILED or
SI_INVALID_HANDLE or
SI_INVALID_PARAMETER
3.12.  SI_SetBaudRate

Description:  Sets the Baud Rate. Refer to the device data sheet for a list of Baud Rates supported by the device.

Supported Devices:  CP2101/2/3/4/5/8/9

Prototype:  

\[
\text{SI_STATUS } \text{SI_SetBaudRate} \ (\text{HANDLE } \text{Handle}, \text{ DWORD } \text{dwBaudRate})
\]

Parameters:  1.  \text{Handle}—Handle to the device as returned by \text{SI_Open}.
2.  \text{dwBaudRate}—A DWORD value specifying the Baud Rate to set.

Return Value:  SI_STATUS = SI_SUCCESS or 
SI_INVALID_BAUDRATE or
SI_INVALID_HANDLE or
SI_INVALID_PARAMETER or
SI_DEVICE_IO_FAILED

3.13.  SI_SetBaudDivisor

Description:  Sets the Baud Rate directly by using a specific divisor value. This function is obsolete; use \text{SI_SetBaudRate} instead.

Supported Devices:  CP2101/2/3/4/5/8/9

Prototype:  

\[
\text{SI_STATUS } \text{SI_SetBaudDivisor} \ (\text{HANDLE } \text{Handle}, \text{ WORD } w\text{BaudDivisor})
\]

Parameters:  1.  \text{Handle}—Handle to the device as returned by \text{SI_Open}.
2.  \text{wBaudDivisor}—A WORD value specifying the Baud Divisor to set.

Return Value:  SI_STATUS = SI_SUCCESS or 
SI_INVALID_HANDLE or
SI_INVALID_PARAMETER or
SI_DEVICE_IO_FAILED
3.14. **SI_SetLineControl**

**Description:** Adjusts the line control settings: word length, stop bits, and parity. Refer to the device data sheet for valid line control settings.

**Supported Devices:** CP2101/2/3/4/5/8/9

**Prototype:**

```
SI_STATUS SI_SetLineControl (HANDLE Handle, WORD wLineControl)
```

**Parameters:**

1. **Handle**—Handle to the device as returned by `SI_Open`.
2. **wLineControl**—A WORD variable that contains the desired line control settings. Possible input settings are as follows:
   - **Bits 0–3** Number of Stop bits
     - 0: 1 stop bit;
     - 1: 1.5 stop bits;
     - 2: 2 stop bits
   - **Bits 4–7** Parity
     - 0: None
     - 1: Odd
     - 2: Even
     - 3: Mark
     - 4: Space
   - **Bits 8–15** Number of bits per word
     - 5, 6, 7, or 8

**Return Value:**

```
SI_STATUS = SI_SUCCESS or
            SI_DEVICE_IO_FAILED or
            SI_INVALID_HANDLE or
            SI_INVALID_PARAMETER
```
3.15. SI_SetFlowControl

**Description:** Adjusts the following flow control settings: set hardware handshaking, software handshaking, and modem control signals. See "Appendix D—Definitions from C++ header file SiUSBXp.h" for pin characteristic definitions.

**Supported Devices:** CP2101/2/3/4/5/8/9

**Prototype:**
```c
SI_STATUS SI_SetFlowControl (HANDLE Handle, BYTE bCTS_MaskCode, BYTE bRTS_MaskCode, BYTE bDTR_MaskCode, BYTE bDSRMaskCode, BYTE bDCD_MaskCode, BYTE bFlowXonXoff)
```

**Parameters:**
1. **Handle**—Handle to the device as returned by `SI_Open`.
2. **bCTS_MaskCode**—The CTS pin characteristic must be as follows: 
   - SI_STATUS_INPUT or
   - SI_HANDSHAKE_LINE.
3. **bRTS_MaskCode**—The RTS pin characteristic must be as follows: 
   - SI_HELD_ACTIVE,
   - SI_HELD_INACTIVE,
   - SI_FIRMWARE_CONTROLLED or
   - SI_TRANSMIT_ACTIVE_SIGNAL.
4. **bDTR_MaskCode**—The DTR pin characteristic must be as follows: 
   - SI_HELD_INACTIVE,
   - SI_HELD_ACTIVE or
   - SI_FIRMWARE_CONTROLLED.
5. **bDSR_MaskCode**—The DSR pin characteristic must be as follows: 
   - SI_STATUS_INPUT or
   - SI_HANDSHAKE_LINE.
6. **bDCD_MaskCode**—The DCD pin characteristic must be as follows: 
   - SI_STATUS_INPUT or
   - SI_HANDSHAKE_LINE.
7. **bFlowXonXoff**—Sets software flow control to be off if the value is 0, and on using the character value specified if value is non-zero.

**Return Value:**
```c
SI_STATUS = SI_SUCCESS or
SI_DEVICE_IO_FAILED or
SI_INVALID_HANDLE or
SI_INVALID_PARAMETER
```
3.16. **SI_GetModem Status**  
**Description:** Gets the Modem Status from the device. This includes the modem pin states.

**Supported Devices:** CP2101/2/3/4/5/8/9

**Prototype:**  
```
SI_STATUS SI_GetModemStatus (HANDLE Handle, PBYTE ModemStatus)
```

**Parameters:**  
1. **Handle**—Handle to the device as returned by `SI_Open`.  
2. **lpbModemStatus**—Address of a BYTE variable that contains the current states of the RS-232 modem control lines. The byte is defined as follows:

   - Bit 0: DTR State
   - Bit 1: RTS State
   - Bit 4: CTS State
   - Bit 5: DSR State
   - Bit 6: RI State
   - Bit 7: DCD State

**Return Value:**  
- `SI_STATUS` = `SI_SUCCESS`
- `SI_DEVICE_IO_FAILED`
- `SI_INVALID_HANDLE`
- `SI_INVALID_PARAMETER`

3.17. **SI_SetBreak**  
**Description:** Sends a break state (transmit or reset) to a CP210x device. Note that this function is not necessarily synchronized with queued transmit data.

**Supported Devices:** CP2101/2/3/4/5/8/9

**Prototype:**  
```
SI_STATUS SI_SetBreak(HANDLE cyHandle, WORD wBreakState)
```

**Parameters:**  
1. **Handle**—Handle to the device as returned by `SI_Open`.  
2. **wBreakState**—The break state to set. If this value is a 0x0000 then the break is reset. If this value is a 0x0001 then a break is transmitted.

**Return Value:**  
- `SI_STATUS` = `SI_SUCCESS`
- `SI_DEVICE_IO_FAILED`
- `SI_INVALID_HANDLE`
- `SI_INVALID_PARAMETER`
3.18. **SI_ReadLatch**

**Description:** Gets the current port latch value (least significant four bits) from the device.

**Supported Devices:** CP2103/4/5

**Prototype:**

```c
SI_STATUS SI_ReadLatch (HANDLE Handle, LPBYTE Latch)
```

**Parameters:**

1. *Handle*—Handle to the device as returned by *SI_Open*.
2. *Latch*—Pointer for a return port latch value (Logic High = 1, Logic Low = 0).

**Return Value:**

SI_STATUS = SI_SUCCESS or
SI_DEVICE_NOT_FOUND or
SI_FUNCTION_NOT_SUPPORTED or
SI_GLOBAL_DATA_ERROR or
SI_INVALID_HANDLE or
SI_INVALID_PARAMETER or
SI_DEVICE_IO_FAILED

3.19. **SI_WriteLatch**

**Description:** Sets the current port latch value (least significant four bits) from the device.

**Supported Devices:** CP2103/4/5

**Prototype:**

```c
SI_STATUS SI_WriteLatch (HANDLE Handle, BYTE Mask, BYTE Latch)
```

**Parameters:**

1. *Handle*—Handle to the device as returned by *SI_Open*.
2. *Mask*—Determines which pins to change (Change = 1, Leave = 0).
3. *Latch*—Value to write to the port latch (Logic High = 1, Logic Low = 0).

**Return Value:**

SI_STATUS = SI_SUCCESS or
SI_DEVICE_NOT_FOUND or
SI_FUNCTION_NOT_SUPPORTED or
SI_GLOBAL_DATA_ERROR or
SI_INVALID_HANDLE or
SI_INVALID_PARAMETER or
SI_DEVICE_IO_FAILED

3.20. **SI_GetPartNumber**

**Description:** Retrieves the part number of the CP210x device for a given handle.

**Supported Devices:** CP2101/2/3/4/5/8/9

**Prototype:**

```c
SI_STATUS SI_GetPartNumber (HANDLE Handle, LPBYTE PartNum)
```

**Parameters:**

1. *Handle*—Handle to the device as returned by *SI_Open*.
2. *Latch*—Pointer for a return part number.

**Return Value:**

SI_STATUS = SI_SUCCESS or
SI_INVALID_PARAMETER or
SI_INVALID_HANDLE or
SI_DEVICE_IO_FAILED
3.21. **SI_DeviceIOControl**

**Description:** Interface for any miscellaneous device control functions. A separate call to `SI_DeviceIOControl` is required for each input or output operation. A single call cannot be used to perform both an input and output operation simultaneously. Refer to `DeviceIOControl` function definition on MSDN Help for more details.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

**Prototype:**

```
SI_STATUS SI_DeviceIOControl (HANDLE Handle, DWORD IOControlCode, LPVOID InBuffer, DWORD BytesToRead, LPVOID OutBuffer, DWORD BytesToWrite, LPDWORD BytesSucceeded)
```

**Parameters:**

1. **Handle**—Handle to the device as returned by `SI_Open`.
2. **IOControlCode**—Code to select control function.
3. **InBuffer**—Pointer to input data buffer.
4. **BytesToRead**—Number of bytes to be read into InBuffer.
5. **OutBuffer**—Pointer to output data buffer.
6. **BytesToWrite**—Number of bytes to write from OutBuffer.
7. **BytesSucceeded**—Address of a DWORD variable that will contain the number of bytes read by an input operation or the number of bytes written by an output operation on return.

**Return Value:** `SI_STATUS = SI_SUCCESS` or `SI_DEVICE_IO_FAILED` or `SI_INVALID_HANDLE`

3.22. **SI_GetDLLVersion**

**Description:** Obtains the version of the DLL that is currently in use. The version is returned in two DWORD values, HighVersion and LowVersion. This corresponds to version A.B.C.D where A = (HighVersion >> 16) & 0xFFFF, B = HighVersion & 0xFFFF, C = (LowVersion >> 16) & 0xFFFF and D = LowVersion & 0xFFFF.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3, CP2101/2/3/4/5/8/9

**Prototype:**

```
SI_STATUS SI_GetDLLVersion (DWORD* HighVersion, DWORD* LowVersion)
```

**Parameters:**

1. **HighVersion**—Address of a DWORD variable that will contain the top 32 bits of the DLL version.
2. **LowVersion**—Address of a DWORD variable that will contain the bottom 32 bits of the DLL version.

**Return Value:** `SI_STATUS = SI_SUCCESS` or `SI_SYSTEM_ERROR_CODE`
3.23. SI_GetDriverVersion

Description: Obtains the version of the Driver that is currently in the Windows System directory. The version is returned in two DWORD values, HighVersion and LowVersion. This corresponds to version A.B.C.D where A = (HighVersion >> 16) & 0xFFFF, B = HighVersion & 0xFFFF, C = (LowVersion >> 16) & 0xFFFF and D = LowVersion & 0xFFFF.


Prototype: SI_STATUS SI_GetDriverVersion (DWORD* HighVersion, DWORD* LowVersion)

Parameters: 1. HighVersion—Address of a DWORD variable that will contain the top 32 bits of the DLL version.
2. LowVersion—Address of a DWORD variable that will contain the bottom 32 bits of the DLL version.

Return Value: SI_STATUS = SI_SUCCESS or SI_SYSTEM_ERROR_CODE
4. Device Interface Functions

The USBXpress firmware library implements a set of device interface functions that provide an Application Programming Interface (API) on the C8051F32x, C8051F34x, C8051F38x, C8051T32x, and C8051T62x microcontrollers. These functions provide a simplified I/O interface to the MCU's USB controller, thus eliminating the need to understand and manage low-level USB hardware or protocol details. The API is provided in the form of a library file precompiled under the Keil C51 tool chain. Device firmware must be developed using the Keil Software C51 tool chain. The device interface functions available are:

- USB_Clock_Start() - Initializes the USB clock
- USB_Init() - Enables the USB interface
- Block_Write() - Writes a buffer of data to the host via the USB
- Block_Read() - Reads a buffer of data from the host via the USB
- GetInterrupt_Source() - Indicates the reason for an API interrupt
- USB_Int_Enable() - Enables the API interrupts
- USB_Int_Disable() - Disables API interrupts
- USB_Disable() - Disables the USB interface
- USB_Suspend() - Suspends the USB interrupts
- USB_Get_Library_Version() - Returns the USBXpress firmware library version

The API is used in an interrupt-driven mode. The user must provide an interrupt handler located at vector address 0x0083 (interrupt 16) for the 'F320/1/6/7 devices, or at vector address 0x008B (interrupt 17) for the 'F34x/'F38x/'T32x/'T62x devices. This handler will be called upon at any USB API interrupt. Once inside this ISR, a call to GetInterrupt_Source is used to determine the source of the interrupt (this call also clears the pending interrupt flags).

The USBXpress firmware library operates the MCU's USB controller at USB Full Speed, and uses the Bulk Transfer type with a data payload of 64 bytes per packet. Code developed for a specific MCU device family ('F320/1, 'F326/7, 'F34x, 'F38x, 'T32x, or 'T62x) must use USBXpress device firmware libraries specific to that family. See "Appendix C—Firmware Library Notes" for more technical details, and differences between the MCU device firmware libraries.

Note: The USB0 hardware interrupt located at vector address 0x0043 (interrupt 8) is claimed by USBXpress, and is used to handle low-level USB protocol details. The USB API interrupt (interrupt 16 for 'F320/1/6/7 devices and interrupt 17 for 'F34x devices) is a virtual interrupt generated by the USBXpress firmware library whenever user code needs to be notified of a USBXpress event. The events are defined in the description of the GetInterrupt_Source function.

Example ISR for Firmware API (interrupt 16 for 'F32x devices, interrupt 17 for 'F34x/'F38x/'T32x/'T62x devices):

```c
void USB_API_TEST_ISR(void) interrupt 16
{
    BYTE INTVAL = GetInterrupt_Source();

    if (INTVAL & TX_COMPLETE)
    {
        Block_Write(In_Packet, 8);
    }

    if (INTVAL & RX_COMPLETE)
    {
        Block_Read(Out_Packet, 8);
    }
```
if (INTVAL & DEV_CONFIGURED) {
    // Initialize all analog peripherals here. This interrupt
    // tells the device that it can now use as much current as
    // specified by the MaxPower descriptor.
    Init(); // Note: example command, not part of the API
}

if (INTVAL & DEV_SUSPEND) {
    // Turn off all analog peripherals
    Turn_Off_All(); // Note: example command, not part of the API
    USB_Suspend(); // This function returns once resume
    // signalling is present.

    // Turn all analog peripherals back on
    Init(); // Note: example command, not part of the API
}

4.1. USB_Clock_Start

Description: Enables the internal oscillator, initializes the clock multiplier, and sets the USB clock to 48 MHz
for USB full speed operation. If the clock multiplier is already initialized, the initialization proce-
dure is skipped. This function should be called before calling USB_Init or accessing any vari-
ables located in the upper 1024 bytes of XRAM (USB clock domain). CLKSEL[1:0] is not affected
by this function. See "Appendix A—SFRs that Should Not be Modified After Calling USB-_Clock_Start and USB_Init" for more details. See "Appendix C—Firmware Library Notes" for instructions on how to use the external oscillator as the USB clock.

Supported Devices:  C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D,
                    C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

Prototype:       void USB_Clock_Start (void)

Parameters:       None

Return Value:     None
4.2. **USB_Init**

**Description:** Enables the USB interface, the USB clock recovery feature, and the use of Device Interface Functions. On return, the USB interface is configured, and C8051F32x interrupts are globally enabled. User software should not globally disable interrupts (set EA = 0), but should enable or disable user configured interrupts individually using the interrupt's source interrupt enable flag present in the IE, EIE1, or EIE2 SFRs. Before calling **USB_Init**, a call to **USB_Clock_Start** should be made to configure the USB clock. See “Appendix A—SFRs that Should Not be Modified After Calling USB_Clock_Start and USB_Init” for more details.

This function allows the user to specify the Vendor and Product IDs as well as Manufacturer, Product Description, and Serial Number strings that are sent to the host as part of the device's USB descriptor during the USB enumeration (device connection).

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

**Prototype:**

```c
void USB_Init (UINT VendorID, UINT ProductID, BYTE *ManufacturerStr, BYTE *ProductStr, BYTE *SerialNumberStr, BYTE MaxPower, BYTE PwAttributes, UINT bcdDevice)
```

**Parameters:**

1. **VendorID**—16-bit Vendor ID to be returned to the host's Operating System during USB enumeration. Set to 0x10C4 to use the default Silicon Laboratories Vendor ID.
2. **ProductID**—16-bit Product ID to be returned to the host's Operating System during USB enumeration. Set to 0xEA61 to associate with the default USBXpress driver.
3. **ManufacturerStr**—Pointer to a character string. See Appendix B for formatting. NULL pointer should not be used because the library does not contain a default value for this string.
4. **ProductStr**—Pointer to a character string. See Appendix B for formatting. NULL pointer should not be used because the library does not contain a default value for this string.
5. **SerialNumberStr**—Pointer to a character string. See Appendix B for formatting. NULL pointer should not be used because the library does not contain a default value for this string.
6. **MaxPower**—Specifies how much bus current a device requires. Set to one half the number of milliamperes required. The maximum allowed current is 500 milliamperes, and hence any value above 0xFA will be automatically set to 0xFA. Example: Set to 0x32 to request 100 mA.
7. **PwAttributes**—Set bit 6 to 1 if the device is self-powered and to 0 if it is bus-powered. Set bit 5 to 1 if the device supports the remote wakeup feature. Bits 0 through 4 must be 0 and bit 7 must be 1. Example: Set to 0x80 to specify a bus-powered device that does not support remote wakeup.
8. **bcdDevice**—The device's release number in BCD (binary-coded decimal) format. In BCD, the upper byte represents the integer, the next four bits are tenths, and the final four bits are hundredths. Example: 2.13 is denoted by 0x0213.

**Return Value:** None
4.3. **Block_Write**

**Description:** Writes a buffer of data to the host via USB. Maximum block size is 4096 bytes. Returns the number of bytes actually written. This matches the parameter NumBytes unless an error condition occurs. A zero is returned if `Block_Write` is called with NumBytes greater than 4096. If NumBytes is greater than 64 bytes, the Bulk Transaction is split into multiple packets, each with a 64-byte data payload (except last packet). `Block_Write` returns after copying the last packet to the device USB transmit buffer. The completion of the transaction is then indicated by the TX_COMPLETE USB API interrupt.

`SI_Read` can read from 0 to 64 kB of data. If `Block_Write` is called multiple times before `SI_Read` is called then there is potential to read all of the data in the host's buffer depending on the amount of data requested in the read. For example, if `Block_Write` is called 4 times, and sends a byte of data in each block the host side can call `SI_Read` requesting 4 bytes and get the data from all 4 of the `Block_Writes` at once.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

**Prototype:**

```
UINT Block_Write (BYTE *Buffer, UINT NumBytes)
```

**Parameters:**

1. `Buffer`—Pointer to a memory location where data to be written is stored.
2. `NumBytes`—Number of bytes to write (1–4096).

**Return Value:** Returns an unsigned 16-bit value indicating the number of bytes actually written.

4.4. **Block_Read**

**Description:** Reads a buffer of data sent from the host via USB. Maximum block size is 64 bytes. The block of data is copied from the USB interface to the memory location pointed to by Buffer. The device USB receive buffer will be emptied on return regardless of whether or not the entire buffer was read by `Block_Read`. The maximum number of bytes to read from the device USB receive buffer is specified in NumBytes. The number of bytes actually read (copied to Buffer) is returned by the function. A zero is returned if there are no bytes to read. Typically, `Block_Read` should be called after receiving a data packet, indicated by an RX_COMPLETE USB API interrupt.

Multiple calls to `Block_Read` might be needed to read all data sent via one `SI_Write` call if the buffer sent to `SI_Write` is more than 64 bytes.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

**Prototype:**

```
BYTE Block_Read (BYTE *Buffer, BYTE NumBytes)
```

**Parameters:**

1. `Buffer`—Pointer to a memory location where data will be copied.
2. `NumBytes`—Number of bytes to read (1–64).

**Return Value:** Returns an unsigned 8-bit value indicating the number of bytes actually read.
### 4.5. Get Interrupt Source

**Description:** Returns an 8-bit value indicating the reason(s) for the API interrupt, and clears the USB API interrupt pending flag(s). This function should be called at the beginning of the user's interrupt service routine to determine which event(s) has/have occurred.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

**Prototype:** `BYTE Get_Interrupt_Source (void)`

**Parameters:** None

**Return Value:** Returns an unsigned 8-bit code indicating the reason(s) for the API interrupt. The code can indicate more than one type of interrupt at the same time. The return values are coded as follows:

- 0x00: No USB API Interrupts have occurred
- 0x01: USB_RESET: USB Reset Interrupt has occurred
- 0x02: TX_COMPLETE: Transmit Complete Interrupt has occurred
- 0x04: RX_COMPLETE: Receive Complete Interrupt has occurred
- 0x08: FIFO_PURGE: Command received (and serviced) from the host to purge the USB buffers
- 0x10: DEVICE_OPEN: Device Instance Opened on host side
- 0x20: DEVICE_CLOSE: Device Instance Closed on host side
- 0x40: DEV_CONFIGURED: Device has entered configured state
- 0x80: DEV_SUSPEND: USB suspend signaling present on bus

### 4.6. USB_Int_Enable

**Description:** A call to this function enables the USB API to generate interrupts. If enabled, a USB API interrupt is generated on the following API events:

1. A USB Reset has occurred.
2. A transmit scheduled by a call to `Block_Write` has completed.
3. The RX buffer is ready to be serviced by a call to `Block_Read`.
4. A command from the host has caused the USB buffers to be flushed.
5. A Device Instance has been opened or closed by the host.

The cause of the interrupt can be determined by a call to `Get_Interrupt_Source`. If USB API interrupts are enabled, the user must provide an interrupt service routine with the entry point located at the interrupt 16 vector (Address = 0x0083). When this function is called, control will transfer to the interrupt 16 handler within one ms, if any interrupts are currently pending.

**Supported Devices:** C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

**Prototype:** `void USB_Int_Enable (void)`

**Parameters:** None

**Return Value:** None
4.7. USB_Int_Disable

Description: This function disables the USB API interrupt generation.

Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

Prototype: void USB_Int_Disable (void)

Parameters: None

Return Value: None

4.8. USB_Disable

Description: This function disables the USB interface and the use of Device Interface Functions. On return, the USB interface is no longer available and API interrupts are turned off. The clock multiplier is turned off to reduce power consumption unless the system clock is set to the '4x Clock Multiplier/2' option (CLKSEL[1:0] = 10b).

Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

Prototype: void USB_Disable (void)

Parameters: None

Return Value: None

4.9. USB_Suspend

Description: This function allows devices to meet the USB suspend current specification. To be USB compliant, a USB device must support the Suspend feature by reducing its total power consumption to be under 500 µA. This function should only be called when the DEV_SUSPEND USB API interrupt is received. All unnecessary user peripherals should be turned off before making this function call, and can be turned back on after the call returns. This routine powers down the USB transceiver and the clock multiplier and then suspends the internal oscillator until USB resume signaling occurs. Once USB traffic is detected, internal oscillator is restarted, USB_Clock_Start is called, and then the function call returns to user code. Note: USB_Suspend will set the system clock to internal oscillator by default. If system clock is set to clock multiplier when USB_Suspend is called, that setting will be restored before this function returns. If it is necessary to use any other setting for system clock, user code should modify CLKSEL on return from USB_Suspend.

Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

Prototype: void USB_Suspend (void)

Parameters: None

Return Value: None
4.10. USB_Get_Library_Version

Description: This function returns the USBXpress firmware library version number in BCD. This function is available in USBXpress firmware libraries from release 2.4 and above. Example: Rev. 2.41 is returned as 0x0241.

Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

Prototype: UINT USB_Get_Library_Version (void)

Parameters: None

Return Value: Returns the USBXpress firmware library version number as an unsigned 16-bit value in BCD format.
APPENDIX A—SFRs THAT SHOULD NOT BE MODIFIED AFTER CALLING USB_CLOCK_START AND USB_INIT

Supported Devices:  
C8051F320/1/6/7, C8051F340/1/3/4/5/6/7/8/9/A/B/C/D,  
C8051F380/1/2/3/4/5/6/7,  
C8051T320/1/2/3/6/7,  
C8051T620/1/2/3

The following is a list of SFRs configured by the API. These should not be altered at any time after the first call to USB_Clock_Start or USB_Init. Most of these SFRs are dedicated to the USB peripheral on the chip and should be of no concern to the programmer under most circumstances.

Off-Limits USB SFRs—USB0XCN, USB0ADR, and USB0DAT

Off-Limits Other SFRs—CLKMUL, OSCICN (Only bits 5–7 are off-limits), CLKSEL (Only bits 4–6 are off-limits). These three SFRs are used to enable the internal oscillator, engage the 4x clock multiplier to 48 MHz, and to use that as the clock for the USB core. For the API to function properly, these should not be modified.

API—User Shared SFRs:

The CLKSEL SFR is used for choosing both the system clock source and USB clock source. Care should be used to OR in the system clock desired into Bits 1–0, so as not to disturb Bits 6–4, which are the USB clock selection bits.

The OSCICN SFR is used to control the internal oscillator. The IFCN[1:0] bits can be modified as required by the user to modify the system clock frequency. Note that the IFCN bits do not affect the 12 MHz clock multiplier input or the USB clock. Care should be taken to preserve bits 5–7 while modifying the IFCN bits.
APPENDIX B—FORMAT OF USER-DEFINED PRODUCT DESCRIPTION AND SERIAL NUMBER STRINGS

Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

It is possible for the API to use strings defined and allocated in user firmware instead of the API default strings. The syntax for defining and using custom strings is:

```
unsigned char CustomString[]={number of string elements,0x03,'A',0,'B',0,'C',0...'Z',0};
```

The number of string elements = number of letters x 2 + 2, since every letter needs to be separated from the next by zeros, and USB requires that the first element be the length, and the second element is 0x03, meaning string descriptor type. This sounds harder than it is, for example:

```
//ABC Inc
unsigned char CustomString1[]={16,0x03,'A',0,'B',0,'C',0,' ',0,'I',0,'n',0,'c',0};

//Widget
unsigned char CustomString2[]={14,0x03,'W',0,'i',0,'d',0,'g',0,'e',0,'t',0};

//12345
unsigned char CustomString3[]={12,0x03,'1',0,'2',0,'3',0,'4',0,'5',0};
```

Then, if the Vendor ID and Product ID were 0xABCD and 0x1123, the call to `USB_Init` would be

```
USB_Init (0xABCD, 0x1123, CustomString1, CustomString2, CustomString3);
```

Note: It is useful to use the code keyword preceding the CustomString definitions, so that the strings are located in code space.
Supported Devices: C8051F320/1/6/7, C8051F340/1/2/3/4/5/6/7/8/9/A/B/C/D, C8051F380/1/2/3/4/5/6/7, C8051T320/1/2/3/6/7, C8051T620/1/2/3

Tool Chain

The USBXpress Firmware Library has been developed with the Keil C51 Tool Chain, and is distributed as a pre-compiled library. Hence the user project should be built using the Keil C51 tool chain, with the USB_API.LIB included as an external library. A header file USB_API.h with macro definitions and function prototypes is also provided.

Memory-Model Concerns

The firmware API library was created using the small memory model. Using this library in a project with a default memory model of large or compact can cause warnings to occur, depending on warning level settings. To avoid this, set the default memory model to small, and override this setting wherever necessary by defining each function with the large compiler keyword.

The “using” Keyword

The “using” keyword should not be used with the USB API ISR. This compile-time optimization is not supported by the USBXpress library code that is used to create the virtual USB API interrupt (interrupt # 16).

Internal Functions and Variables

All internal function names and global variable names in the USBXpress firmware library begin with the prefix “USBXcore”. To avoid conflict with these PUBLIC symbols that will, if duplicated, result in “MULTIPLE PUBLIC DEFINITIONS” errors, global variables and function names in user firmware should not begin with this prefix.

Using External Oscillator or Clock

By default, USBXpress uses the internal 12 MHz oscillator along with the 4x Clock Multiplier as the USB Clock. To override this, the user firmware can provide its own USB_Clock_Start function. The Keil linker will then override the library function with the user-supplied function. If a high precision external crystal or clock is used, you may want to turn off the USB clock recovery feature. To do this, user firmware code should include a dummy function definition as shown below. This will override the corresponding internal function in the library.

```c
void USBXcore_ClkRec(void) large {}
```

Saving XDATA Space

The USB_Init function parameters are passed in direct memory locations in user XDATA space determined by the linker. If user firmware needs this contiguous space, these 17 bytes can be relocated to unused XDATA space within the USBXpress reserved area. To do this, the following should be added to the command line while invoking the linker (the value for “address” is shown in Table 1):

XDATA(?XD?_USB_INIT?USB_API(<address>))

Example: BL51.exe file1.obj, file2.obj, fileN.obj, USB_API.LIB TO prj1 RS(256) PL(68) PW(78) XDATA(?XD?_USB_INIT?USB_API(0x03EF))
Firmware Library Code Size and Other Details

The Flash memory occupied by the USBXpress firmware library depends on the number of library functions used by the user application. This is because the linker includes only the called functions in the build. If all USBXpress functions are used, the library would occupy ~3 kB of code memory. The low-level settings configured by the USBXpress firmware library are shown in Table 1.

Table 1. Firmware Library Technical Details

<table>
<thead>
<tr>
<th></th>
<th>C8051F320/1</th>
<th>C8051F326/7</th>
<th>C8051F340/1/2/3/4/5/6/7/8/ 9/A/B/C/D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Oscillator¹</td>
<td>Enabled (OSCICN.7 = 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x Clock Multiplier¹</td>
<td>Enabled (Source: Internal Oscillator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Clock Recovery¹</td>
<td>Enabled (CLKREC = 0x89)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Clock Source¹</td>
<td>Clock multiplier (48 MHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Speed</td>
<td>Full Speed (12 Mbps)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Transfer Type</td>
<td>Bulk Transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max data payload size (Control Endpoint, EP0)</td>
<td>64 bytes per data packet</td>
<td>64 bytes per data packet</td>
<td>64 bytes per data packet</td>
</tr>
<tr>
<td>Number of bulk data endpoints used</td>
<td>2 (EP2 in Split Mode)</td>
<td>2 (EP1 in Split Mode)</td>
<td>2 (EP2 in Split Mode)</td>
</tr>
<tr>
<td>Max data payload size (Bulk data endpoints)</td>
<td>64 bytes per data packet</td>
<td>64 bytes per data packet</td>
<td>64 bytes per data packet</td>
</tr>
<tr>
<td>Double buffering</td>
<td>Enabled for both IN and OUT endpoints (FIFO can hold two packets each at any time).</td>
<td>Enabled for OUT endpoint (FIFO can hold two packets at any time). Disabled for IN endpoint.</td>
<td>Enabled for both IN and OUT endpoints (FIFO can hold two packets each at any time).</td>
</tr>
<tr>
<td>XDATA space reserved by the library²</td>
<td>448 bytes XDATA (0x0640 to 0x07FF) [includes USB FIFO space]</td>
<td>128 bytes (0x0380 to 0x03FF) XDATA + 256 bytes (0x00 to 0xFF) USB FIFO</td>
<td>448 bytes XDATA (0x0640 to 0x07FF) [includes USB FIFO space]</td>
</tr>
<tr>
<td>Starting address to relocate USB_Init function parameters (see &quot;Saving XDATA Space&quot; on page 26)</td>
<td>0x07AF</td>
<td>0x03EF</td>
<td>0x07AF</td>
</tr>
<tr>
<td>USBXpress Firmware Library Name</td>
<td>USBX_F320_1.LIB</td>
<td>USBX_F326_7.LIB</td>
<td>USBX_F34X.LIB</td>
</tr>
<tr>
<td>C8051F380/1/2/3/4/5/6/7</td>
<td>C8051T620/1 and C8051T320/1/2/3</td>
<td>C8051T622/3 and C8051T326/7</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. The clock settings listed in this table are valid only if the default USBXpress clock functions (USB_Clock_Start and USBXcore_ClkRec) are not overridden by user firmware.
2. This reserved space includes the relocated USB_Init parameters using linker commands. See "Saving XDATA Space" on page 26 for more details. This only applies to the Keil libraries.
Table 1. Firmware Library Technical Details (Continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Setting/Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Oscillator¹</td>
<td>Enabled (OSCICN.7 = 1)</td>
</tr>
<tr>
<td>4x Clock Multiplier¹</td>
<td>Enabled (Source: Internal Oscillator)</td>
</tr>
<tr>
<td>USB Clock Recovery¹</td>
<td>Enabled (CLKREC = 0x8F)</td>
</tr>
<tr>
<td>USB Clock Source¹</td>
<td>Clock multiplier (48 MHz)</td>
</tr>
<tr>
<td>USB Speed</td>
<td>Full Speed (12 Mbps)</td>
</tr>
<tr>
<td>USB Transfer Type</td>
<td>Bulk Transfer</td>
</tr>
<tr>
<td>Max data payload size (Control Endpoint, EP0)</td>
<td>64 bytes per data packet</td>
</tr>
<tr>
<td>Number of bulk data endpoints used</td>
<td>2 (EP2 in Split Mode)</td>
</tr>
<tr>
<td>Max data payload size (Bulk data endpoints)</td>
<td>64 bytes per data packet</td>
</tr>
<tr>
<td>Double buffering</td>
<td>Enabled for both IN and OUT endpoints (FIFO can hold two packets each at any time).</td>
</tr>
<tr>
<td>XDATA space reserved by the library²</td>
<td>448 bytes XDATA (0x0640 to 0x07FF) [includes USB FIFO space]</td>
</tr>
<tr>
<td>Starting address to relocate USB_Init function parameters (see &quot; Saving XDATA Space&quot; on page 26)</td>
<td>0x07AF</td>
</tr>
<tr>
<td>USBXpress Firmware Library Name</td>
<td>USBX_F38X.LIB, USBX_T620_1_320_3.LIB, USBX_T622_3_326_7.LIB</td>
</tr>
</tbody>
</table>

Notes:
1. The clock settings listed in this table are valid only if the default USBXpress clock functions (USB_Clock_Start and USBXcore_ClkRec) are not overridden by user firmware.
2. This reserved space includes the relocated USB_Init parameters using linker commands. See " Saving XDATA Space" on page 26 for more details. This only applies to the Keil libraries.
Type Definitions from Firmware Library Header File USB_API.h

// UINT type definition
#ifndef _UINT_DEF_
define _UINT_DEF_
typedef unsigned int UINT;
#undef _UINT_DEF_ /* _UINT_DEF_ */

// BYTE type definition
#ifndef _BYTE_DEF_
define _BYTE_DEF_
typedef unsigned char BYTE;
#undef _BYTE_DEF_ /* _BYTE_DEF_ */
APPENDIX D—DEFINITIONS FROM C++ HEADER FILE SiUSBXp.H


// Return codes
#define SI_SUCCESS 0x00
#define SI_DEVICE_NOT_FOUND 0xFF
#define SI_INVALID_HANDLE 0x01
#define SI_READ_ERROR 0x02
#define SI_WRITE_ERROR 0x04
#define SI_RESET_ERROR 0x05
#define SI_INVALID_PARAMETER 0x06
#define SI_INVALID_REQUEST_LENGTH 0x07
#define SI_DEVICE_IO_FAILED 0x08
#define SI_INVALID_BAUDRATE 0x09
#define SI_FUNCTION_NOT_SUPPORTED 0x0a
#define SI_GLOBAL_DATA_ERROR 0x0b
#define SI_SYSTEM_ERROR_CODE 0x0c
#define SI_READ_TIMED_OUT 0x0d
#define SI_WRITE_TIMED_OUT 0x0e
#define SI_IO_PENDING 0x0f

// GetProductString() function flags
#define SI_RETURN_SERIAL_NUMBER 0x00
#define SI_RETURN_DESCRIPTION 0x01
#define SI_RETURN_LINK_NAME 0x02
#define SI_RETURN_VID 0x03
#define SI_RETURN_PID 0x04

// RX Queue status flags
#define SI_RX_NO_OVERRUN 0x00
#define SI_RX_EMPTY 0x00
#define SI_RX_OVERRUN 0x01
#define SI_RX_READY 0x02

// Buffer size limits
#define SI_MAX_DEVICE_STRLEN 256
#define SI_MAX_READ_SIZE 4096*16
#define SI_MAX_WRITE_SIZE 4096

// Type definitions
typedef int SI_STATUS;
typedef char SI_DEVICE_STRING[SI_MAX_DEVICE_STRLEN];

// Input and Output pin Characteristics


```c
#define SI_HELD_INACTIVE 0x00
#define SI_HELD_ACTIVE 0x01
#define SI_FIRMWARE_CONTROLLED 0x02
#define SI_RECEIVE_FLOW_CONTROL 0x02
#define SI_TRANSMIT_ACTIVE_SIGNAL 0x03
#define SI_STATUS_INPUT 0x00
#define SI_HANDSHAKE_LINE 0x01

// Mask and Latch value bit definitions
#define SI_GPIO_00 0x01
#define SI_GPIO_1 0x02
#define SI_GPIO_2 0x04
#define SI_GPIO_3 0x08

// GetDeviceVersion() return codes
#define SI_CP2101_VERSION 0x01
#define SI_CP2102_VERSION 0x02
#define SI_CP2103_VERSION 0x03
#define SI_CP2104_VERSION 0x04
#define SI_CP2105_VERSION 0x05

// Common variable and type definitions used
typedef unsigned long DWORD;
typedef int BOOL;
typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef BYTE near *PBYTE;
typedef DWORD near *PDWORD;
typedef DWORD far *LPDWORD;
typedef void far *LPVOID;
```

SI_SUCCESS
The function succeeded.

SI_DEVICE_NOT_FOUND
The device cannot be found on the system. Make sure the device is plugged in and powered. If the device is plugged in, make sure that all previous application handles to the device have been closed (SI_Close). If a previous instance of the application was not able to close its handle to the device before exiting, disconnect and reconnect the device. To avoid having to temporarily remove the device in this case, you may have your application store the current handle value (returned by SI_Open) in the Windows registry so that if the application crashes, the handle is still accessible and can be closed (SI_Close).

SI_INVALID_HANDLE
The value of the Handle passed to the function is not valid. A valid handle is obtained by declaring a HANDLE variable in your program and passing the address of that HANDLE to the SI_Open function. A Handle may become invalid if the device is removed from the system, so first verify that the device is connected.

SI_WRITE_ERROR
The write operation failed. The device may have been removed.

SI_INVALID_PARAMETER
An invalid parameter was passed to the DLL function called. See the function definition for valid parameter types and/or ranges.

SI_INVALID_REQUEST_LENGTH
See SI_Read and SI_Write function descriptions for valid request lengths.

SI_DEVICE_IO_FAILED
Device IO operation failed. The device may have been removed.

SI_INVALID_BAUDRATE
See the CP210x device-specific data sheet for supported baud rates.

SI_FUNCTION_NOT_SUPPORTED
The function called is not supported by the device. For example, attempting to use the SI_ReadLatch and SI_WriteLatch functions on a device other than the CP2103 will cause the functions to return this value.

SI_GLOBAL_DATA_ERROR
An error has occurred such that the thread global data cannot be retrieved. Unload and reload the DLL if this return code is received.

SI_SYSTEM_ERROR_CODE
Call GetLastError (Win32 Base) to retrieve Windows System Error Code. The error codes are defined on MSDN.
SI_READ_TIMED_OUT or SI_WRITE_TIMED_OUT

The read or write request timed out based on the current timeout values.

SI_IO_PENDING

I/O is pending, wait on the OVERLAPPED object supplied to the SI_Read or SI_Write function using WaitForSingleObject() or GetOverlappedResult() as documented on MSDN by Microsoft, and/or SI_CancelIo().
APPENDIX F—UPDATING HOST CODE TO WORK UNDER USBXPRESS 3.X.X

Note: The USBXpress 3.X.X package works different functionally from previous versions (2.42 and earlier). Do not mix and match any of these old DLLs or Drivers with any part of the 3.X.X package. This will result in data error and possibly a system crash.

The SI_Close() function has changed from SI_Close(&HANDLE Handle) to SI_Close(HANDLE Handle). Versions before 3.X.X would set the HANDLE value to INVALID_HANDLE_VALUE when a close was called. Now it is the responsibility of the developer to set the handle value to be invalid. For Visual Basic users, this change will also require a Declare statement modification. The correct way to call this function in VB is:

```vbnet
Public Declare Function SI_Close Lib "SiUSBXp.DLL" _
    ByVal cyHandle As Long) As Integer
```

The Read and Write functions have changed to include a parameter for a pointer to an initialized OVERLAPPED object. For C++ users the prototype has been set so that this parameter is false by default allowing older code to be ported directly. For VB users the Declare statements will have to be updated as follows:

```vbnet
Public Declare Function SI_Read Lib "SiUSBXp.DLL" _
    ByVal cyHandle As Long, ByVal lpBuffer As Byte, _
    ByVal dwBytesToRead As Long, ByVal lpdwBytesReturned As Long, _
    ByVal lpOverlapped As Long) As Integer

Public Declare Function SI_Write Lib "SiUSBXp.DLL" _
    ByVal cyHandle As Long, ByVal lpBuffer As Byte, _
    ByVal dwBytesToWrite As Long, ByVal lpdwBytesWritten As Long, _
    ByVal lpOverlapped As Long) As Integer
```
DOCUMENT CHANGE LIST

Revision 1.6 to Revision 1.7
- Updated all Host API information to reflect 3.X.X changes. See "Appendix F—Updating Host Code to Work Under USBXpress 3.X.X" for important compatibility information.
- Device API documentation:
  - Updated description in "3. Device Interface Functions" to reflect support for 'F326/7 and 'F34x devices.
  - Added description of function "3.10. USB_Get_Library_Version".
  - Updated Table 1 to show details of the 'F326/7 and 'F34x device firmware libraries.

Revision 1.7 to Revision 1.8
- Modified return values for SI_Write.
- Modified descriptions of blocking in SI_Write/ SI_Read.
- Further explained the Block_Write in relation to SI_Read.

Revision 1.8 to Revision 1.9
- Modified return values for SI_Read.
- Removed references to unused return code SI_RX_QUEUE_NOT_READY
- Modified to explain SI_GetDLLVersion and SI_GetDriverVersion

Revision 1.9 to Revision 2.0
- Updated functions return values in "3. Host API Functions" on page 3.
- Corrected prototype declaration for "3.5. SI_Read" on page 6 and "3.6. SI_Write" on page 7.

Revision 2.0 to Revision 2.1
- Added CP2104 and CP2105 to Relevant Devices on Page 1.
- Updated Host API functions to reflect support for CP2104 and CP2105 devices.

Revision 2.1 to Revision 2.2
- Added support for C8051F38x, C8051T32x, and C8051T62x devices.

Revision 2.2 to Revision 2.3
- Added support for CP2108 and CP2109.
- Updated CancelIo() to SI_CancelIo() and added the SI_CancelIo() API information.
- Added "2. USBXpress Versions" on page 2.
- Updated XDATA space reserved by the library for C8051F326/7, C8051T622/3, and C8051T326/7 in
Disclaimer
Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and “Typical” parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A “Life Support System” is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information
Silicon Laboratories Inc., Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, “the world’s most energy friendly microcontrollers”, Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOmodem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.