AN197: Serial Communications Guide for the CP210x

This application note applies to the following devices: CP2101, CP2102, CP2103, CP2104, CP2105, and CP2108.

This document is intended for developers creating products based on the CP210x USB to UART Bridge Controller. It provides information about serial communications and how to obtain the port number for a specific CP210x device. Code samples are provided for opening, closing, configuring, reading, and writing to a COM port. Also included is a `GetPortNum` function that can be copied and used to determine the port number on a CP210x device by using its Vendor ID (VID), Product ID (PID), and serial number.

KEY POINTS

- Shows how to open a COM Port
- Discusses how to prepare an open COM Port for data transmission
- Shows how to close the COM port
- Includes a sample program to demonstrate serial functions
- Shows how to discover the CP210x COM Port
- Includes application design notes

1. Opening a COM Port

Before configuring and using a COM port to send and receive data, it must first be opened. When a COM port is opened, a handle is returned by the `CreateFile()` function that is used from then on for all communication. Here is example code that opens COM3:

```
HANDLE hMasterCOM = CreateFile("\\\\.\\COM3",
      GENERIC_READ | GENERIC_WRITE,
      0,
      0,
      OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
      0);
```

The first parameter in the `CreateFile()` function is a string that contains the COM port number to use. This string will always be of the form `\\\\.\\COMX` where 'X' is the COM port number to use. The second parameter contains flags describing access, which will be `GENERIC_READ` and `GENERIC_WRITE` for the example in this document, and allows both read and write access. Parameters three and four must always be 0, and the flag in parameter five must always be `OPEN_EXISTING` when using `CreateFile()` for COM applications. The sixth parameter should always contain the `FILE_ATTRIBUTE_NORMAL` flag. In addition, the `FILE_FLAG_OVERLAPPED` is an optional flag that is used when working with asynchronous transfers (this option is used for the example in this document). If overlapped mode is used, functions that read and write to the COM port must specify an `OVERLAPPED` structure identifying the file pointer, which is demonstrated in 2.1 Purging the COM Port and 2.2 Saving the COM Port's Original State (more information on overlapped I/O is located at https://msdn.microsoft.com/en-us/library/windows/desktop/ms686358(v=vs.85).aspx). The seventh, and last, parameter must always be 0.

If this function returns successfully, then a handle to the COM port will be assigned to the `HANDLE` variable. If the function fails, then `INVALID_HANDLE_VALUE` will be returned. Upon return, check the handle and if it is valid, then prepare the COM port for data transmission.

2. Preparing an Open COM Port for Data Transmission

Once a handle is successfully assigned to a COM port, several steps must be taken to set it up. The COM port must first be purged and its initial state should be retrieved. Then the COM port's new settings can be assigned and set up by a device control block (DCB) structure (more information is provided on the DCB structure in 2.3 Setting up a DCB Structure to Set the New COM State and at https://msdn.microsoft.com/en-us/library/windows/desktop/aa363214(v=vs.85).aspx).

2.1 Purging the COM Port

First, the COM port should be purged to clear any existing data going to or from the COM port using the `PurgeComm()` function:

```
PurgeComm(hMasterCOM, PURGE_TXABORT | PURGE_RXABORT | PURGE_TXCLEAR | PURGE_RXCLEAR);
```

The first parameter in the `PurgeComm()` function is a handle to the open COM port that will be purged. The second parameter contains flags that further describe what actions should be taken. All four flags, `PURGE_TXABORT`, `PURGE_RXABORT`, `PURGE_TXCLEAR`, and `PURGE_RXCLEAR` should always be used. The first two flags terminate overlapped write and read operations, and the last two flags clear the output and input buffers.

If this function returns successfully then a non-zero value is returned. If the function fails, then it returns 0. Upon return, check the return value; if it is non-zero, continue to set up the COM port (more information on the `PurgeComm()` function is located at https://msdn.microsoft.com/en-us/library/windows/desktop/aa363428(v=vs.85).aspx).

2.2 Saving the COM Port's Original State

Since the COM port settings can be modified to meet different needs, it is good practice to obtain the COM port's current state and store it so that when the COM port is closed, the COM port can be restored back to its original state. This can be done using the `GetCommState()` function:

```
DCB dcbMasterInitState;
GetCommState(hMasterCOM, &dcbMasterInitState);
```

The first parameter in the `GetCommState()` function is a handle to the open COM port to obtain settings from. The second parameter is an address to a DCB structure to store the COM port's settings. This DCB structure should also be used as the initial state when specifying new settings for the COM port (see 2.3 Setting up a DCB Structure to Set the New COM State).

If this function returns successfully then a non-zero value is returned. If the function fails, then it returns 0. Upon return, check the return value; if it is non-zero, continue to set up the COM port (more information on the `GetCommState()` function is located at https://msdn.microsoft.com/en-us/library/windows/desktop/aa363260(v=vs.85).aspx).

2.3 Setting up a DCB Structure to Set the New COM State

All of a COM port's settings are stored in a DCB structure. In section 2.2 Saving the COM Port's Original State a DCB structure was retrieved that contained the initial settings of the COM port by using the `GetCommState()` function. To change a COM port's settings, a DCB structure must be created and filled out with the desired settings. Then the `SetCommState()` function can be used to activate those settings:

```
DCB dcbMaster = dcbMasterInitState;

dcbMaster.BaudRate = 57600;
dcbMaster.Parity = NOPARITY;
dcbMaster.ByteSize = 8;
dcbMaster.StopBits = ONESTOPBIT;

SetCommState(hMasterCOM, &dcbMaster);

Delay(60);
```

Here a new DCB structure dcbMaster has been initialized to `dcbMasterInitState`, which are the current settings of the COM port. After it has been initialized to the current settings, new settings can be assigned.

2.3.1 Baud Rate

The baud rate property is set to 57600 bps, but can be set to any of the baud rates supported by the CP210x. (See the current data sheet for the list of supported baud rates for the CP210x.)

2.3.2 Parity

The parity is set to `NOPARITY`, however it can also be set to `ODDPARITY`, `EVENPARITY`, `SPACEPARITY`, and `MARKPARITY` if supported by the CP210x. (See the current data sheet for the list of supported parities for the CP210x.)

2.3.3 Byte Size

The byte size is set to 8, so there are 8 data bits in every byte of data sent. This can also be set to 5, 6, or 7 if supported by the CP210x. (see the data sheet for the list of supported byte sizes for the CP210x.)

2.3.4 Stop Bits

The stop bits are set to `ONESTOPBIT`, but could also be set to `TWOSTOPBITS` or `ONE5STOPBITS` (1.5). (See the current data sheet for the list of supported stop bits for the CP210x.) All combinations of data and stop bits can be used except for the combination of 5 data bits with 2 stop bits and the combination of 6, 7, or 8 data bits with 1.5 stop bits.

After each of these settings is set to the desired value, the `SetCommState()` function can be called to set up the COM port. The first parameter in the `SetCommState()` function is a handle to the open COM port to change the settings on. The second parameter is an address to a DCB structure containing the COM port's new settings (more information on serial settings using DCB structures is located at https://msdn.microsoft.com/en-us/library/windows/desktop/aa363214(v=vs.85).aspx).

If this function returns successfully, a non-zero value is returned. If the function fails, it returns 0. Upon return, check the return value; if it is non-zero, delay for 60 ms to allow time for the settings to change and then continue to set up the COM port. This delay is not required; however, a conservative time of 60 ms is good practice to ensure that the settings are changed before any other operations take place.

3. Transmitting Data Across the COM Port

Once the COM port is successfully opened and configured, data can be written or read.

3.1 Writing Data

There are several things that need to happen in a write, so it is a good idea to create a function for the writes to be called whenever a write must occur. Here is an example of a write function:

```
bool WriteData(HANDLE handle, BYTE* data, DWORD length, DWORD* dwWritten)
{
   bool success = false;
   OVERLAPPED o = {0};

   o.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

   if (!WriteFile(handle, (LPCVOID)data, length, dwWritten, &o))
   {
      if (GetLastError() == ERROR_IO_PENDING)
         if (WaitForSingleObject(o.hEvent, INFINITE) == WAIT_OBJECT_0)
            if (GetOverlappedResult(handle, &o, dwWritten, FALSE))
            success = true;
   }
   else
      success = true;

   if (*dwWritten != length)
      success = false;

   CloseHandle(o.hEvent);

   return success;
}
```

The parameters passed in to this function are the handle to an open COM port, a pointer to an array of bytes that will be written, the number if bytes that are in the array, and a pointer to a variable to store and return the number of bytes written.

Two local variables are declared at the beginning of the function: a bool named success that will store the success of the write (this is initialized to false, and only set true when the write succeeds) and an overlapped object o which is passed to the `WriteFile()` function and alerts if the transfer is complete or not (this is always initialized to {0} before the `hEvent` is assigned). Creating an event with the `CreateEvent (NULL, FALSE, FALSE, NULL)` function sets the `hEvent` property of o to prepare it to be passed to the `WriteFile()` function (more information on `CreateEvent()` is located at https://msdn.microsoft.com/en-us/library/windows/desktop/ms682396(v=vs.85).aspx).

Next, the `WriteFile()` function is called with the handle, data, length of the data, and variable to store the amount of data that was written (more information on `WriteFile()` is located at https://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx). If this function returns successfully, a non-zero value is returned. If the function fails, it returns 0. The if statement will determine if the write succeeded and if it did not, the last error is retrieved to see if there really was an error or the write just wasn't finished. If `ERROR_IO_PENDING` is returned then object o is then waited on until either the write finishes or fails (if something other than `ERROR_IO_PENDING` is returned by the `GetLastError()` function, then there is the possibility of surprise removal; see 6. Application Design Notes for comments on surprise removal). When the wait is over, the result is obtained so that the amount of bytes written is updated. The success variable will then be assigned with the appropriate value, and the handle of `o.hEvent` is closed. Then the amount of bytes written is checked, and finally the function returns the success of the write, which will be true if the write successfully completed.

3.2 Reading Data

There are several things that need to happen in a read, so it is a good idea to create a function for the reads to be called whenever a read must occur. Here is an example of a read function:

```
bool ReadData(HANDLE handle, BYTE* data, DWORD length, DWORD* dwRead, UINT timeout)
{
   bool success = false;
   OVERLAPPED o = {0};

   o.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

   if (!ReadFile(handle, data, length, dwRead, &o))
   {
      if (GetLastError() == ERROR_IO_PENDING)
         if (WaitForSingleObject(o.hEvent, timeout) == WAIT_OBJECT_0)
            success = true;
         GetOverlappedResult(handle, &o, dwRead, FALSE);
   }
   else
      success = true;

   CloseHandle(o.hEvent);

   return success;
}
```

The parameters passed in to this function are the handle to an open COM port, a pointer to an array of bytes that will be read, the number if bytes that are in the array, a pointer to a variable to store and return the number of bytes read, and a timeout value.

Two local variables are declared at the beginning of the function: a bool named success that will store the success of the read (this is initialized to false, and only set true when the read succeeds), and an overlapped object o which is passed to the `ReadFile()` function and alerts if the transfer is complete or not (this is always initialized to {0} before the `hEvent` is assigned). Creating an event with the `CreateEvent(NULL, FALSE, FALSE, NULL)` function sets the `hEvent` property of o to prepare it to be passed to the `ReadFile()` function (more information on `CreateEvent()` is located at https://msdn.microsoft.com/en-us/library/windows/desktop/ms682396(v=vs.85).aspx).

Next, the `ReadFile()` function is called with the handle, data, length of the data, and variable to store the amount of data that was written (more information on the `ReadFile()` function is located at https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx). If this function returns successfully then a non-zero value is returned. If the function fails, then it will return 0. The if statement will determine if the write succeeded and if it didn't, the last error is retrieved to see if there really was an error or the write just wasn't finished. If `ERROR_IO_PENDING` is returned then object o is then waited on until either the write finishes or fails (if something other than `ERROR_IO_PENDING` is returned by the `GetLastError()` function, then there is the possibility of surprise removal; see 6. Application Design Notes for comments on surprise removal). When the wait is over, the result is obtained so that the amount of bytes read is updated. The success variable will then be assigned with the appropriate value, and the handle of `o.hEvent` is closed. Finally, the function returns the success of the read, which will be true if the read successfully completed.

4. Closing the COM Port

After all communication is finished, then the COM port should then be closed. First, the COM port should be set back to its initial state, and then the handle to the COM port should be closed and set to an invalid handle. Example code is shown below:

```
SetCommState(hMasterCOM, &dcbMasterInitState);

Delay(60);

CloseHandle(hMasterCOM);
hMasterCOM = INVALID_HANDLE_VALUE;
```

The `SetCommState()` function works the same as described in 2.3 Setting up a DCB Structure to Set the New COM State. A delay of 60 ms is used to make sure the settings have time to be set. Finally the device is closed using the `CloseHandle()` function. This function just takes in the handle of the COM port. After this function is called, it is important to set the variable to an `INVALID_HANDLE_VALUE`.

5. Sample Program to Demonstrate Serial Communications

Included in the AN197 software package is a directory named `CP210xSerialTest` which contains the source code and executables for a Visual Studio project that makes use of all the serial communication functions described in section 2. Preparing an Open COM Port for Data Transmission, section 3. Transmitting Data Across the COM Port, and section 4. Closing the COM Port. The program is a basic dialog based application that accepts two COM port numbers, and then will send a test array of 64 bytes of data back and forth between them.

6. Application Design Notes

The functions used in 2. Preparing an Open COM Port for Data Transmission, 3. Transmitting Data Across the COM Port, and 4. Closing the COM Port are Windows COMM API functions. The examples provided are just samples of the recommended way of dealing with serial communication. For more specific information on these functions, see the MSDN website at: https://msdn.microsoft.com/en-us/library/ff802693.aspx.

It should also be noted that the `SetCommState()` function does not save the settings between opening and closing the COM port. As stated before, it is good practice to get the current settings after the COM port is opened, and then restore them before it is closed.

All of the functions here will return an error code. It is a good idea to nest these functions in order to catch errors if they occur by using the `GetLastError()` function. This will also solve any surprise removal problems by allowing the discovery of an invalid handle to be found and dealt with. The example application (CP210xSerialTest) has several cases that will detect surprise removal. In this example, there are checks on every function to make sure that the return code is true. If it is not, then it will display where the error occurred in the output window. As long as correct and supported settings are passed to the functions they should execute normally. Most failures can occur from having an `INVALID_HANDLE_VALUE`, however, the handles must be set to this value after a surprise removal occurs.

Because regular COM ports will always be visible, then data can always be written to them successfully, even if there is no way to read it. However, because the CP210x is a virtual COM port, if the device is removed, then the handle that it uses becomes invalid when trying to write to it. If for some reason the CP210x device is unplugged the write will fail and `ERROR_OPERATION_ABORTED` will be returned by `GetLastError()`. When this happens, the handle needs to be closed and then set to `INVALID_HANDLE_VALUE`. Alternatively, a regular COM port can always be read from, but if there is no data then it will time out. When using the CP210x as the virtual COM port and it is removed before a read occurs, then the read will fail and `ERROR_ACCESS_DENIED` will be returned by `GetLastError()`. Again when this happens, the handle needs to be closed and then set to `INVALID_HANDLE_VALUE`.

7. References

MSDN - use this to search for specific Windows API functions

http://msdn.microsoft.com/

Serial Communication Reference

https://msdn.microsoft.com/en-us/library/ff802693.aspx

Communication Resource Reference

https://msdn.microsoft.com/en-us/library/windows/desktop/aa363196(v=vs.85).aspx

8. Document Change List

8.1 Revision 1.0

February 16, 2017

- Updated format to current style guide.
- Updated all references from AN144 to AN721, which replaces AN144.
- Updated links to MSDN articles.
- Removed section 6 on finding the COM port, since the example is now outdated.

8.2 Revision 0.9

October 19, 2012

- Added CP2108 to Relevant Devices list.

8.3 Revision 0.8

October 14, 2010

- Added CP2104/5 to Relevant Devices.
- Updated GetPortNum function and added support for Windows 7 in Section 7.
- Removed support for Windows 98.

8.4 Revision 0.7

August 14, 2008

- Corrected registry paths in .

8.5 Revision 0.6

October 2007

- Updated XP/2000 references to include Server 2003/Vista.
- Updated WinXP/2000 key listing in section .
- Updated code in section .

8.6 Revision 0.5

February 2005

- Added CP2103 to Relevant Devices on page 1.

8.7 Revision 0.4

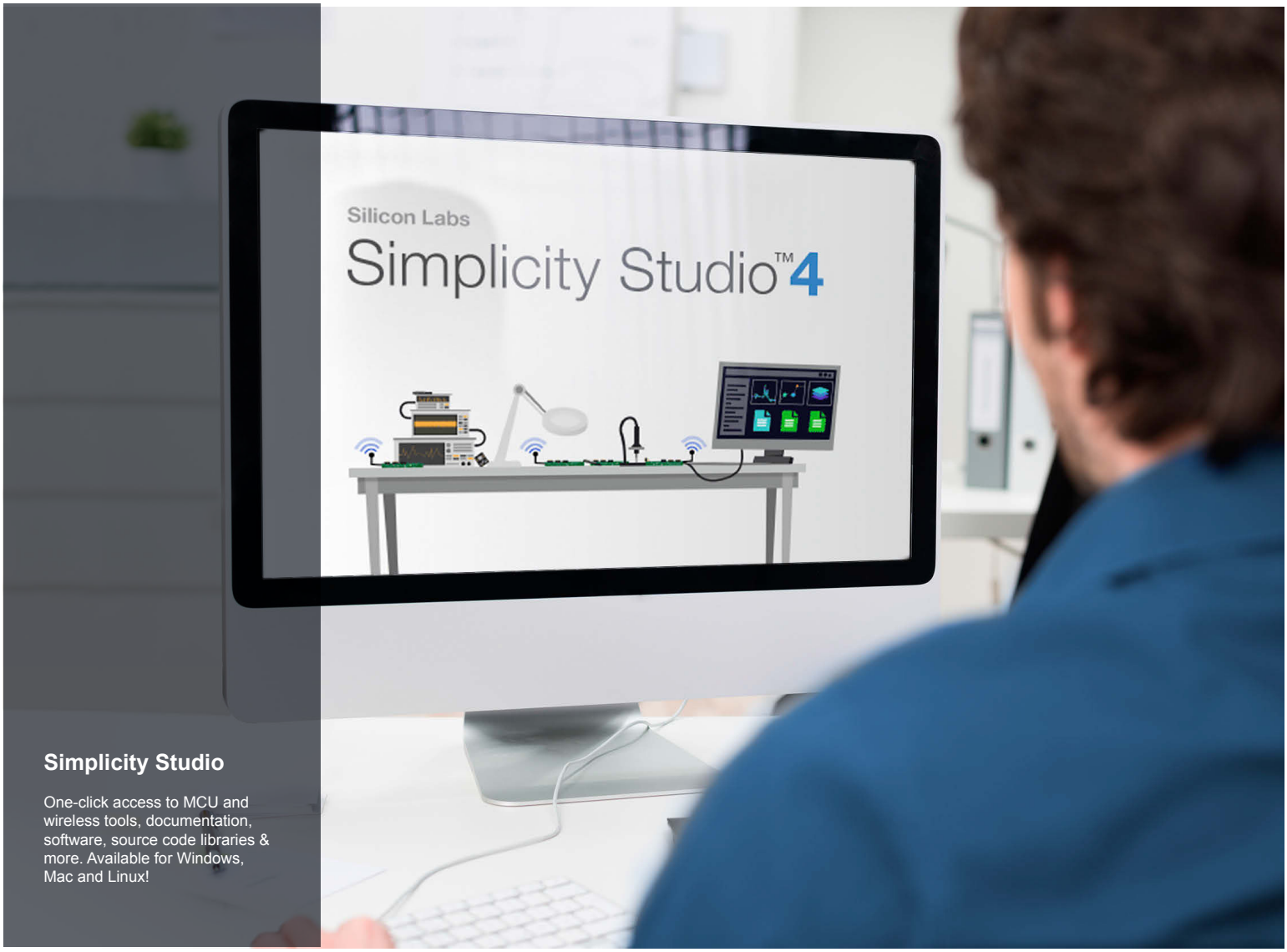
December 2004

- Updated .

8.8 Revision 0.3

October 2004

- Initial release.



Simplicity Studio

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

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

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

http://www.silabs.com