Creating a USB to Serial Bridge Solution using Cypress Low and Full-speed M8 USB Devices

1. Introduction

Peripheral manufacturers have historically used RS-232 as a communications channel to control and to pass data to and from their devices. The adoption of the Universal Serial Bus (USB) as an industry standard serial interface has, however, created an interesting challenge for these manufacturers. The inherent benefits associated with USB (improved performance, reduced cabling, hot-plug capability and better interoperability) dictate that systems migrate support from RS-232 serial interfaces to USB. In addition, customers expect manufacturers to follow the technology curve to newer and better solutions. As more system manufacturers move away from legacy ports such as RS-232, peripheral manufacturers are faced with the requirement to replace their existing RS-232 solutions with USB. The Cypress USB to Serial Reference design provides a complete solution for replacing a legacy serial interface with a USB interface on an existing or new product.

1.1 Objectives

The primary objective of this reference design is to provide OEMs who have RS232-based products with a staged migration path from a pure RS232 product to a USB-based product. Many such OEMs have a large investment in PC application software that interfaces with their associated peripheral over RS232. The Cypress USB-Serial driver is designed to allow the upgrade of hardware to USB, without requiring the substantial investment required to convert their PC software.

A further objective is to provide a staged migration path from RS232 solution to USB solution, that will allow the customer to get a USB version of the product to market as quickly as possible, and then allowing the customer to cost-reduce the product at a later stage.

1.2 Capabilities

The Reference Design is comprised of 2 elements – a Windows device driver and firmware for the CY7C637xx and CY7C640/1xx USB microcontroller families. The driver is universal – it transparently converts Windows serial driver calls into USB communications with the USB-Serial "bridge" device. The driver implements USB communications with the bridge device in a format compatible with the USB HID specification. When the customer is ready to convert the PC application software to communicate directly over USB, the HID class driver (built into the Windows operating system) can be used without any changes being required to the bridge device.

The firmware component implements a general purpose USB-RS232 bridge function, that will work unmodified with "standard" RS232 implementations, and also with many "non-standard" applications of the PC serial port. However, because of the wide variety of ingenious "non-standard" applications of the serial port, it is not possible for this firmware to work unmodified in every application.

However, the firmware can be readily modified to interface with almost any serial device, no matter how non-standard its characteristics. The firmware source code is included in the Reference Design Kit, and customers are encouraged to make enhancements to the firmware to optimize performance for their specific applications.

It is also possible to write firmware for other Cypress USB microcontrollers to meet the requirements of specific applications that cannot be served by the included firmware – for example the EZ-USB family.

1.3 Features

The following is a summary of the features of the reference design:

- Software PC serial port emulation is achieved by creating a virtual COM port which can be accessed by any user-mode Windows application software exactly as if it were a "real" PC serial port.
- Driver supports Windows 98, Windows Me, Windows 2000 and Windows XP.
- Firmware supports all standard baud rates in the range 600 – 57.6k baud (dynamically variable via serial port driver calls). By customizing the firmware to suit specific applications, baud rates of up to 192k baud may be achieved.
- Firmware supports half-duplex serial communication. By customizing the firmware to

suit specific applications, full duplex serial communication may be accommodated.

- Supports RTS, CTS, DRT, DSR, RI, CD control and monitoring directly from user-mode software using Windows serial port device driver calls.
- Supports word lengths of 5, 6, 7 and 8 bytes (dynamically variable via serial port driver calls).
- Supports odd, even and no parity (dynamically variable via serial port driver calls).
- Can sustain data throughputs in excess of 4000 bytes per second for full-speed USB implementations (CY7C640/1xx) and 800 bytes per second for low-speed USB signaling (CY7C637xx).

1.4 Limitations

As previously mentioned, the supplied firmware is general purpose in scope, and this results in some limitations – most notably the half-duplex capability and the 600 - 57.6k baud rate range. Where many of the general purpose features are not required, timing efficiencies may allow the customization of the firmware to support full duplex communications and higher baud rates.

1.5 Scope

This document explains the implementation of a low/medium bandwidth USB to serial bridge. The low bandwidth solution utilizes the CY7C63743 Low-Speed USB microcontroller and the medium bandwidth solution is based on the CY7C64013 Full-Speed USB microcontroller.

The document begins with an overview of the migration path from RS-232 to USB. It then follows up with a description of the RS-232 and USB protocols as well as an introduction to the Cypress USB parts. Finally, this is followed by a discussion of the actual implementation, which includes a description of the firmware, the hardware and interaction with the host driver.

While such a bridge solution might have value as a stand-alone device, it is anticipated that a primary application of this “bridge” will be to be embedded within the RS-232 device itself, providing a drop-in solution to implement USB support for an RS-232 device. This configuration is shown in Figure 3.

2. USB to Serial Migration Path

Historically, serial devices have been connected to hosts via an RS-232 serial connection. This connection provides an interface for control of the device as well as a data path for data being sent to and/or received from the device. This application is illustrated in Figure 1.

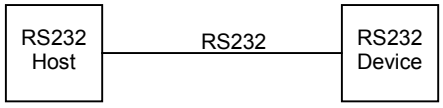

Figure 1. Legacy RS232 application

Conceptually, the USB to Serial solution described herein provides a “bridge” between a USB host and an RS-232 device. Fundamentally, this bridge performs two functions: (1) it receives USB data sent from the host and manages the transmission of that data over an RS-232 connection to the device. (2) It receives RS-232 data sent from the device and passes that data via USB to the host computer. In addition, the bridge also provides transparent control and monitoring of various RS-232 non-data signals. The application of such a bridge is illustrated in Figure 2.

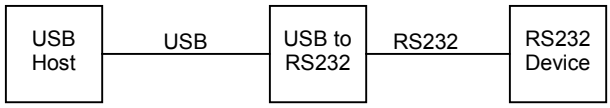

Figure 2. USB to RS232 “Bridge”

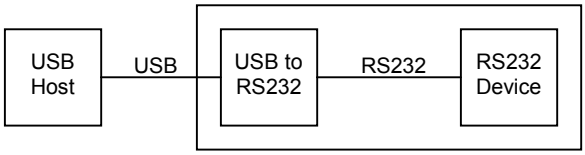

Figure 3. Adding USB support to a Legacy RS232 device

Finally, while the embedded bridge solution provides an effective way to quickly add USB support to an existing RS-232 device, it is expected that peripheral manufacturers will ultimately choose to remove the bridge function altogether, offering a solution that implements a direct-to-USB method. The final step in this migration path is shown in Figure 4.

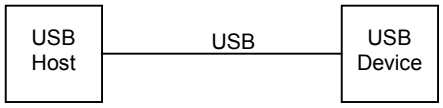

Figure 4. Upgrading Legacy devices to USB

3. RS-232 Overview

RS-232 is a serial data communications protocol. This section describes the different features of the protocol that are relevant to this design.

3.1 Signal Lines

The basic RS-232 protocol uses 9 signal lines. Each signal and its traditional use is described below.

	Signal Name	Description
TXD	Transmit Data	This line carries serial data that is sent from host to device
RXD	Receive Data	This line carries serial data that is sent from device to host
DTR	Data Terminal Ready	Indicates that the host is ready to communicate
DSR	Data Set Ready	Indicates that the device is ready to communicate
RTS	Request To Send	Signals to the device that the host is ready to receive data
CTR	Clear To Send	Signals to the host that the device is ready to receive data
RI	Ring Indicator	Signals to the host that there is an in-coming call
CD	Carrier Detect	Signals to the host that a phone con-nection has been made
SG	Signal Ground	Ground reference

Figure 5 provides a graphic illustration of the signal connection and direction between an RS-232 host and an RS-232 device.

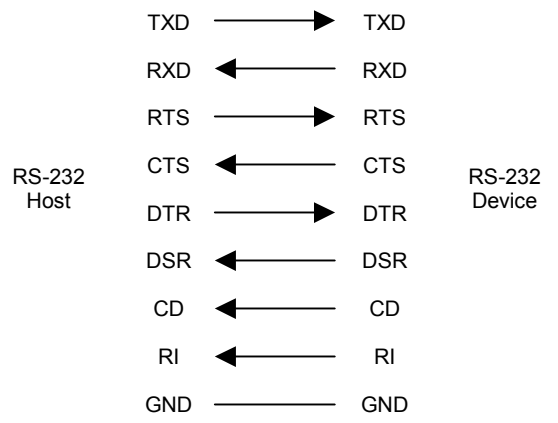

Figure 5. Typical RS-232 Signal Direction

The use of these signal lines is not restricted to the description given and in many applications the signals are either unused or used for a different purpose.

3.2 RS-232 Data Frames

When idle, the value on the data lines (TXD and RXD) is a ‘1’. To indicate the beginning of a transmission, a ‘0’ start bit is driven on the line for a bit time. The start bit is followed by multiple data bits (typically 5 to 8), beginning with the least significant bit and ending with the most significant bit. An error checking, or parity, bit is optionally appended to the end of the data. Finally, a ‘1’ stop bit is sent to indicate the end of the transmission. The duration of the stop bit can be 1 or 2 bit times. This places the line back in the idle state and available for a new transmission.

The duration of each bit depends on the baud rate. At a baud rate of 9600 the length of each bit is 0.1 milliseconds. At 57600 baud, this time is reduced to 1.7 microseconds.

Figure 6 illustrates a data frame with one start bit, eight data bits, a parity bit and one stop bit being sent at 9600 baud.

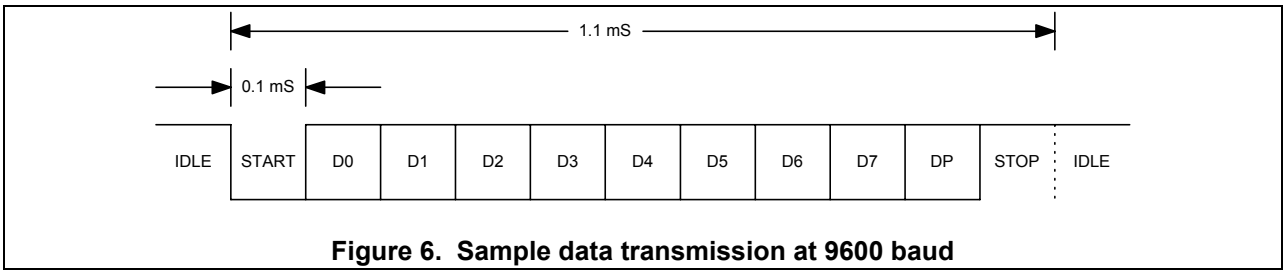

Figure 6. Sample data transmission at 9600 baud

3.3 Flow Control

Flow control is not required in many RS-232 applications. It is usually used whenever large amounts of data are being sent or received and it becomes necessary to temporarily disable data transmission and reception. Although there are several forms of flow control, the most commonly used method involves the use of RTS and CTS.

RTS is used to indicate whether the host is able to receive data or not, and CTS is used to indicate whether the device is able to receive data or not.

3.4 Error Checking

Error checking in RS-232 is an optional feature that is done by appending a parity bit to the end of each data transmission. Even parity means that the total number of "1s" in the data frame, including the parity bit, is an even number. Odd parity means that the total number is an odd number.

3.5 Electrical Characteristics

RS-232 has voltage levels that range from -3V to -15V for a logic 1, and +3V to +15V for a logic 0.

4. Design Discussion

The following sections discuss the details of the USB to Serial design implementation.

The basic premise behind this design is to give the microcontroller the functionality of an RS-232 port. The microcontroller will be able to send and receive data according to the RS-232 protocol as well as perform all the functions required for flow control, frame generation and parity generation and checking.

The microcontroller will also communicate with the host through USB, sending and receiving data, as well as configuring and communicating information about the serial communication settings.

4.1 Design Overview

The design enables the microcontroller to completely implement the functionality of an RS-232 port, with the ability to change settings like baud rate, parity type, number of data bits and number of stop bits. The application requires the ability to transfer three types of data to/from the device: control/status data, receive data and transmit data.

Control/status data is sent over the control pipe to Endpoint 0. Receive data is sent to the host from Endpoint 1. Transmit data is sent from the host to Endpoint 2. This data scheme is illustrated in Figure 7.

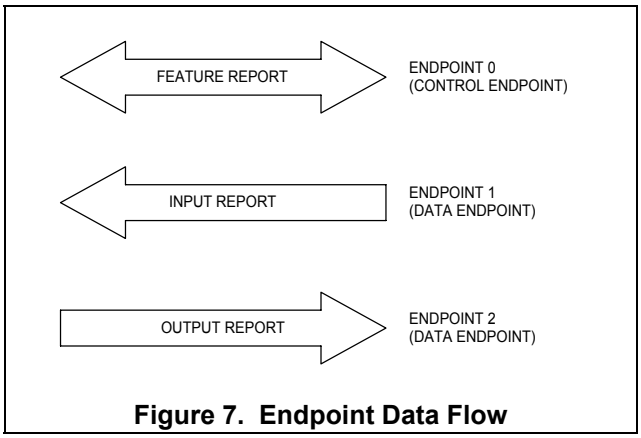

Figure 7. Endpoint Data Flow

4.2 Firmware Description

At the heart of the firmware are four routines: the **Main** routine which checks what tasks are pending and calls the appropriate routines, the **Rx_Data** routine with is called whenever a start bit is detected at the beginning of an RS-232 data reception, the **Tx_Data** routine which is called whenever data is

sent from the host which needs to be transmitted to the RS-232 device and the **Setup_Serial** routine which is called whenever a feature report is sent.

There are a number of event flags that the Main routine uses to detect the need for and schedule various handler routines. These event flags are shown in Figure 8.

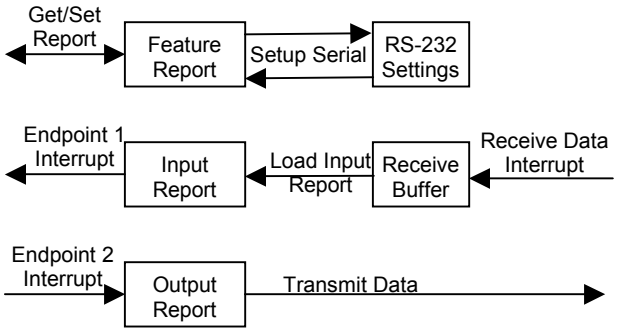

Figure 8. Application Data Flow

Variable	Description
setup_serial	The flag indicates that a feature report has been received over the control endpoint. When this variable is true, the Main routine calls the **Setup_Serial** routine, which configures the serial communication settings.
tx_pending	This flag indicates that data has been received from the host via EP2. When this variable is TRUE, the Main routine calls the **Tx_Data** routine to transmit the data out over the RS-232 interface.
load_ep1_fifo	Indicates that the input report previously loaded in the EP1 FIFO has been successfully transmitted to the host and that the firmware should build a new input report in the EP1 FIFO. When this variable is TRUE, the Main routine calls the **Load_Ep1_Fifo** routine, which builds a new input report in the EP1 FIFO.

4.3 Configuring the Serial Port

Endpoint 0 is the control endpoint and it is used for USB enumeration. The control endpoint is also used by the host to send and receive a 'feature report'. This report allows the host to control settings such as the baud rate, number of bits, and parity. The feature report can also be retrieved by the host to get the setting information about the device. The feature report is five bytes long. The first four bytes contain the value of the baud rate. The final byte contains the port setting information and is shown in Figure 9.

The Cypress COM port emulation driver begins communication with the device by sending a feature report that informs the microcontroller of the settings at which the application would like to communicate with the device.

This feature report is sent as a five byte HID Feature Report. The first four bytes of the report contain data about the baud rate. The last byte contains the rest of the information about of the port settings, and it is illustrated in Figure 10. This report is sent to Endpoint 0 using a Set Report request.

Byte	Function
Byte 0	Baud Rate Byte 0
Byte 1	Baud Rate Byte 1
Byte 2	Baud Rate Byte 2
Byte 3	Baud Rate Byte 3
Byte 4	Configuration Byte

Figure 9. Control Report

bit 7	bit 6	bit 5	bit 4	bit 3	Bit 2	bit 1	bit 0
Reset	0	Parity Type	Parity Enable	Stop Bits	0	Data Bits	

Figure 10. Configuration Byte

Field	Description
Reset	The Reset bit provides a mechanism for the host to reset the USB to Serial controller. All serial variables will be reset to their initial state.
Parity Type	The Parity Type field indicates whether the parity mode is odd or even. If Parity is not enabled (bit 4), this field has no effect. 1 = Odd Parity 0 = Even Parity Parity Type bit is '1', parity is odd; if Parity Type is '0', even parity is selected.
Parity Enable	The Parity Enable bit indicates whether parity is included with each data byte transfer. 1 = Enable Parity 0 = Disable Parity
Stop Bits	The Stop Bits field indicates the number of stop bits included at the conclusion of each byte transfer. 0 = One Stop Bit 1 = Two Stop Bits
Data Bits	The Data Bits field indicates the number of data bits included in each data byte transfer. Data Bit counts from 5 to 8 are supported. The number included in this field represents the number of data bits minus 5. (i.e., a count of 0 in this field selects 5 data bits; a count of 3 in this field selects 8 data bits.)

When Endpoint 0 receives the data it stores it in a buffer, and sets the value of the **setup_serial** flag to

TRUE. When the main routine detects the **setup_serial** flag it calls the **Setup_Serial** routine. The routine looks at the data sent in the feature report and sets the values of a number of variables accordingly.

Below is a list of these variables and what they are used for:

Variable	Description
bit_rate	This variable depends on the baud rate. It is used to determine the number of times that the receive and transmit routines repeat the software timing loops to achieve the correct baud rate. The following equation was used to determine the possible values of bit_rate: $\textbf{bit\_rate} = \frac{115200}{baud} - 1$ The baud rates that are supported are 57600, 38400, 19200, 9600, 4800, 2400, 1200, 600 and 300 bps.
data_bits	This holds the value of the word length.
tx_data_bits	This contains the total number of bits to be sent, including start, data, parity and stop bits.
parity_on	This indicates whether parity checking and generation are enabled (parity off = 0, parity on = 1).
parity_type	This indicates whether the parity generation and checking routines are based on odd or even parity (even = 0, odd = 1).

After the **Setup_Serial** routine sets these variables to their proper values, it then goes on to enable the microcontroller for serial communication. At this point the microcontroller is completely ready to perform serial communication at the baud rate and the port settings specified by the 'control byte'.

Receiving Serial Data from the Host

The host sends transmit data to the device by means of an output report over Endpoint 2. The format of an output report is described in Figures 11 and 12. (Figure 12 shows the 8-byte output report used with the CY7C63743 and Figure 11 shows the 32-byte output report used with the CY7C63413.) Figures 13 and 14 show the format of the control byte as well as a description of the fields used in this byte.

Byte	Description
0	Control
1	Transmit Byte Count
2	Transmit Data Byte 0
---	---
31	Transmit Data Byte 29

Figure 11. 32-Byte Output Report

Byte	Description
0	Control and Transmit Byte Count
1	Transmit Data Byte 0
---	---
7	Transmit Data Byte 6

Figure 12. 8-Byte Output Report

bit 7	bit 6	bit 5	bit 4	bit 3	Bit 2	bit 1	bit 0
0	0	DTR	RTS	Reset	0	0	0

Figure 13. Control Byte

Field	Description
DTR	The DTR bit provides a mechanism for the host to directly control the state of the DTR signal.
RTS	The RTS bit provides a mechanism for the host to directly control the state of the RTS signal.

Figure 14. Control Byte

Transmitting data to the device must be initiated by the host. The host begins by sending an output report to Endpoint 2. The output report causes an interrupt to occur on Endpoint 2 and the microcontroller vectors to the corresponding ISR. The Endpoint 2 ISR first checks if an ACK had occurred, signaling a successful data transfer. If the EP2 FIFO contains a valid output report, the number of data bytes that are being sent is saved. The values of the handshaking signals are also saved and written to Port 1. If the output report contains any transmit data (i.e., the count field is non-zero), the **tx_pending** flag is set and the EP2 mode is set to NAK_OUT. Alternatively, if the output report contains no transmit data, the ACK bit is cleared and the ISR simply exits.

4.4 Transmitting Serial Data to the Device

When the **tx_pending** flag is set, the **Main** routine calls the **Tx_Data** routine, which is responsible for transmitting the data to the device over the TXD line.

The **Tx_Data** routine is similar to the **Rx_Data** routine in the way that it performs the timing to send data at the correct baud rate. The routine begins by checking CTS and making sure that the device is ready to receive. If the device is not ready to receive, **Tx_Data** exits. If the device is ready to receive, **Tx_Data** fetches a byte of data from the output report and sends it out serially over the TXD line. In doing so, it creates the appropriate bit timing, generates the start bit, transmits the appropriate

number of data bits, optionally transmits the parity bit and finally, allows for the correct number of stop bits.

4.5 Receiving Serial Data from the Device

Pin 7 of Port 0 corresponds to the RXD line of an RS-232 port. To enable the microcontroller to receive data from the device, a GPIO interrupt with negative (falling edge) polarity is enabled for this pin. Whenever a falling edge occurs on this line, a GPIO interrupt occurs which causes the microcontroller to vector to the **Rx_Data** interrupt service routine.

The **Rx_Data** routine is responsible for reading the serial data at the correct rate. This is done through the use of software timing loops. The **Rx_Data** routine can be broken up into four parts.

The first section is a simple timing loop that stalls the processing for around .5 bit-times. This is used to ensure that the serial data is read at approximately the middle of the bit.

The second section reads the incoming data bits at the specified baud rate. Each bit is read from the RXD port pin and then placed in a register. The data is then shifted to the right in preparation for the next incoming bit. This process is repeated until all of the bits have been received. In the source code, the value enclosed by "[ ]" in the comment field indicates the number of CPU clock cycles it takes to execute that instruction. Note that it is critical that this routine not be modified unless the user has a clear understanding of the timing aspects of the routine.

The third section is used only when parity is enabled. This section basically looks like the previous one except the data is not placed in a memory location, it is only used to toggle the parity bit to check for errors.

The fourth and final section does a number of things. If the number of data bits is less than eight, it shifts the data value to the right to right-justify the data. It then checks if any parity errors had occurred. If an error did occur then it saves the location of the error. If no error had occurred, it goes on to save the data in a circular buffer from which data will be retrieved when it is to be sent to the host. Finally it checks whether the buffer is full, and if so it de-asserts RTS to signal to the device not to send any more data.

The circular data buffer, which is used to store the incoming data, is important to the proper processing of the data. There are two variables associated with this buffer. The first variable, called **rx_buffer_in** points to the location that the next data byte will be placed. The other variable, called **rx_buffer_out**, points to the next location that data will be retrieved from when the data is to be sent to the host. Every time new data is entered into the buffer, the **rx_buffer_in** variable is incremented. Every time data is removed from the buffer the **rx_buffer_out** variable is incremented. When both the variables are equal that means that the buffer is empty. When the value of **rx_buffer_in** is one less than **rx_buffer_out**, the buffer is full. This buffer could be of any size, depending on the needs of the application. The default is 32 bytes. Note that as it is currently implemented, the buffer size must be a power of 2 (i.e., the buffer must be 8, 16, 32 or 64 bytes).

The comments embedded in the code easily explain the flow and the function of the firmware. It is important to pay attention to the code at the end, which pertains to flow control. This code will not be included in the routine if the label **HW_FLOW_CONTROL** is not defined.

4.6 Sending Serial Data to the Host

The host periodically requests an input report from EP1 at a rate determined by the EP1 descriptor. For the CY7C64013 implementation, the default value is 1ms, which is the minimum for full-speed USB. The interval can be changed by simply changing the corresponding value in the EP1 descriptor.

When an IN request occurs, it generates an EP1 interrupt, which causes the microcontroller to vector to the **Ep1_Isr** interrupt service routine. The ISR first checks if an ACK had occurred which signals that the host has successfully removed the previous report from the EP1 FIFO. If the previous report has been successfully sent, the ISR sets the **load_ep1_fifo** flag to indicate to the **Main** routine that a new input report should be built in the EP1 FIFO.

The **Main** routine will detect the **load_ep1_fifo** flag and call the **Load_Ep1_Fifo** routine to create a new input report. The routine checks for data in the receive buffer. If there is any data available in the buffer, it fetches the data and loads it into the input report. If an error has occurred, no more data is

loaded into the input report and the ERROR flag is set accordingly. When the host requests the next IN transaction from EP1, the input report will be sent to the host.

The content and format of the input report is illustrated in Figures 15 and 16. Figure 17 and 18 provide details of the status byte and Figure 19 describes in detail the fields of the status byte.

Byte	Description
Byte 0	Status and Receive Count Byte
Byte 1	Receive Data Byte 0
---	---
Byte 7	Receive Data Byte 6

Figure 15. 8-Byte Input Report (CY7C63743)

Byte	Description
Byte 0	Status Byte
Byte 1	Receive Byte Count
Byte 2	Receive Data Byte 0
---	---
Byte 7	Receive Data Byte 29

Figure 16. 32-Byte Input Report (CY7C64013)

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
RI	CD	DSR	CTS	Error	Receive Count (0-7)		

Figure 17. Status and Receive Count Byte

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
RI	CD	DSR	CTS	Error	0	0	0

Figure 18. Status Byte

Field	Description
RI	The RI bit provides a method for the host to determine the state of the RI signal.
CD	The CD bit provides a method for the host to determine the state of the CD signal.
DSR	The DSR bit provides a method for the host to determine the state of the DSR signal.
CTS	The CTS bit provides a method for the host to determine the state of the CTS signal.
ERROR	The ERROR bit…
Receive Count	The Receive Count field provides an indication to the host of the number of received data bytes included in this report.

Figure 19. Status Field Descriptions

5. Design Options

5.1 Control Signals and Hardware Handshaking

Before proceeding to discuss the details of sending and receiving data it is important to discuss the use of the control signals and hardware handshaking. The firmware was written with the assumption that the design will be used with a device that supports RTS/CTS flow control. In cases where that is not true then changes to the firmware should be made.

In the case where handshaking is not used, the instructions in the firmware that pertain to handshaking should not be included. To simplify the task of enabling/disabling flow control, the firmware includes an assembler directive that will either enable or disable CTS/RTS flow control. To disable flow control simply comment out the DEFINE **HW_FLOW_CONTROL** line at the beginning of the code. To keep CTS/RTS handshaking enabled, keep that line as it is.

If the serial device implements another method of flow control then changes to the firmware must be made to support it. In most cases these changes will not be difficult to put in place but will require reading this application note thoroughly in order to gain a complete understanding of the firmware structure and how flow control fits into the picture.

It is also important to consider the initial condition of the handshaking signals prior to enumeration and before the control report comes in. When the device is powered on, a reset occurs and sets all bits on Port 1 to 0. The firmware must assign a value to Port 1 that will not cause any strange behavior or communicate any false information to the device.

5.2 Hardware Considerations

Interfacing the USB to Serial microcontroller with the serial device can be done two ways. The first option is to connect the signal lines of the microcontroller directly to the corresponding pins of the serial device at CMOS levels.

If this is not possible and it is necessary to interface at RS-232 levels then the voltage levels of the microcontroller need to be converted to and from RS-232. Converting the voltages can be done through the use a chip designed for this function (there are a number of them on the market) or by constructing a circuit to perform this using discrete

components. This will not be discussed in this application note.

Figure 20 illustrates how the 63743 can interface with a serial device at RS-232 voltages through the use of a level converter.

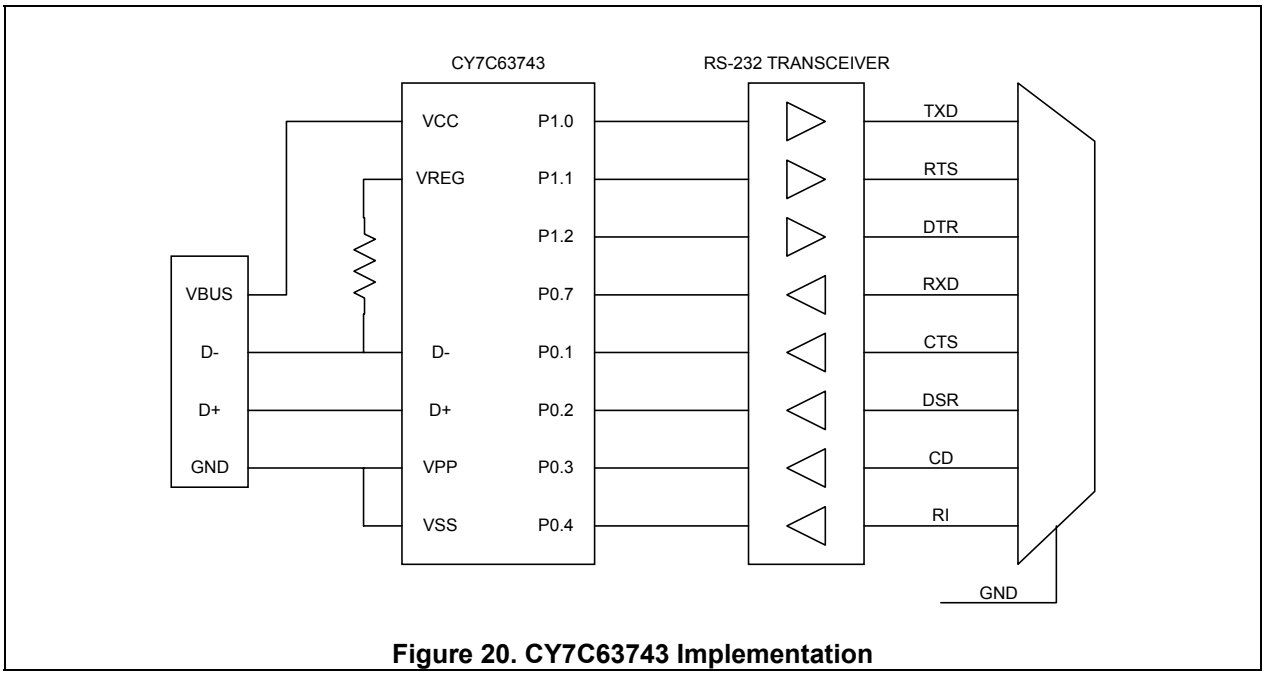

Figure 20. CY7C63743 Implementation

6. Customization

The driver supplied with this Reference Design was developed for Cypress Semiconductor by Industrial Computing Ltd. Industrial Computing is authorized to contract with Cypress customers to make changes or add additional features to the driver, provided that the resulting modified driver is redistributed exclusively for use with products containing Cypress USB devices.

Customers wishing to make changes to the driver themselves should contact Cypress Semiconductor, who may, in certain circumstances, for a fee, disclose the source code of the driver. Industrial Computing is not authorized to distribute the driver source code.

Industrial Computing may be contacted as follows:

Industrial Computing Ltd,
4 The Footpath,
Harston,
Cambridge,
CB2 5NS
UK

Tel: +44 1223 871646
Fax: +44 1223 870914
Email sales@indcomp.co.uk
Website http://www.indcomp.co.uk/

Appendix A – USB to Serial Driver Overview

Overview

This Appendix describes a Windows driver system that allows a USB device supporting the HID device class to appear to Windows applications as a standard serial port.

Driver structure

The core of the system is a single WDM driver 'HIDCOM.SYS'. This acts as a USB peripheral driver, connecting to the top of USBD.SYS, the standard system-provided USB driver. HIDCOM also contains code to emulate a serial port, as per the Windows NT/2000 specification for serial-port drivers. Although the peripheral follows the HID class specification, the HIDCOM driver does not use the system-provided HID class drivers – it is not possible to provide a Windows ME compatible HID client that uses the CCPORT mechanism described below.

On Windows NT/2000, this single driver is sufficient. On the Windows 9x/ME family, serial port drivers normally register with VCOMM.VXD (a system-provided driver) to inform the system of their presence. All user-mode communication with the serial port is then via VCOMM.VXD. Because there is no straightforward way for a WDM driver to register with VCOMM, Microsoft has provided a solution consisting of another WDM driver CCPORT.SYS and a VxD WDMMDMLD.VXD. These additional drivers allow a standard 'NT-style' WDM serial driver to appear as a serial port within Windows 9x. (Their intended function is actually the support of USB modems.)

Operating principles

General

When the emulated serial device is connected, the HID characteristics of the device (basically, the in, out and feature report sizes) are read and stored. These are obtained by directly requesting descriptors from the device. Operating system services are used to parse the HID descriptors.

When the device is opened, the driver creates three threads – read, write and config. All three of these threads are normally blocked, waiting for events. The threads are destroyed when the device is closed.

Data flow

The read and write threads handle the serial data flow. The read thread keeps a read pending on the USB driver all the time. The size of this read is based on the size of the IN report. Every time this read is satisfied, the data in the packet is transferred into an incoming FIFO buffer, then the read IRP is recycled. The emulated control lines are extracted from the packet at the same time. If there are any comm-events waiting, checks are made to see if these events can be satisfied. If there is no room in the incoming buffer, the data is discarded, and the queue-overrun error flag is set. After data has been received, attempts are made to satisfy any read IRPs that are queued by a user-mode application. An IRP which arrives when there is already sufficient data in the receive buffer is satisfied immediately.

The write thread sits blocked on an internal event. Whenever another part of the driver requires that an OUT packet is sent, this event is set, causing the write thread to initiate a USB write. This event is set whenever a write IRP is received or if the state of the configuration data is sent. A small write FIFO (100 bytes) is used to buffer data between write IRPs and the USB writing system – each write IRP's data is copied into the buffer, before being extracted by our USB write mechanism for transmission. This FIFO allows small writes (e.g. single characters, which are common) to be aggregated into the large packets. Each time a write is completed, the write thread checks if there is data in the write FIFO – if there is, another packet is constructed. Where an application writes more data than can be accommodated in the write FIFO, write IRPs are queued. The driver keeps track of the amount of data in queued write IRPs, only signaling TX_EMPTY when all IRPs have been satisfied.

IOCTLs

A large number of IOCTLs are defined for the standard serial driver. Where these involve changing control line states, a HID write is triggered (this may well include no other data, if no write transmission is waiting). Where they involve changes to the configuration (e.g. baud rate, word length, etc.), an event is set which triggers the config thread into transmitting a configuration packet. In the case of both control line changes and config changes, the operation is asynchronous – i.e. the driver completes the IRP immediately, quite possibly before the relevant data packet has been sent.

Document Revision History

Revision #	Date	Who	Comments
1.0B2	7/09/01	BEH	Initial document created for Beta 2 release
1.0	8/22/01	DGW	Driver overview added.