Commell MPX24794G Programmer's Guide

History

Revision	Date	Description	Author
1.0	08/19/2011	Firmware version: 0x02, 0x10	PT
1.1	06/21/2012	• Added configuring I/O section • Added Read/Modify/Write considerations • Added shadow registers • Added sample codes • Added more comments	PT

Contents

1 Hardware

The MPX24794G Module is a USB 2.0 compliant device, which implements 32 bits General Purpose Input/Output (GPIO) functions. Figure 1 depicts the MPX24794G USB device module. The MPX24794G USB device adopts Cypress CY8C24794 PSoC as the core component.

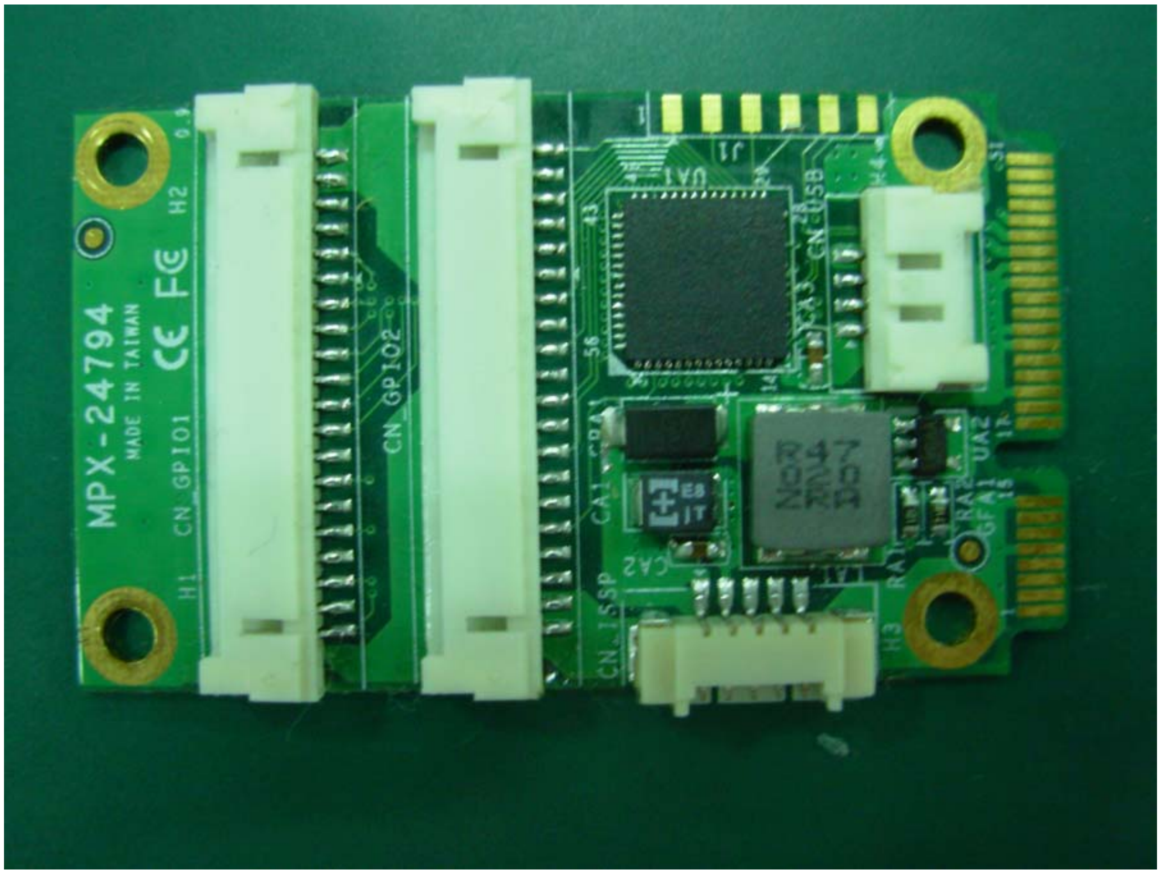
The GPIO contains input buffers, output drivers, register bit storage, and configuration logic for connecting the PSoC device to the outside world.

IO Ports are arranged with (up to) 8 bits per port. Each full port contains eight identical GPIO blocks, with connections to identify a unique address and register bit number for each block.

Features

- USB 2.0 Full Speed compliant
- 32-bit GPIO grouped into four 8-bit ports
- Digital IO (digital input and output controlled by software)
- Each IO pin also has up to 7 drive modes
- Each port can be individually programmed to either Input or Output
- Produced in Mini-PCIe card form factor (easily locked on motherboard)

Figure 1 MPX24794G USB Device Module



1.1 Cypress CY8C24794

The Cypress CY8C24794-24LXTI PSoC is the core component of MPX24794G USB-GPIO device.

1.2 General Purpose Input Output

The GPIO contains input buffers, output drivers, register bit storage, and configuration logic for connecting the PsoC device to the outside world.

IO Ports are arranged with (up to) 8 bits per port. Each full port contains eight identical GPIO blocks, with connections to identify a unique address and register bit number for each block. MPX-24794G uses Port 0, Port 2, Port 3, and Port 4 to implement 32-pin programmable general purpose IO.

Each IO pin also has up to seven drive modes that the user can program in order to meet the real world requirements.

The following drive modes are supported for your programming.

1. Mode 0: Resistive pull down
2. Mode 1: Strong drive
3. Mode 2: High impedance
4. Mode 3: Resistive pull up
5. Mode 4: Open drain, drives high
6. Mode 5: Slow strong drive
7. Mode 6: High impedance analog
8. Mode 7: Open drain, drives low

1.2.1 Digital IO

This section contains the text from the PSoC TRM, Document No. 001-14483 Rev. *F. Please refer to this manual for detail information.

One of the basic operations of the GPIO ports is to allow the M8C to send information out of the PSoC device and get information into the M8C from outside the PSoC device. This is accomplished by way of the port data register (PRTxDR). Writes from the M8C to the PRTxDR register store the data state, one bit per GPIO. In the standard non-bypass mode, the pin drivers drive the pin in response to this data bit, with a drive strength determined by the Drive mode setting. The actual voltage on the pin depends on the Drive mode and the external load.

The M8C can read the value of a port by reading the PRTxDR register address. When the M8C reads the PRTxDR register address, the current value of the pin voltage is translated into a logical value and returned to the M8C. Note that the pin voltage can represent a different logic value than the last value written to the PRTxDR register. This is an important distinction to remember in situations such as the use of a read modify write to a PRTxDR register. Examples of read modify write instructions include ***AND***, ***OR***, and ***XOR***.

The following is an example of how a read modify write, to a PRTxDR register, could have an unexpected and even indeterminate result in certain systems. Consider a scenario where all bits of Port 1 on the PSoC device are in the strong 1 resistive 0 drive mode; so that in some cases, the system the PSoC is in may pull up one of the bits.

```
mov     reg[PRT1DR], 0X00
or      reg[PRT1DR], 0X80
```

In the first line of code above, writing a 0x00 to the port will not affect any bits that happen to be driven by the system the PSoC is in. However, in the second line of code, it can not guarantee that only bit 7 will be the one set to a strong 1. Because the OR instruction will first read the port, any bits that are in the pull up state will be read as a '1'. These ones will then be written back to the port. When this happens, the pin will go into a strong 1 state; therefore, if the pull up condition ends in the system, the PSoC will keep the pin value at a logic 1.

1.2.2 Mapping

MPX-24794G maps the PRTxDR registers to the outside GPIO as the following table.

Table 1 PRTxDR/GPIO mapping

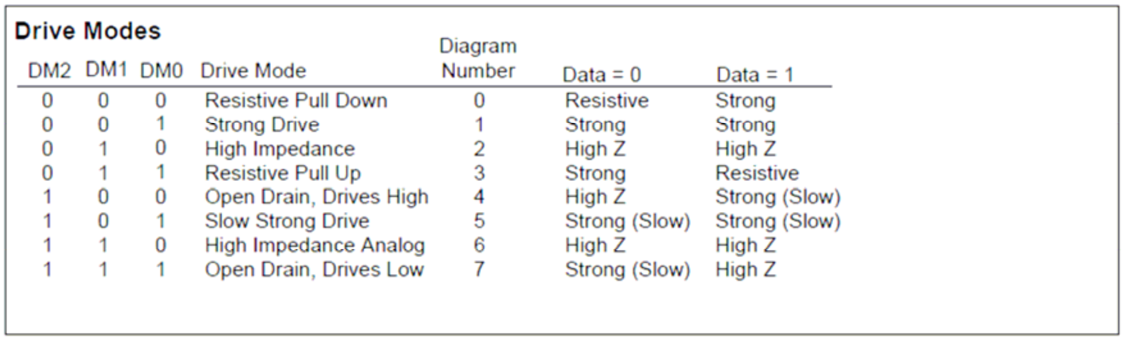
PRTxDR	GPIO	Description
PRT0DR[7..0]	GPIO[7..0]	Each bit of the Port_0 is connecting to each GPIO pin 7 to pin 0 correspondingly.
PRT2DR[7..0]	GPIO[15..8]	Each bit of the Port_2 is connecting to each GPIO pin 15 to pin 8 correspondingly.
PRT3DR[7..0]	GPIO[23..16]	Each bit of the Port_3 is connecting to each GPIO pin 23 to pin 16 correspondingly.
PRT4DR[7..0]	GPIO[31..24]	Each bit of the Port_4 is connecting to each GPIO pin 31 to pin 24 correspondingly.

Please be noted that writes from the M8C to the PRTxDR register store the data state, one bit per GPIO. In the standard non-bypass mode, the pin drivers drive the pin in response to this data bit with a drive strength determined by the Drive mode setting (see Drive Mode section). The actual voltage on the pin depends on the Drive mode and the external load.

1.3 Drive Mode

Please refer to "6.2.4 PRTxDMx Registers" (page 105) of the CY8C24x94 PSoC Technical Reference Manual" for detail information about the drive mode of GPIO ports.

Figure 2 Drive Modes



Drive Modes

DM2	DM1	DM0	Drive Mode	Diagram Number	Data = 0	Data = 1
0	0	0	Resistive Pull Down	0	Resistive	Strong
0	0	1	Strong Drive	1	Strong	Strong
0	1	0	High Impedance	2	High Z	High Z
0	1	1	Resistive Pull Up	3	Strong	Resistive
1	0	0	Open Drain, Drives High	4	High Z	Strong (Slow)
1	0	1	Slow Strong Drive	5	Strong (Slow)	Strong (Slow)
1	1	0	High Impedance Analog	6	High Z	High Z
1	1	1	Open Drain, Drives Low	7	Strong (Slow)	High Z

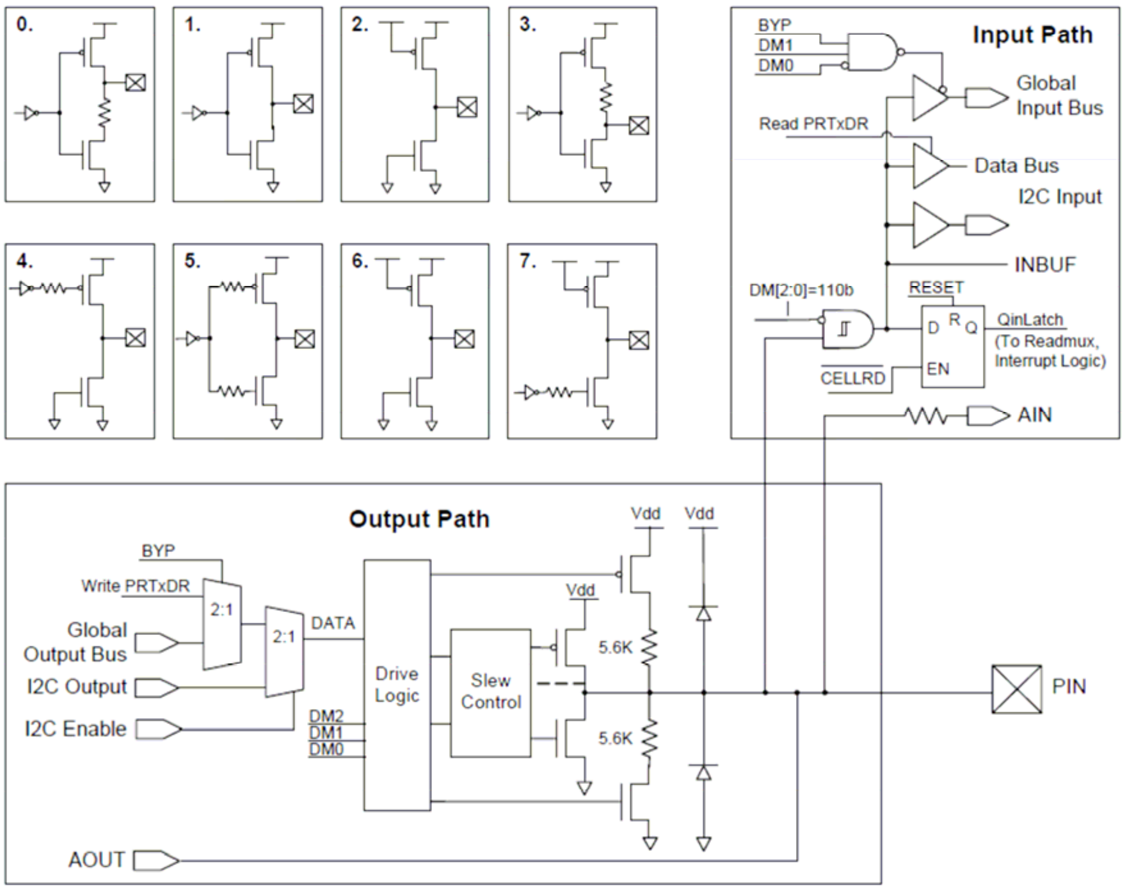
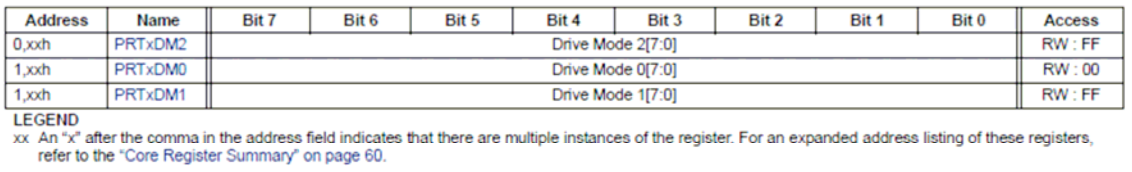


Figure 6-1. GPIO Block Diagram

1.4 PRTxDMx Registers



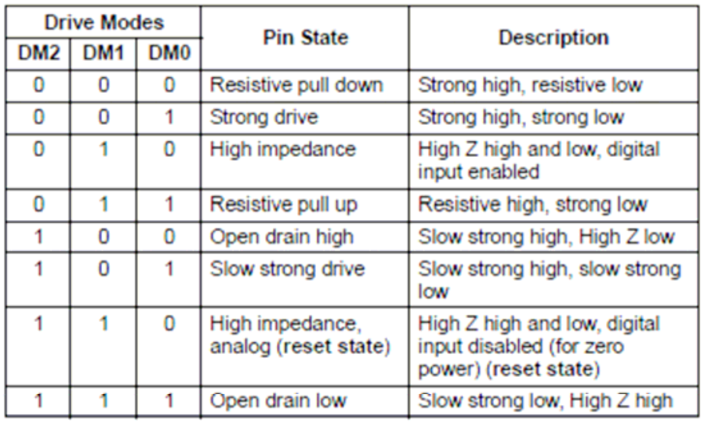
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Access
0,xxh	PRTxDM2	Drive Mode 2[7:0]								RW : FF
1,xxh	PRTxDM0	Drive Mode 0[7:0]								RW : 00
1,xxh	PRTxDM1	Drive Mode 1[7:0]								RW : FF

LEGEND
xx An "x" after the comma in the address field indicates that there are multiple instances of the register. For an expanded address listing of these registers, refer to the "Core Register Summary" on page 60.

The Port Drive Mode Bit Registers (PRTxDMx) are used to specify the Drive mode for GPIO pins.

Bits 7 to 0: Drive Mode x[7:0]. In the PRTxDMx registers there are eight possible drive modes for each port pin. Three mode bits are required to select one of these modes, and these three bits are spread into three different registers (PRTxDM0, PRTxDM1, and PRTxDM2). The bit position of the effected port pin (for example, Pin[2] in Port 0) is the same as the bit position of each of the three drive mode register bits that control the Drive mode for that pin (for example, bit[2] in PRT0DM0, bit[2] in PRT0DM1, and bit[2] in PRT0DM2). The three bits from the three registers are treated as a group. These are referred to as DM2, DM1, and DM0, or together as DM[2:0]. Drive modes are showed in the following table.

Table 2 Pin Drive Modes



Drive Modes			Pin State	Description
DM2	DM1	DM0		
0	0	0	Resistive pull down	Strong high, resistive low
0	0	1	Strong drive	Strong high, strong low
0	1	0	High impedance	High Z high and low, digital input enabled
0	1	1	Resistive pull up	Resistive high, strong low
1	0	0	Open drain high	Slow strong high, High Z low
1	0	1	Slow strong drive	Slow strong high, slow strong low
1	1	0	High impedance, analog (reset state)	High Z high and low, digital input disabled (for zero power) (reset state)
1	1	1	Open drain low	Slow strong low, High Z high

For analog IO, the Drive mode should be set to one of the High Z modes, either 010b or 110b. The 110b mode has the advantage that the block's digital input buffer is disabled, so no ***crowbar*** current flows even when the analog input is not close to either power rail. When digital inputs are needed on the same pin as analog inputs, the 010b Drive mode should be used. If the 110b Drive mode is used, the pin will always be read as a zero by the CPU and the pin will not be able to generate a useful interrupt. (It is not strictly required that a High Z mode be selected for analog operation.)

For global input modes, the Drive mode must be set to 010b.

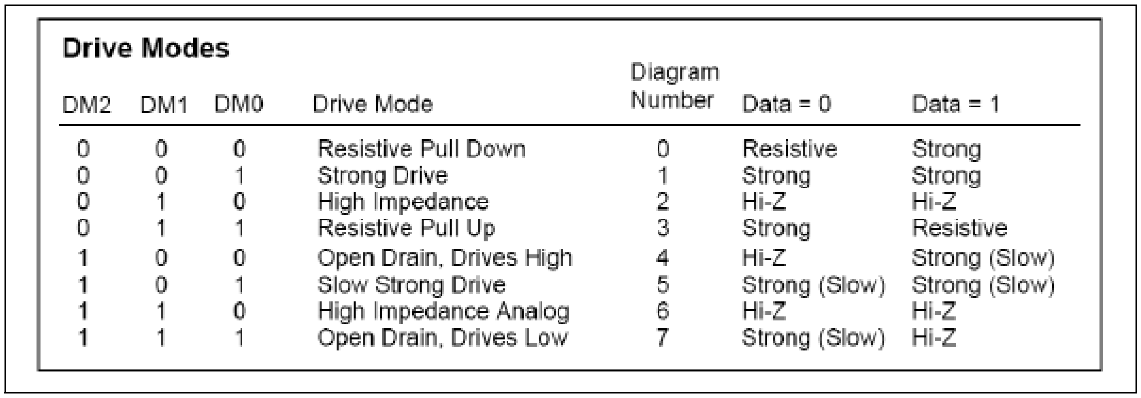
The GPIO provides a default Drive mode of high impedance, analog (High Z). This is achieved by forcing the reset state of all PRTxDM1 and PRTxDM2 registers to FFh.

The resistive drive modes place a ***resistance*** in series with the output, for low outputs (mode 000b) or high outputs (mode 011b). Strong Drive mode 001b gives the fastest edges at high DC drive strength. Mode 101b give the same drive strength but with slow edges. The open-drain modes (100b and 111b) also use the slower edge rate drive. These modes enable open-drain functions such as I2C mode 11b (although the slow edge rate is not slow enough to meet the I2C fast mode specification).

1.5 Configuring I/O

The PSoC Designer has a panel that allows you to select the 'Drive" mode on each pin. Typical selection are **High Z, High Z Analog, Open Drain High, Open Drain Low, Pull Down, Pull Up, Strong,** and **Strong Slow**.

Figure 3 Drive Modes (from TRM)

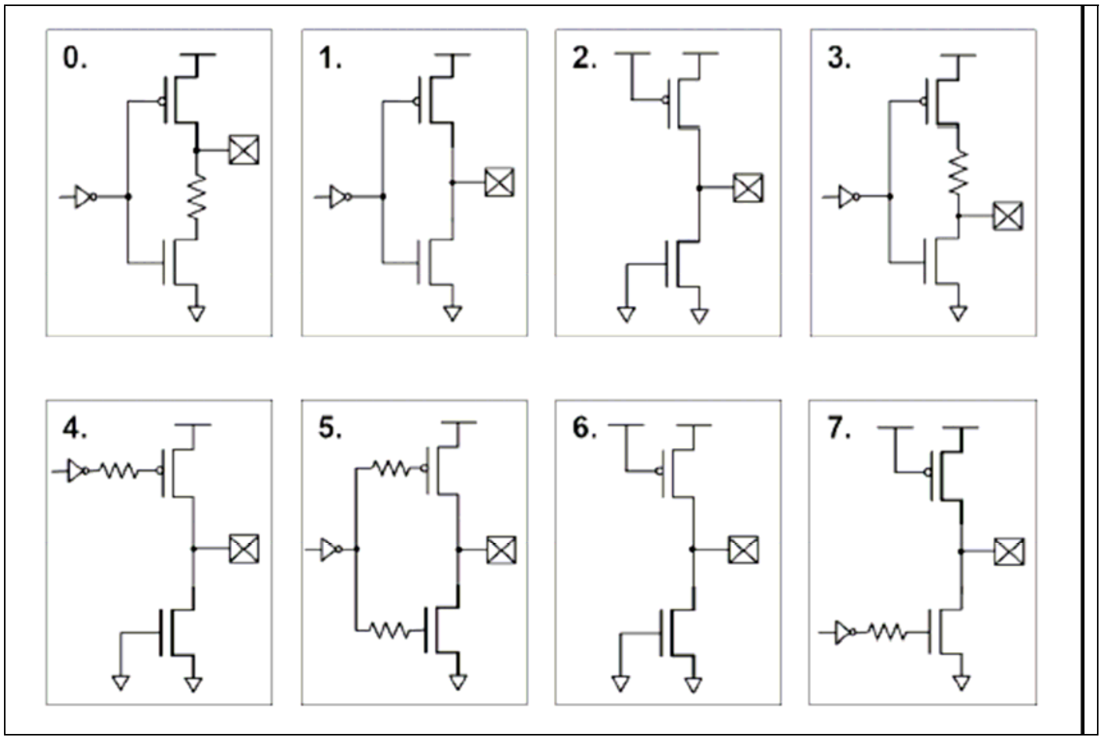


Drive Modes

DM2	DM1	DM0	Drive Mode	Diagram Number	Data = 0	Data = 1
0	0	0	Resistive Pull Down	0	Resistive	Strong
0	0	1	Strong Drive	1	Strong	Strong
0	1	0	High Impedance	2	Hi-Z	Hi-Z
0	1	1	Resistive Pull Up	3	Strong	Resistive
1	0	0	Open Drain, Drives High	4	Hi-Z	Strong (Slow)
1	0	1	Slow Strong Drive	5	Strong (Slow)	Strong (Slow)
1	1	0	High Impedance Analog	6	Hi-Z	Hi-Z
1	1	1	Open Drain, Drives Low	7	Strong (Slow)	Hi-Z

The PSoC Technical Reference Manual (Section 6.1, PSoC TRM, Document No. 001-14463 Rev. *F) provides the Drive Modes chart shown above and also a Diagram representation of the mode shown below. The 'Drive Modes' selection chart details the various modes. The first three columns indicate the settings for the three ports configuration registers PRTxDMx. The two columns on the right indicate the required data output state of register PRTxDR.

Figure 4 Diagram of port pin configuration.



The eight diagrams help describe the pin configuration. The top of each diagram represents the Supply voltage while the bottom is Ground. The box with the X in on the right represents the pin, while the line or resistor on the left represents the PRTxDR output register bit. When the PRTxDR output is set to a logical 0, this activates the low side switch. When set to a logical 1, the high side switch is active. For the simple case of a 'Strong' output, Diagram 1, setting the PRTxDR bit to a 1 activates the high side switch driving the pin to Vdd. When a 0 is written to the PRTxDR bit the low side switch activates, driving the pin to ground. If a resistor is shown attached to the input of the switch then a 'Slow' slew rate mode is used.

Refer to the two columns in the Drive Modes table above ('Data=0', 'Data=1') to determine the active state and to help clarify the Diagrams.

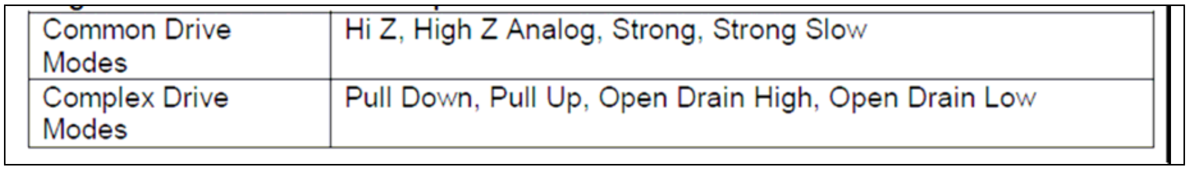
Figure 5 - Summary of Drive Modes and PRTxDR output register settings

Mode	Description
0. Pull Down	Write 1 to PRTxDR activates the high side switch driving the pin to Vdd. Write 0 to PRTxDR activates the low side switch driving the pin to Ground through the 5.6k ohm internal resistor.
1. Strong	Write 1 to PRTxDR activates the high side switch driving the pin to Vdd. Write 0 to PRTxDR activates the low side switch driving the pin to Ground.
2. High Z	Both high and low side switch deactivated. The pin floats in a high impedance state. Writing a 1 or 0 to PRTxDR has no effect.

3. Pull Up	Writing 1 to PRTxDR activates the high side switch driving the pin to Vdd through the 5.6k ohm internal resistor. Writing 0 to PRTxDR activates the low side switch driving the pin to Ground.
4. Open Drain High	Writing 1 to PRTxDR activates the high side switch driving the pin to Vdd. Writing 0 to PRTxDR deactivates the high side switch, the low side switch is always deactivated, the output pin floats in a high impedance.
5. Strong Slow	Same as case 1 but without the slew control speed up circuitry. It will not snap on or off as fast. There is a one clock period delay compared to the case 1. Strong Mode.
6. High Z Analog	Same as case 2. The pin is disconnected from the digital input circuitry and is available for analog use only.
7. Open Drain Low	Writing 1 to PRTxDR deactivates the low side switch, the high side switch is always deactivated, the output pin floats in a high impedance state. Writing a 0 to PRTxDR activates the low side switch driving the pin to Ground.

Please be noted that the startup code does not initialize the PRTDRx output register. For the most common drive modes of 'Hi Z', 'Hi Z Analog', 'Strong' and 'Strong Slow' the initialization of the PRTDRx output port does not alter the pin configuration. Notice in the 'Drive Modes' chart, the configuration is the same independent of the PRTDRx output register (Data=0, Data=1 columns) for these common drive modes. If a Pull Up/Down or Open Drain High/Low is used then the initialization of the PRTDRx output register is needed to select the desired mode, we will call these modes '**Complex Drive Modes'**.

Figure 6 Common and Complex Drive Modes



Common Drive Modes	Hi Z, High Z Analog, Strong, Strong Slow
Complex Drive Modes	Pull Down, Pull Up, Open Drain High, Open Drain Low

1.6 Read/Modify/Write Considerations

The Technical Reference Manual Section 6.1 warns against using read/modify/write instructions on the PRTxDR registers or any write only register.

The read/modify/write instructions include: AND, OR, XOR. In C code it would look like this:

```
PRT1DR |= 1; /* turn pin 0 on */
PRT1DR &= 1; /* turn pin off */
```

```
PRT1DR ^= 1; /* toggle pin 0 */
```

This will translate to a single read/modify/write instruction in assembly:

```
OR   reg[PRT1DR], 1
AND  reg[PRT1DR], 1
XOR  reg[PRT1DR], 1
```

When we perform these actions on the PRTxDR output register, we essentially perform a read of all pin states, make the change and then write out the value to all pins on the port. While we can do this and it works in most of the common drive mode cases, it can be problematic as the pin state we read back does not necessarily reflect the last value written. This is especially true when using the complex drive modes.

1.6.1 The Simple Cases

In many cases our digital IO will be a simple case that does not require a lot of consideration. So for example if we have only Strong outputs and Hi-Z inputs on a set of port pins, then the read/modify/write operations will generally do what you want them to do unless they are shorted or overloaded. Any simple Hi-Z input pins will retain the Hi-Z mode without regard to what is written to the PRTxDR output register. One could also maintain and use a private shadow register of the output port state and use a simple write operation to transfer it. The point is that it is completely valid in most cases to write code that turns on a IO output bit using a read/modify/write operation like "PRTxDR |=1;". At the same time, it is important to understand the sharing issues involved so you recognize when a problem case exists. So for example if you mixed the above direct read/modify/write use above the LED user module on the same port, then you have a serious potential problem: the LED user module is using a shadow register and your other code is not, resulting in the outputs being set to something you do not expect. Solutions would be to use the LED user module for all outputs, learn how to use the common shadow register that the LED user module creates, or take out the LED user module and perform all IO with the read/modify/write or private shadow register.

The trick is to understand when you have a situation that is a tougher case and requires serious effort to coordinate a sharing solution on a particular port. You may discover this the hard way in development where your outputs or inputs do not function as expected. Worst case is you get a design finished and into production and then get occasional malfunctions due to overlooking these sharing issues.

1.6.2 Shared Outputs and Complex Mode Inputs

If you have any shared independent outputs on any single port, then a sharing solution must be implemented to coordinate writes to the port. If you have any inputs that use the Complex Drive Modes, then this must be coordinate as well since the mode depends on the value of the output port. If you have any outputs including hardware modules that use the Complex Drive Mode then a sharing solution must be implemented to retain the correct mode.

1.6.3 Shadow Registers Solution

While performing read/modify/write operations on an output register is convenient and an acceptable sharing solution is most cases, in a general sense it is problematic and could result in unexpected behavior. This is especially true if we wish to define software libraries that operate on a subset of arbitrary pins independent of the pins configurations.

A solution implemented by the PSoC Designer for the software user modules that operate on output pins is to define a RAM variable that retains the output state called a **shadow register**. All modifications to the output port are performed on this variable and then a simple write operation is use to transfer this to the output port register. If your application code needs to share the output register of a port and you also use one of these PSoC Designer User Modules (LED, LCD, I2Cm, LED7SEG), then it is required to use the shadow register associated with the port that the PSoC Designer generates. To change the state of an output pin, we make changes to this variable and then perform a write to the port.

For example, if we use an I2Cm and configure it to use P1_2, P1_3, we will find if we look at the code for this module(psocgpioint.h) it exports a variable called Port_1_Data_SHADE:

```
extern BYTE Port_1_Data_SHADE;
```

Since any output on Port1 needs to have a single coordinated solution (there can be only one shadow register), we need to use this shadow register:

```
if (PRT1DR & 2) {
   // Switch is turned on
   Port_1_Data_SHADE |= 1; // turn Light bit on
   PRT1DR = Port_1_Data_SHADE; // write port output
} else {
   // Switch is turned off
   Port_1_Data_SHADE &= ~1; // turn Light bit off
   PRT1DR = Port_1_Data_SHADE; // write port output
}
```

The PSoC Designer LED user module provides a convenient way to work with pin outputs and it takes care of the shadow register handling:

```
if (PRT1DR & 2) {
   // Switch is turned on
   LED_1_On(); // turn the Light On
} else {
   // Switch is turned off
   LED_1_Off(); // turn the Light off
}
```

We could rename this user module label in the Designed to 'Light' so that our code would read more naturally:

```
if (PRT1DR & 2) {
   // Switch is turned on
   Light_On(); // turn the Light on
} else {
   // Switch is turned off
   Light_Off(); // turn the Light off
}
```

A useful resource that explains Shadow Register use and more is contained in the PSoC Designer,
Menu->Help->Documentation->IDE Users Guide.pdf, starting at section 7.1.4.

1.6.4 Using Complex Drive Modes

If we have inputs or outputs that use the complex drive modes (Pull Up/Down, Open Drain) then we may need to initialize the output register to obtain the desired state. The PSoC Designer does not generate initialization code to perform this.

The default state of the output register on reset is 0. For this case, no initialization of PRTxDR is required for an input using Pull Down. Another case where no initialization of PRTxDR is needed is for an output that is configured as Open Drain High and the desired default state is off (High Impedance).

1.6.5 Reset State and Configuration

The reset state of port registers for the most common PSoC parts covered by the TRM are as follows:

```
PRTxDM0=00H, PRTxDM1=FFH, PRTxDM2=FFH, PRTxDR=00H.
```

This corresponds to the High Z Analog drive mode as the default state on reset.

Any attached hardware should be arranged to transition through the reset stage given a High Z pin state for the short duration of startup and reset. This may require external resistors (pull-ups/pull-downs) if the hardware is sensitive during this short transition period.

2 Firmware Guide

This section describes the firmware that runs on the CY8C24794 controller. This MPX-24794G firmware implements the command / response protocol supporting the communication between the USB host and MPX-24794G card.

2.1 Descriptors

This paragraph defines the following information of the MPX-24794G USB device.

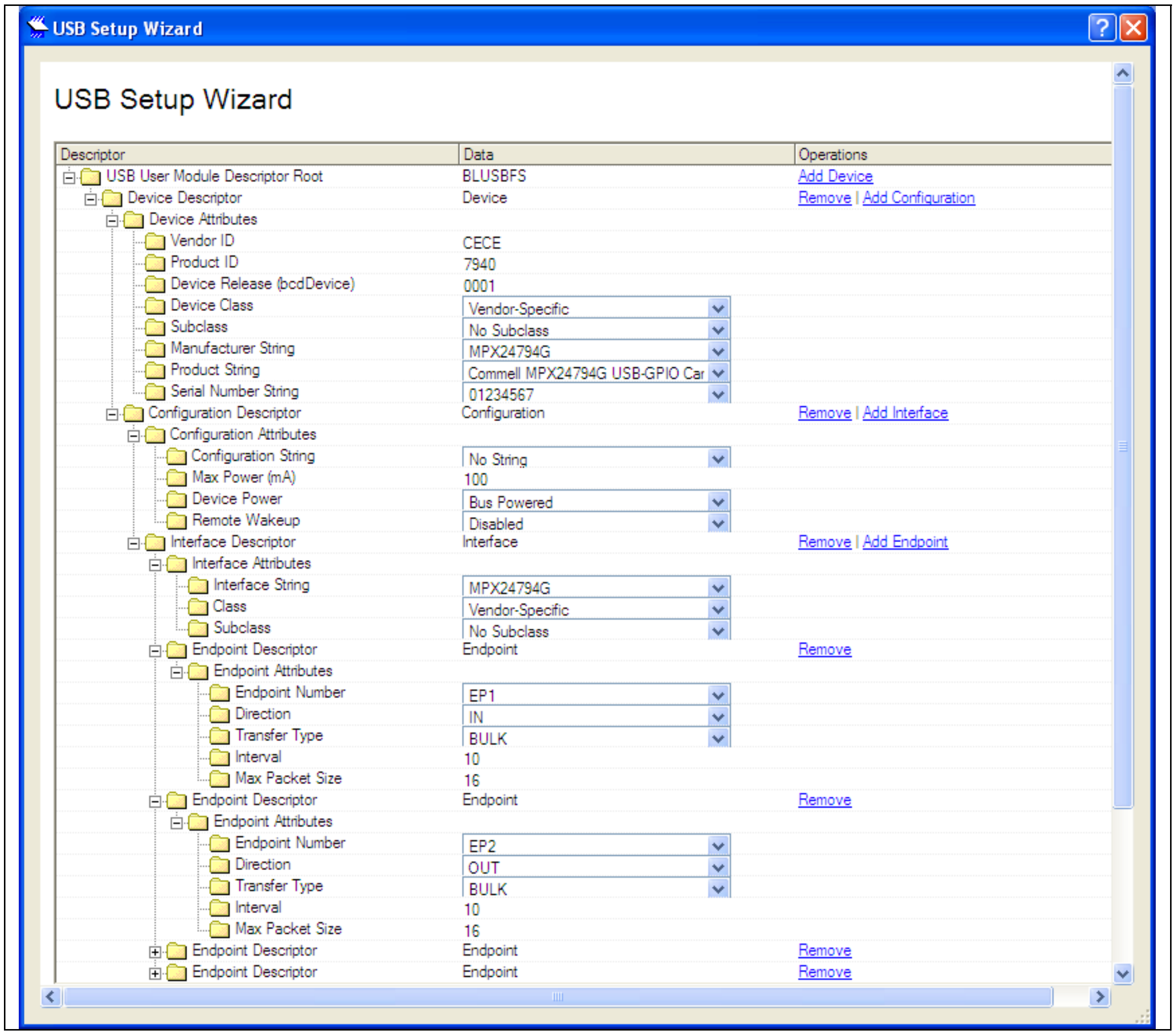
- Device Attributes
- Configuration Descriptor
- Configuration Attributes
- Interface Descriptors
- Interface Attributes
- Endpoint Descriptors

2.1.1 Device Descriptors

Vendor ID = 0xCECE
Product ID = 0x7940

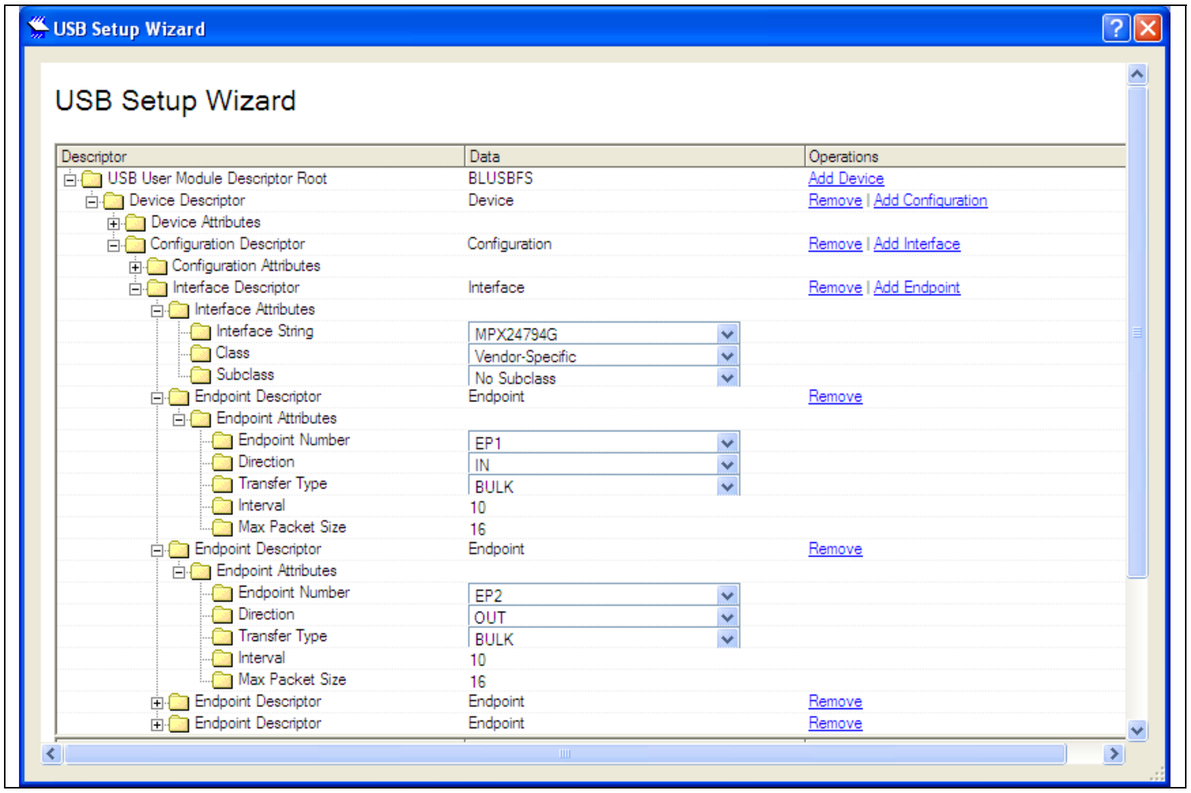
VID_CECE&PID_7940

Figure 7 MPX-24794G Device Descriptors



USB Setup Wizard

Descriptor	Data	Operations
USB User Module Descriptor Root	BLUSBFS	Add Device
Device Descriptor	Device	Remove | Add Configuration
Device Attributes		
Vendor ID	CECE	
Product ID	7940	
Device Release (bcdDevice)	0001	
Device Class	Vendor-Specific	
Subclass	No Subclass	
Manufacturer String	MPX24794G	
Product String	Commell MPX24794G USB-GPIO Car	
Serial Number String	01234567	
Configuration Descriptor	Configuration	Remove | Add Interface
Configuration Attributes		
Configuration String	No String	
Max Power (mA)	100	
Device Power	Bus Powered	
Remote Wakeup	Disabled	
Interface Descriptor	Interface	Remove | Add Endpoint
Interface Attributes		
Interface String	MPX24794G	
Class	Vendor-Specific	
Subclass	No Subclass	
Endpoint Descriptor	Endpoint	Remove
Endpoint Attributes		
Endpoint Number	EP1	
Direction	IN	
Transfer Type	BULK	
Interval	10	
Max Packet Size	16	
Endpoint Descriptor	Endpoint	Remove
Endpoint Attributes		
Endpoint Number	EP2	
Direction	OUT	
Transfer Type	BULK	
Interval	10	
Max Packet Size	16	
Endpoint Descriptor	Endpoint	Remove
Endpoint Descriptor	Endpoint	Remove

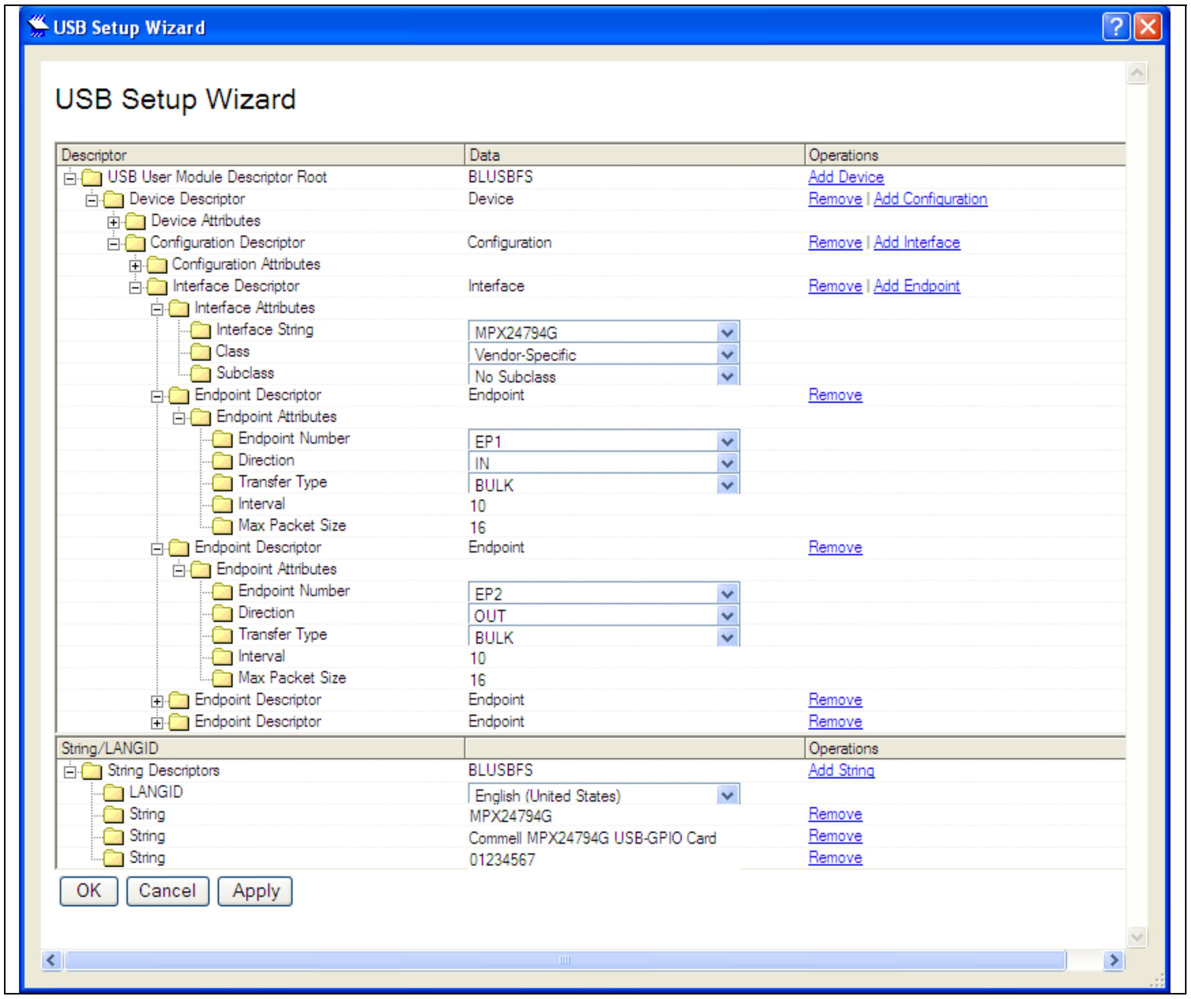


USB Setup Wizard

Descriptor	Data	Operations
USB User Module Descriptor Root	BLUSBFS	Add Device
Device Descriptor	Device	Remove | Add Configuration
Device Attributes		
Configuration Descriptor	Configuration	Remove | Add Interface
Configuration Attributes		
Interface Descriptor	Interface	Remove | Add Endpoint
Interface Attributes		
Interface String	MPX24794G	
Class	Vendor-Specific	
Subclass	No Subclass	
Endpoint Descriptor	Endpoint	Remove
Endpoint Attributes		
Endpoint Number	EP1	
Direction	IN	
Transfer Type	BULK	
Interval	10	
Max Packet Size	16	
Endpoint Descriptor	Endpoint	Remove
Endpoint Attributes		
Endpoint Number	EP2	
Direction	OUT	
Transfer Type	BULK	
Interval	10	
Max Packet Size	16	
Endpoint Descriptor	Endpoint	Remove
Endpoint Descriptor	Endpoint	Remove

2.1.2 String Descriptors

Figure 8 MPX-24794G String Descriptors



USB Setup Wizard

Descriptor	Data	Operations
USB User Module Descriptor Root	BLUSBFS	Add Device
Device Descriptor	Device	Remove | Add Configuration
Device Attributes		
Configuration Descriptor	Configuration	Remove | Add Interface
Configuration Attributes		
Interface Descriptor	Interface	Remove | Add Endpoint
Interface Attributes		
Interface String	MPX24794G	
Class	Vendor-Specific	
Subclass	No Subclass	
Endpoint Descriptor	Endpoint	Remove
Endpoint Attributes		
Endpoint Number	EP1	
Direction	IN	
Transfer Type	BULK	
Interval	10	
Max Packet Size	16	
Endpoint Descriptor	Endpoint	Remove
Endpoint Attributes		
Endpoint Number	EP2	
Direction	OUT	
Transfer Type	BULK	
Interval	10	
Max Packet Size	16	
Endpoint Descriptor	Endpoint	Remove
Endpoint Descriptor	Endpoint	Remove

String/LANGID		Operations
String Descriptors	BLUSBFS	Add String
LANGID	English (United States)	
String	MPX24794G	Remove
String	Commell MPX24794G USB-GPIO Card	Remove
String	01234567	Remove

2.2 USB Vendor Specific Commands

The MPX24794G GPIO-USB card supports vendor specific commands. Users use these commands to get some information.

2.2.1 Vendor Device Commands

2.2.1.1 Read Firmware Version

Vendor IN command: Read Firmware Version
bmRequestType = 0x00
bRequest = 0x00
wValue = don't care

wIndex = don't care
wLength = 0x0002

2.3 Programming Firmware

This section describes the MPX-24794G firmware implementation and how to programming the firmware via USB transactions.

2.3.1 Command Code vs. Response Code

MPX-24794G firmware designs command code and response code in pair except the first bulk in transaction, which is trying to read back to firmware ready information.

The USB Bulk Out is implemented in Endpoint 2. This EP2 is used to take the commands issued from the application resides in the USB host side. Each command is denoted by a unique byte.

The firmware takes action according to the command packet from the USB host side. The firmware then prepares the return result packet after the completion of the requested command.

The USB Bulk In is implemented in Endpoint 1. This EP1 is used to return requested result back to the USB host. The application should therefore issue a bulk in transaction in order to retrieve back the execution result of the prior bulk out transaction.

The result packet is composed according to its corresponding command packet. Each command code has its own unique response code. The application should check the response code according to the command code.

The following table shows the command codes and its response codes defined in the MPX-24794G firmware.

Table 3 Command Code vs. Response Code

Name	Code	Description
CMD_READ_VERSION	0x00	
RSP_READ_VERSION	0x01	
CMD_READ_GPIO_7_0	0x02	
RSP_READ_GPIO_7_0	0x03	
CMD_READ_GPIO_15_8	0x04	

RSP_READ_GPIO_15_8	0x05	
CMD_READ_GPIO_23_16	0x06	
RSP_READ_GPIO_23_16	0x07	
CMD_READ_GPIO_31_24	0x08	
RSP_READ_GPIO_31_24	0x09	
CMD_READ_GPIO_31_0	0x0A	
RSP_READ_GPIO_31_0	0x0B	
CMD_WRITE_GPIO_7_0	0x10	
RSP_WRITE_GPIO_7_0	0x11	
CMD_WRITE_GPIO_15_8	0x12	
RSP_WRITE_GPIO_15_8	0x13	
CMD_WRITE_GPIO_23_16	0x14	
RSP_WRITE_GPIO_23_16	0x15	
CMD_WRITE_GPIO_31_24	0x16	
RSP_WRITE_GPIO_31_24	0x17	
CMD_WRITE_GPIO_31_0	0x18	
RSP_WRITE_GPIO_31_0	0x19	
CMD_READ_PRTxDMx	0x20	
RSP_READ_PRTxDMx	0x21	
CMD_WRITE_PRTxDMx	0x22	
RSP_WRITE_PRTxDMx	0x23	
CMD_READ_PRTxIE	0x30	
RSP_READ_PRTxIE	0x31	
CMD_WRITE_PRTxIE	0x32	
RSP_WRITE_PRTxIE	0x33	
CMD_READ_PRTxGS	0x34	
RSP_READ_PRTxGS	0x35	
CMD_WRITE_PRTxGS	0x36	
RSP_WRITE_PRTxGS	0x37	
CMD_READ_PRTxICx	0x38	
RSP_READ_PRTxICx	0x39	
CMD_WRITE_PRTxICx	0x3A	
RSP_WRITE_PRTxICx	0x3B	
CMD_SET_DRIVE_MODE	0x40	
RSP_SET_DRIVE_MODE	0x41	
RSP_READY	0xFD	
CMD_INVALID_COMMAND	0xFE	
RSP_INVALID_COMMAND	0xFF	

2.3.2 Formats

Table 4 Bulk Out (EP2) Command Format

Offset	Name	Description
+ 0	Command code	Mandatory
+ 1	Parameter 0	Optional
+ 2	Parameter 1	Optional
+ 3	Parameter 2	Optional
…	…	…
+ n	Parameter n	Optional

Notes:

1. Up to 16 bytes in a bulk out transaction

Table 5 Bulk In (EP1) Response Format

Offset	Name	Description
+ 0	Response code	See Response Code Table
+ 1	Length	Whole packet length
+ 2	Error code	See Error Code Table
+ 3	Return 0	Optional
+ 4	Return 1	Optional
+ 5	Return 2	Optional
…	…	Optional
+ n	Checksum	Checksum of the whole packet

Notes:

1. Checksum is the arithmetic-negation of the sum of the whole packet.
2. The whole packet is from the response code to the checksum byte.
3. Length is the number of bytes of the whole packet.
4. The sum of the whole packet should be equal to zero in low byte.
5. Example (in hex): FD 07 0 55 AA 02 FB.
6. Read in 16 bytes for each bulk in transaction.

2.3.2.1 Initial Bulk In Content

The Bulk In buffer contains the following data after the USB bus reset.

Offset	Name	Description
+ 0	0xFD	
+ 1	0x07	
+ 2	Error Code	
+ 3	0x55	
+ 4	0xAA	

+ 5	Version	Firmware version
+ 6	Checksum	

2.3.2.2 CMD_READ_FIRMWARE

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x00	Read firmware version command

Bulk In (EP1)

Offset	Name	Description
+ 0	0x01	Read firmware version response
+ 1	0x05	Length
+ 2	Error Code	
+ 3	Version	Firmware version
+ 4	Checksum	

2.3.2.3 CMD_READ_GPIO_7_0

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x02	Read GPIO pin [7..0] command

Bulk In (EP1)

Offset	Name	Description
+ 0	0x03	Response code
+ 1	Length	
+ 2	Error Code	
+ 3	PRT0DR	
+ 4	Checksum	

2.3.2.4 CMD_READ_GPIO_15_8

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x04	Read GPIO pin [15..8] command

Bulk In (EP1)

Offset	Name	Description

+ 0	0x05	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT2DR	
+ 4	Checksum	

2.3.2.5 CMD_READ_GPIO_23_16

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x06	Read GPIO pin [23..16] command

Bulk In (EP1)

Offset	Name	Description
+ 0	0x07	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT3DR	
+ 4	Checksum	

2.3.2.6 CMD_READ_GPIO_31_24

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x08	Read GPIO pin [31..24] command

Bulk In (EP1)

Offset	Name	Description
+ 0	0x09	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT4DR	
+ 4	Checksum	

2.3.2.7 CMD_READ_GPIO_31_0

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x0A	Read GPIO pin [31..0] command

Bulk In (EP1)

Offset	Name	Description
+ 0	0x0B	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT4DR	
+ 4	PRT3DR	
+ 5	PRT2DR	
+ 6	PRT0DR	
+ 7	Checksum	

2.3.2.8 CMD_WRITE_GPIO_7_0

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x10	Write GPIO pin [7..0] command
+ 1	PRT0DR	Value to write to

Bulk In (EP1)

Offset	Name	Description
+ 0	0x11	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT0DR*	Immediately read back after write
+ 4	Checksum	

2.3.2.9 CMD_WRITE_GPIO_15_8

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x12	Write GPIO pin [15..8] command
+ 1	PRT2DR	Value to write to

Bulk In (EP1)

Offset	Name	Description
+ 0	0x13	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT2DR*	Immediately read back after write
+ 4	Checksum	

2.3.2.10 CMD_WRITE_GPIO_23_16

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x14	Write GPIO pin [23..16] command
+ 1	PRT3DR	Value to write to

Bulk In (EP1)

Offset	Name	Description
+ 0	0x15	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT3DR*	Immediately read back after write
+ 4	Checksum	

2.3.2.11 CMD_WRITE_GPIO_31_24

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x16	Write GPIO pin [31..24] command
+ 1	PRT4DR	Value to write to

Bulk In (EP1)

Offset	Name	Description
+ 0	0x17	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT4DR*	Immediately read back after write
+ 4	Checksum	

2.3.2.12 CMD_WRITE_GPIO_31_0

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x18	Write GPIO pin [31..0] command
+ 1	PRT4DR	Value to GPIO pin [31..24]
+ 2	PRT3DR	Value to GPIO pin [23..16]
+ 3	PRT2DR	Value to GPIO pin [15..8]

+4	PRT0DR	Value to GPIO pin [7..0]

Bulk In (EP1)

Offset	Name	Description
+ 0	0x19	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRT4DR*	Immediately read back after write
+ 4	PRT3DR*	Immediately read back after write
+ 5	PRT2DR*	Immediately read back after write
+ 6	PRT0DR*	Immediately read back after write
+ 7	Checksum	

2.3.2.13 CMD_READ_PRTxDMx

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x20	Read PRTxDMx command
+ 1	PRTx	Port number (x = 0, 2, 3, 4)

Bulk In (EP1)

Offset	Name	Description
+ 0	0x21	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRTxDM2	x = 0, 2, 3, 4
+ 4	PRTxDM1	
+ 5	PRTxDM0	
+ 6	Checksum	

2.3.2.14 CMD_WRITE_PRTxDMx

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x22	Write PRTxDMx command
+ 1	PRTx	x = 0, 2, 3, 4
+ 2	PRTxDM2	Value to write to
+ 3	PRTxDM1	Value to write to
+ 4	PRTxDM0	Value to write to

Bulk In (EP1)

Offset	Name	Description
+ 0	0x23	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRTxDM2*	Immediately read back after write
+ 4	PRTxDM1*	Immediately read back after write
+ 5	PRTxDM0*	Immediately read back after write
+ 6	Checksum	

2.3.2.15 CMD_READ_PRTxIE

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x30	Read PRTxIE command
+ 1	PRTx	Port number (x = 0, 2, 3, 4)

Bulk In (EP1)

Offset	Name	Description
+ 0	0x31	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRTxIE	
+ 4	Checksum	

2.3.2.16 CMD_WRITE_PRTxIE

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x32	Write PRTxIE command
+ 1	PRTx	Port number (x = 0, 2, 3, 4)
+ 2	PRTxIE	Value to write to

Bulk In (EP1)

Offset	Name	Description
+ 0	0x33	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRTxIE*	Immediately read back after write
+ 4	Checksum	

2.3.2.17 CMD_READ_PRTxGS

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x34	Read PRTxGS command
+ 1	PRTx	Port number (x = 0, 2, 3, 4)

Bulk In (EP1)

Offset	Name	Description
+ 0	0x35	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRTxGS	
+ 4	Checksum	

2.3.2.18 CMD_WRITE_PRTxGS

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x36	Write PRTxGS command
+ 1	PRTx	Port number (x = 0, 2, 3, 4)

Bulk In (EP1)

Offset	Name	Description
+ 0	0x37	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRTxGS*	Immediately read back after write
+ 4	Checksum	

2.3.2.19 CMD_READ_PRTxICx

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x38	Read PRTxICx command
+ 1	PRTx	Port number (x = 0, 2, 3, 4)

Bulk In (EP1)

Offset	Name	Description
+ 0	0x39	Response
+ 1	Length	

+ 2	Error Code	
+ 3	PRTxIC1	
+ 4	PRTxIC0	
+ 5	Checksum	

2.3.2.20 CMD_WRITE_PRTxICx

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x3A	Write PRTxICx command
+ 1	PRTx	Port number (x = 0, 2, 3, 4)
+ 2	PRTxIC1	Value to write to
+ 3	PRTxIC0	Value to write to

Bulk In (EP1)

Offset	Name	Description
+ 0	0x3B	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRTxIC1	Immediately read back after write
+ 4	PRTxIC0	Immediately read back after write
+ 5	Checksum	

2.3.2.21 CMD_SET_DRIVE_MODE

Bulk Out (EP2)

Offset	Name	Description
+ 0	0x40	Set port drive mode command
+ 1	PRTx	Port number (x = 0, 2, 3, 4)
+ 2	DriveMode	Drive mode number (0, 1, 2, 3, 4, 5, 6, 7)

Bulk In (EP1)

Offset	Name	Description
+ 0	0x41	Response
+ 1	Length	
+ 2	Error Code	
+ 3	PRTxDM2	Immediately read back after write
+ 4	PRTxDM1	Immediately read back after write
+ 5	PRTxDM0	Immediately read back after write
+ 6	Checksum	

Notes:

- Please refer to the CY8C24x94 PSoC TRM for detail information regarding to the drive mode number.
- This command is provided for programming convenience. You can also achieve the same purpose via CMD_WRITE_PRTxDMx command.
- This command limits the whole port (eight pins) to be behavior as either input or output and can’t be mixed.
- All four ports (PRT0, PRT2, PRT3, and PRT4) are programming to have drive mode of value 0x07 by default. This means that GPIO[31..0] pins are default to digital output.
- Set to drive mode 0x02 for the port that you want to behavior to as digital input. 5V is considered to be the logical value of 1; others are regarded to be the logical value of 0.

2.3.3 Scenario

The following scenario shows a way to programming the MPX-24794G device module via vendor specific USB bulk in and bulk out commands.

1. Insert MPX-24794G USB device
2. Install the device driver as instructed by the Windows Device Driver Wizard. Please see the manual for details.
3. The host application opens a connecting MPX-24794G device via the API.
4. The application issues the first command/response paired transfer. Issue the “CMD_READ_FIRMWARE” command and check the “RSP_READ_FIRMWARE” response is recommended.
5. The application then issues paired Bulk Out and Bulk In transactions. The Bulk Out transaction sends the command to the MPX-24794G firmware and the following Bulk In transaction brings back its corresponding response (the result of the just requested command) packet.

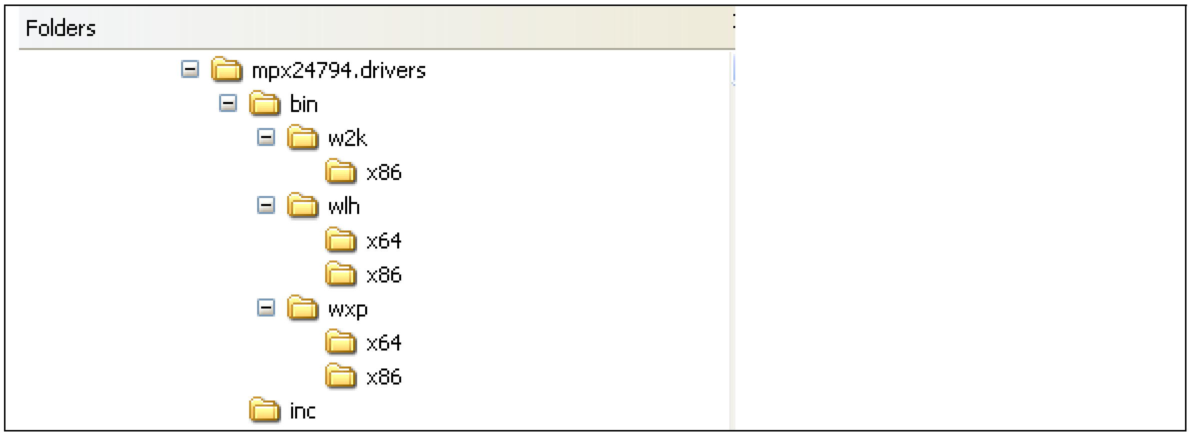
3 Software Guide

This chapter describes how to install the corresponding drivers for your target operating system and how to programming the installed device driver and intermediate layer libraries.

3.1 Device Drivers Guide

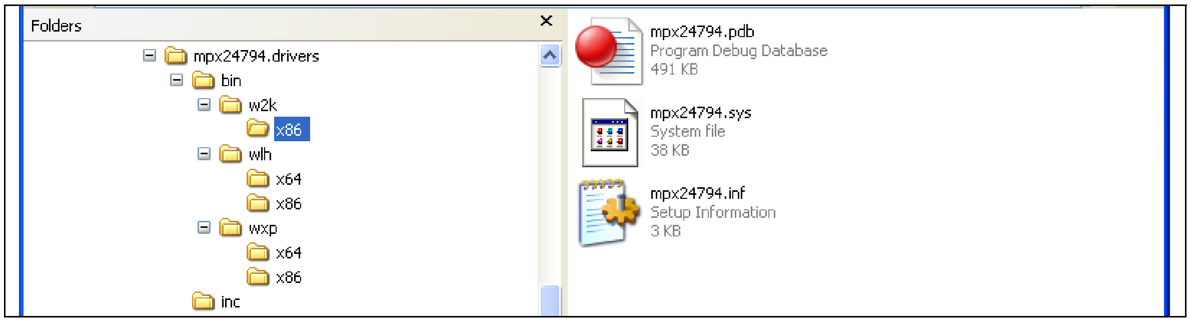
3.1.1 Device Drivers Organization

Figure 9 Device Drivers Organization



3.1.2 Windows 2000

Figure 10 Device Driver for Windows 2000



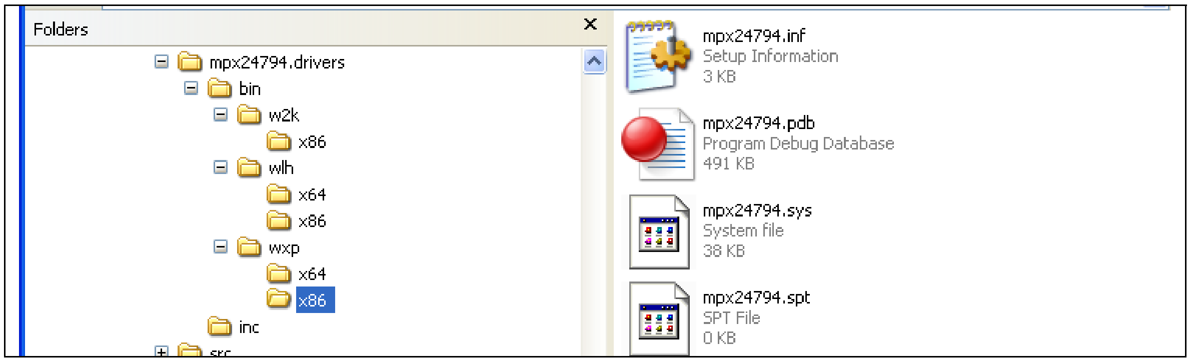
mpx24794.pdb
Program Debug Database
491 KB

mpx24794.sys
System file
38 KB

mpx24794.inf
Setup Information
3 KB

3.1.3 Windows XP (32-bit) Device Driver

Figure 11 Windows XP 32-bit Device Driver



mpx24794.inf
Setup Information
3 KB

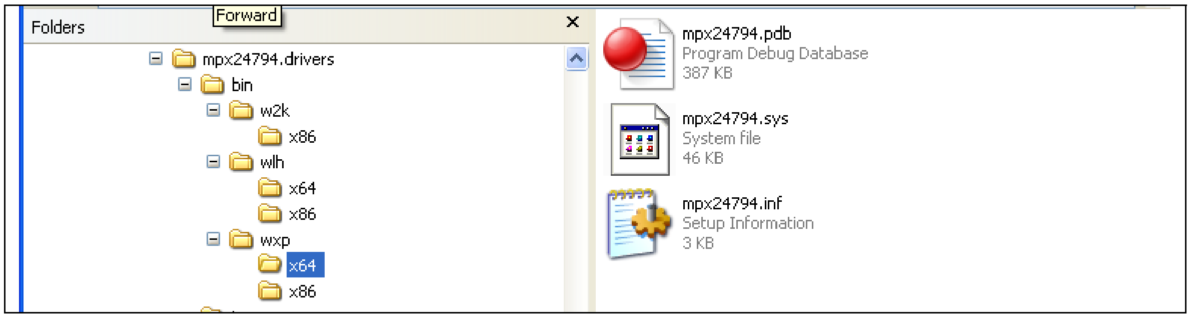
mpx24794.pdb
Program Debug Database
491 KB

mpx24794.sys
System file
38 KB

mpx24794.spt
SPT File
0 KB

3.1.4 Windows XP (64-bit) Device Driver

Figure 12 Windows XP 64-bit Device Driver



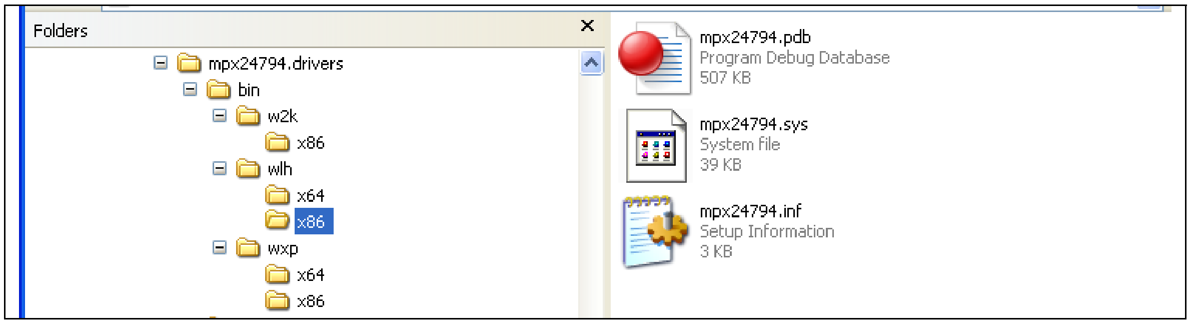
mpx24794.pdb
Program Debug Database
387 KB

mpx24794.sys
System file
46 KB

mpx24794.inf
Setup Information
3 KB

3.1.5 Windows Vista (32-bit) and Windows 7 (32-bit) Device Driver

Figure 13 Windows Vista and Windows 7 Device Driver (32-bit)



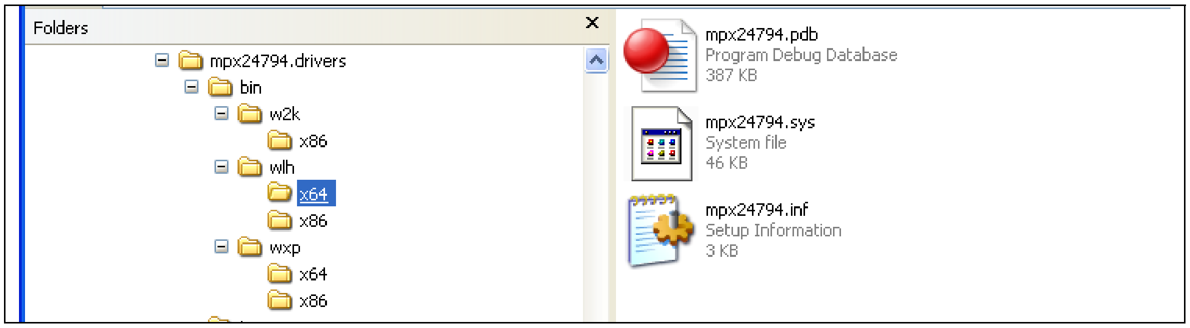
mpx24794.pdb
Program Debug Database
507 KB

mpx24794.sys
System file
39 KB

mpx24794.inf
Setup Information
3 KB

3.1.6 Windows Vista (64-bit) and Windows 7 (64-bit) Device Drivr

Figure 14 Windows Vista and Windows 7 Device Driver (64-bit)



3.2 How to Install Device Driver

Please refer to the "MPX-24794G User's Manual".

3.3 Cypress cyUSB

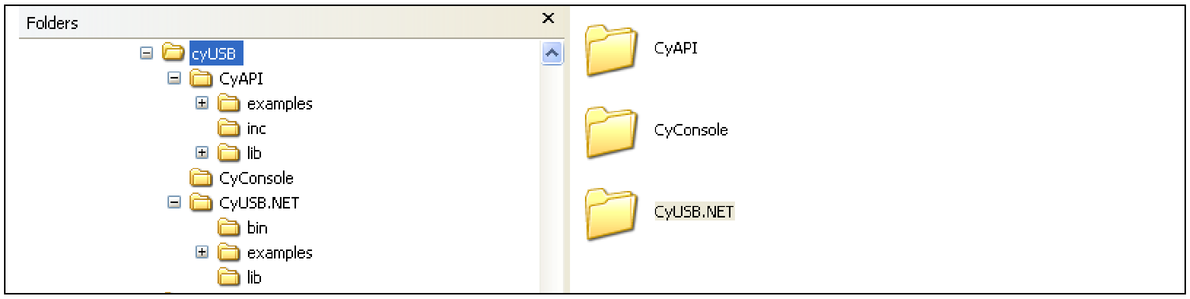
Users are encouraged to use Cypress cyUSB tools to test and program MPX-24794G USB device.

Please refer to CyConsole for testing the MPX-24794G USB device.

Please refer to CyAPI information for C/C++ programming.

Please refer to CyUSB.NET information for .NET programming.

Figure 15 Cypress cyUSB Tools



3.3.1 CyConsole

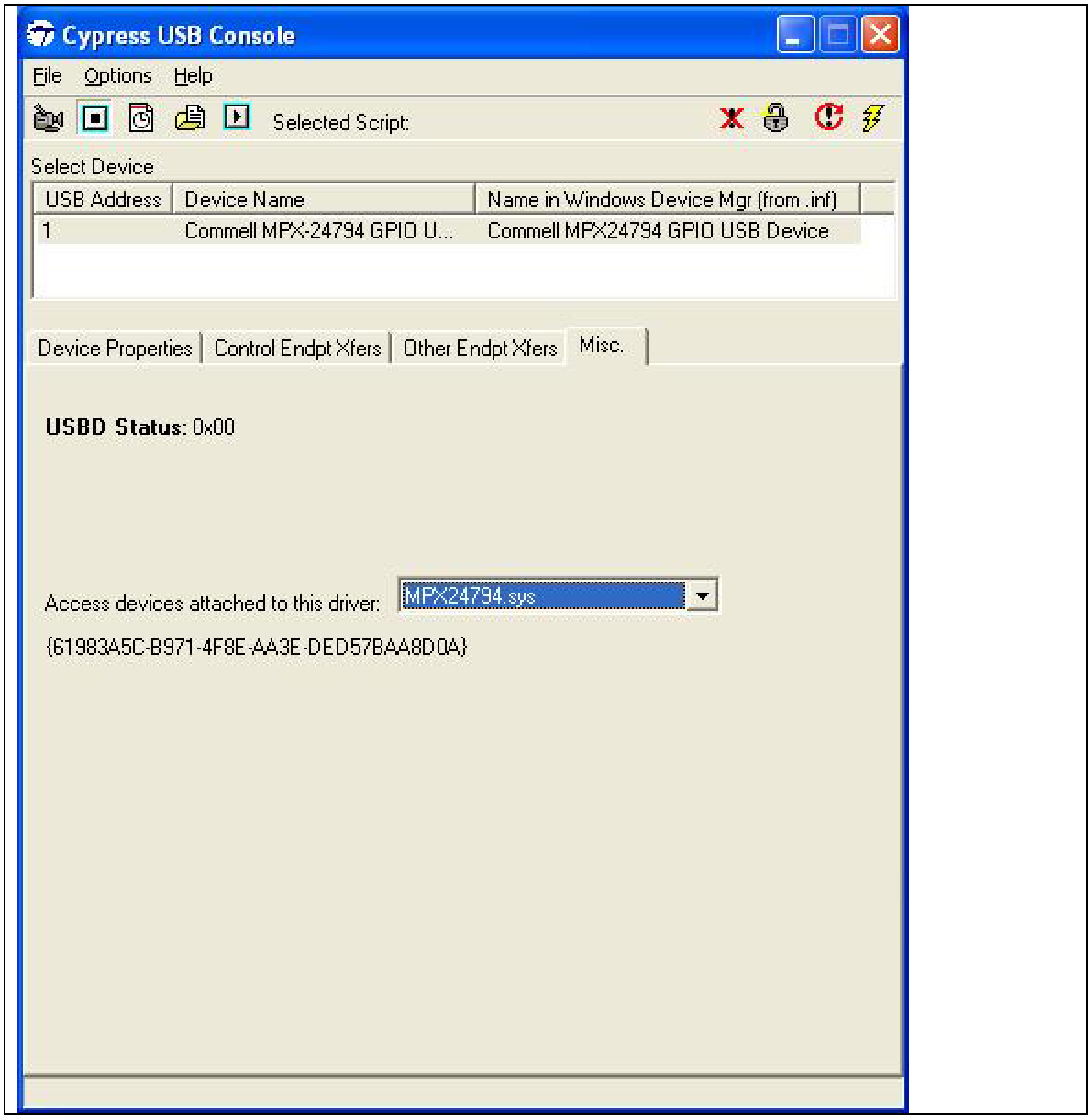
CyConsole can use used to test the Commell IO commands defined for MPX-24794G device.

Figure 12 shows how to select the installed mpx24794.sys device driver when you launch the CyConsole application.

Figure 13 shows the result of issue the first Bulk In transaction. The first Bulk In transaction always reads in the MPX-24794G firmware ready and version information. Application is encouraged to issue a Bulk In transaction prior to any Commell vendor specific IO commands. Please refer to the "Initial Bulk In Content" section for the description of the return package.

Figure 14 shows an example of (command, response) pair scenario. The application first issues a command via Bulk Out transaction according to the command listing. The application then issues a Bulk In transaction in order to retrieve back the result of the executed command. Please refer to the user's guide for detail package descriptions.

Figure 16 Select mpx24794.sys



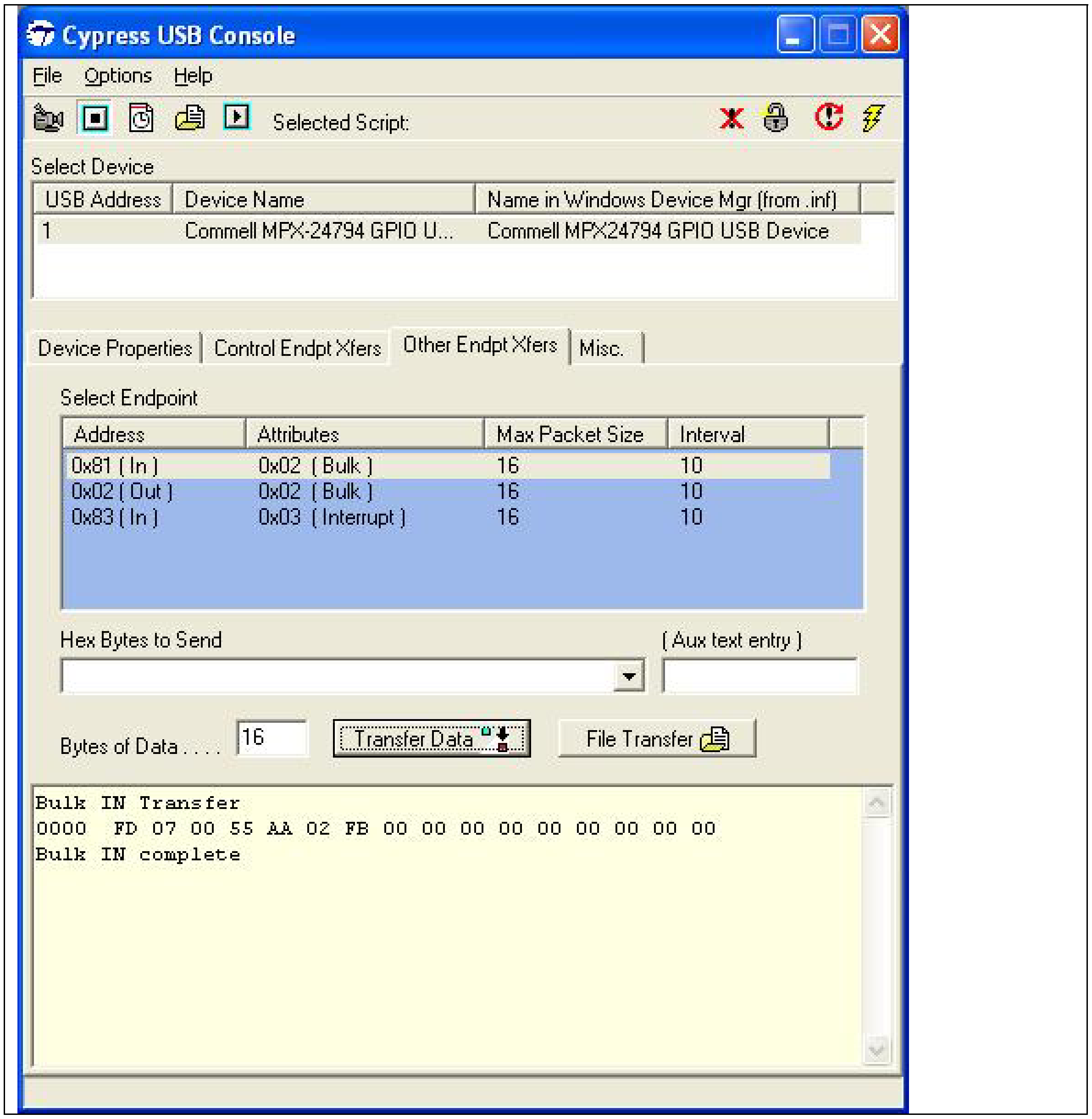
USB Address	Device Name	Name in Windows Device Mgr (from .inf)
1	Commell MPX-24794 GPIO U...	Commell MPX24794 GPIO USB Device

USBD Status: 0x00

Access devices attached to this driver: MPX24794.sys

{61983A5C-B971-4F8E-AA3E-DED57BAA8D0A}

Figure 17 First Bulk In Transaction



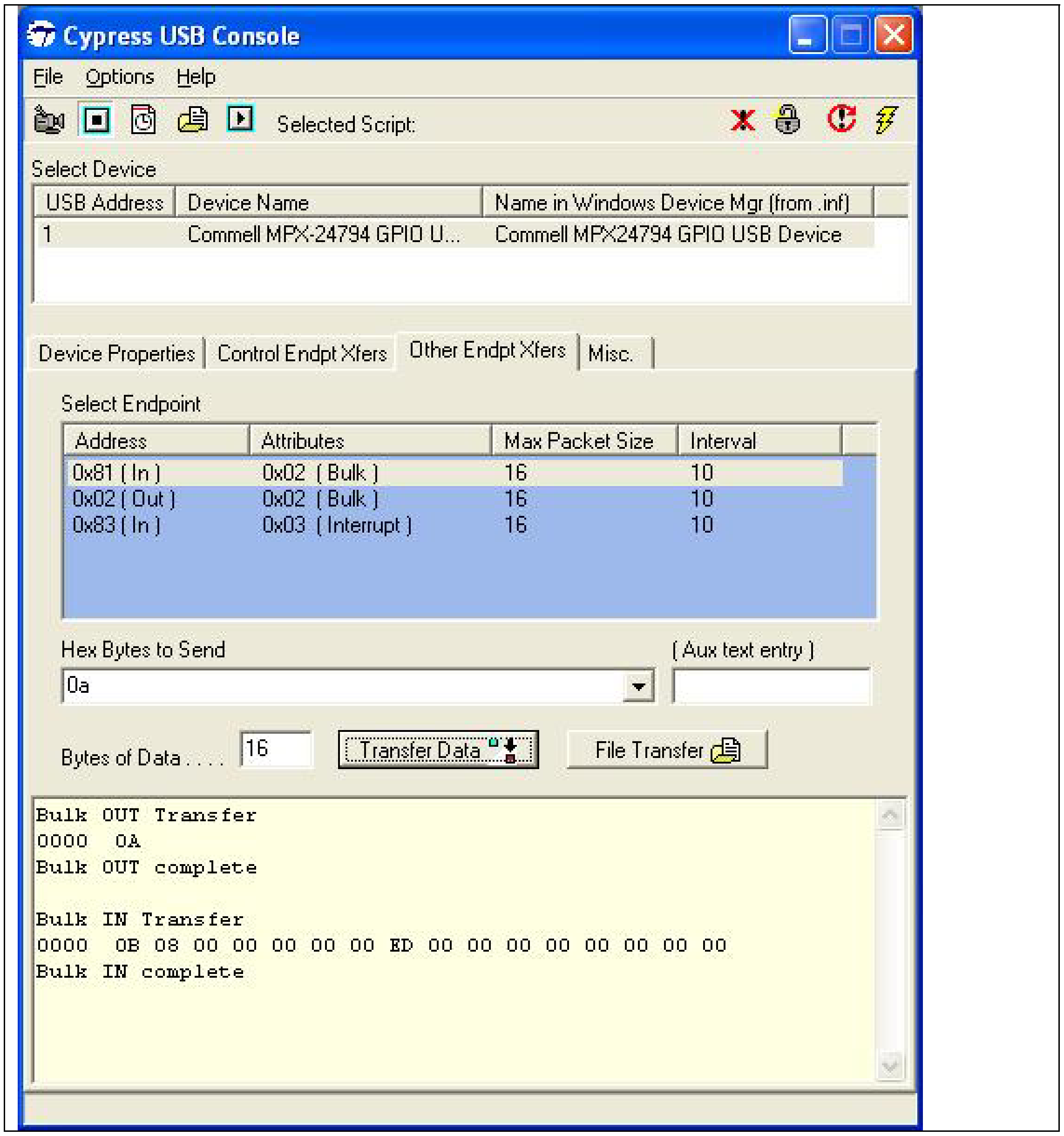
USB Address	Device Name	Name in Windows Device Mgr (from .inf)
1	Commell MPX-24794 GPIO U...	Commell MPX24794 GPIO USB Device

Address	Attributes	Max Packet Size	Interval
0x81 (In)	0x02 (Bulk)	16	10
0x02 (Out)	0x02 (Bulk)	16	10
0x83 (In)	0x03 (Interrupt)	16	10

Bytes of Data 16

```
Bulk IN Transfer
0000  FD 07 00 55 AA 02 FB 00 00 00 00 00 00 00 00 00
Bulk IN complete
```

Figure 18 CMD_READ_GPIO_31_0



Select Device

USB Address	Device Name	Name in Windows Device Mgr (from .inf)
1	Commell MPX-24794 GPIO U...	Commell MPX24794 GPIO USB Device

Select Endpoint

Address	Attributes	Max Packet Size	Interval
0x81 (In)	0x02 (Bulk)	16	10
0x02 (Out)	0x02 (Bulk)	16	10
0x83 (In)	0x03 (Interrupt)	16	10

Hex Bytes to Send: 0a

Bytes of Data 16

```
Bulk OUT Transfer
0000  0A
Bulk OUT complete

Bulk IN Transfer
0000  0B 08 00 00 00 00 00 ED 00 00 00 00 00 00 00 00
Bulk IN complete
```

3.3.2 CyAPI

Please refer to the documentations in the CyAPI folder.

The mpx24794.h file defines the Commell GUID of the Commell MPX-24794G GPIO USB card. Includes this header file into your program file.

3.3.3 CyUSB.NET

Please refer to the documentations in the CyUSB.NET folder.

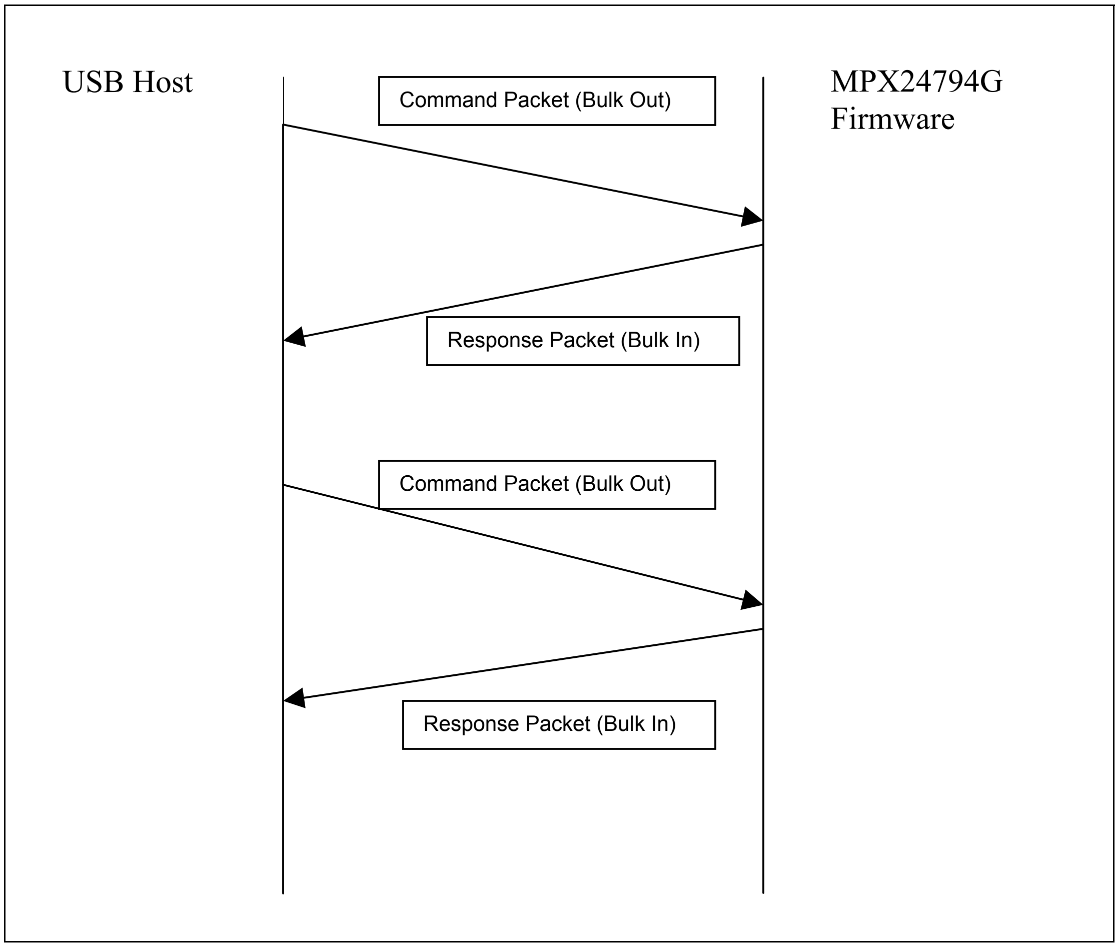
3.4 Programming Guide

This section describes how your Windows programs interact with the MPX-24794G firmware via USB transfers. MPX-24794G implements Command / Response Protocol for communication between USB host and MPX-24794G firmware. Command packets are sent to the MPX-24794G firmware via USB Bulk Out transfers, while Response packets are received immediately via USB Bulk In transfers.

3.4.1 MPX-24794G Command & Response Protocol

Applications communicate with MPX-24794G firmware via Command and Response protocol. Command packet is sent via Bulk Out transfers and Response packet is received via Bulk In transfers. The following diagram shows how the command and response protocol is operating.

Figure 19 Command and Response Protocol


USB Host
MPX24794G Firmware
Command Packet (Bulk Out)
Response Packet (Bulk In)
Command Packet (Bulk Out)
Response Packet (Bulk In)

Notes:

- Bulk out transfer is used to send command packet out from the host side to the MPX24794G firmware side.
- Bulk in transfer is used to receive response packet from the MPX24794G firmware side.
- A command packet is 16 bytes.
- A response packet is 16 bytes.
- The Command Response Protocol is paired protocol. That is a command packet should immediately come with a response packet.

3.4.2 Shadow Register Programming

You programs need to implement shadow register solution if your application meets the criteria of the read/modify/write effects.

Here is an example of how you implement shadow register solution in your C/C++ programs.

```
/* Declare each shadow registers as global variables */
extern unsigned char Port_0_Data_SHADE;
extern unsigned char Port_2_Data_SHADE;
extern unsigned char Port_3_Data_SHADE;
extern unsigned char Port_4_Data_SHADE;
```

Keep these global variables for the output register values for the Port 0, Port 2, Port 3, and Port 4. Initialize these variables as your program wants to initialize PRT0DR, PRT2DR, PRT3DR, and PRT4DR registers. Keep these variables up to synchronous while your program is using the command/response protocol.

3.4.3 C/C++ Programming

The CyAPI.lib needs to be linked with your application. Both the "cyapi.h" file and "mpx24794.h" header files need to be included into your project.

Your new CCyUSBDevice( ) needs to have "MPX24794G_DRV_GUID" constant specified as the second parameter in order to access Commell MPX-24794G GPIO USB card.

Example:

```
#include "cyapi.h"
#include "mpx24794.h"

void main()
{
  CCyUSBDevice *USBDevice;
  USB_DEVICE_DESCRIPTOR descr;

  /* Create an instance of CCyUSBDevice */
  USBDevice = new CCyUSBDevice(NULL, MPX24794G_DRV_GUID, true);
}
```

3.4.4 .NET Class Library

CyUSB.dll is a managed Microsoft .NET class library. It provides a high-level, powerful programming interface to USB devices.

Because CyUSB.dll is a managed .NET library, its classes and methods can be accessed from any of the Microsoft Visual Studio .NET managed languages such as Visual Basic .NET, C#, Visual J#, and managed C++.

To use the library, you need to add a reference to CyUSB.dll to your project's Reference folder. Then any source file that access the CyUSB namespace will need to include a line to include the namespace in the appropriate syntax.

The library employs a model of *DeviceList*, *Devices* and *EndPoints*. An application will normally create an instance of the USBDeviceList class, which represents a list of USB devices. Each of those devices can then be accessed individually.

Please refer to the "Cypress CyUSB .NET DLL Programmer's Reference" of the Cypress USB Suite v3.4.2 for detail information.

4 Sample Code

This sample code is implemented by using Microsoft Visual Studio 2010. You can also download this sample code from our Web site.

```
/**
  mpx24794g-demo.cpp
      A demo program shows how to communicate with MPX24794G card.
      06/21/2012
 */


#include "stdafx.h"
#include <Windows.h>
#include <stdio.h>
#include <conio.h>
#include "CyAPI.h"
#include "mpx24794guids.h"
#include "mpx24794cmds.h"

#define SIZE_BULK_PACKET          16

#define OFFSET_RESPONSE_CODE      0
#define OFFSET_LENGTH                                 1
#define OFFSET_ERROR_CODE                2
#define OFFSET_RETURN_0                         3
#define OFFSET_RETURN_1                         (OFFSET_RETURN_0 + 1)
#define OFFSET_RETURN_2                         (OFFSET_RETURN_0 + 2)
#define OFFSET_RETURN_3                         (OFFSET_RETURN_0 + 3)
#define OFFSET_RETURN_4                         (OFFSET_RETURN_0 + 4)
#define OFFSET_RETURN_5                         (OFFSET_RETURN_0 + 5)

#define ERR_ZERO                                                    0
#define ERR_SUCCESS                                          ERR_ZERO
#define ERR_XFER_TIMEOUT                        0X80
#define ERR_WRONG_RESPONSE_CODE   0X81
#define ERR_CHECKSUM_ERROR             0X82
#define ERR_INVALID_PACKET             0X83


int BulkOutAndIn(unsigned char bResponseCode, CCyBulkEndPoint *pBulkOut,
unsigned char *pCommand, CCyBulkEndPoint *pBulkIn, unsigned char *pResponse);
int CheckResponse(unsigned char bResponseCode, unsigned char *pResponse);

void GetVersion(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn);
void CheckPorts(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn);
void CheckPins_7_0(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn);
void CheckPins_15_8(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn);
void CheckPins_23_16(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn);
void CheckPins_31_24(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn);

CCyUSBDevice *pDevice = NULL;
CCyBulkEndPoint *pBulkIn1 = NULL;
CCyBulkEndPoint *pBulkOut2 = NULL;

int _tmain(int argc, _TCHAR* argv[])
{
	int ch;

	pDevice = new CCyUSBDevice(NULL, MPX24794G_DRV_GUID, true);
	if(0 == pDevice)
	{
		printf("No MPX24794G found\n");
		return 0;
	}

	printf("%s\n", pDevice->FriendlyName);
	printf("VID=%X; PID=%X\n", pDevice->VendorID, pDevice->ProductID);

	pBulkOut2 = (CCyBulkEndPoint *)pDevice->EndPoints[2];
	pBulkIn1 = (CCyBulkEndPoint *)pDevice->EndPoints[1];

	if((NULL == pBulkOut2) || (NULL == pBulkIn1))
	{
		printf("Wrong endpoints\n");
		goto done;
	}

	printf("Get Firmware Version:\n");
	GetVersion(pBulkOut2, pBulkIn1);

	printf("\n");
	printf("Check ports\n");
	CheckPorts(pBulkOut2, pBulkIn1);

	CheckPins_7_0(pBulkOut2, pBulkIn1);
	CheckPins_15_8(pBulkOut2, pBulkIn1);
	CheckPins_23_16(pBulkOut2, pBulkIn1);
	CheckPins_31_24(pBulkOut2, pBulkIn1);

done:
	printf("\n\n");
#ifdef _DEBUG
	printf("Press any key to exit...\n");
	ch = _getch();
#endif /* DEBUG */

	if(pDevice)
		pDevice->Close();

	return 0;
} /* tmain */
```

```
void GetVersion(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn)
{
	int err;
	unsigned char *pCommand;
	unsigned char *pResponse;
	pCommand = (unsigned char *) new unsigned char [SIZE_BULK_PACKET];
	pResponse = (unsigned char *) new unsigned char [SIZE_BULK_PACKET];

	if((NULL == pCommand) || (NULL == pResponse))
	{
		printf("Failed to allocate buffer\n");
		goto done;
	}

	*(pCommand + 0) = CMD_READ_VERSION;

	err = BulkOutAndIn(RSP_READ_VERSION, pBulkOut, pCommand, pBulkIn, 
pResponse);
	if(ERR_ZERO == err)
	{
		printf("Firmware version: 0X%2.2X\n", *(pResponse + 
OFFSET_RETURN_0));
		printf("Response Packet: ");
		for(int i = 0; i < SIZE_BULK_PACKET; i++)
		{
			printf("%2.2X ", *(pResponse + i));
		}
		printf("\n");
	}

done:
	if(pCommand)
		delete [] pCommand;

	if(pResponse)
		delete [] pResponse;
} /* GetVersion */
```

```
void CheckPorts(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn)
{
	int err;
	unsigned char *pCommand;
	unsigned char *pResponse;

	pCommand = (unsigned char *) new unsigned char [SIZE_BULK_PACKET];
	pResponse = (unsigned char *) new unsigned char [SIZE_BULK_PACKET];

	if((NULL == pCommand) || (NULL == pResponse))
	{
		printf("Failed to allocate buffer\n");
		goto done;
	}

	/* Check GPIO_7_0 */
	printf("Check GPIO_7_0\n");
	*(pCommand + 0) = CMD_WRITE_GPIO_7_0;
	*(pCommand + 1) = 0XFF;
	err = BulkOutAndIn(RSP_WRITE_GPIO_7_0, pBulkOut, pCommand, pBulkIn, 
pResponse);
```

```
        printf("0X%X = RSP_WRITE_GPIO_7_0\n", err);

        *(pCommand + 0) = CMD_READ_GPIO_7_0;
        err = BulkOutAndIn(RSP_READ_GPIO_7_0, pBulkOut, pCommand, pBulkIn, 
pResponse);
        if(ERR_ZERO == err)
        {
                printf("GPIO_7_0: 0X%2.2X\n", *(pResponse + OFFSET_RETURN_0));
                printf("Response Packet: ");
                for(int i = 0; i < SIZE_BULK_PACKET; i++)
                {
                        printf("%2.2X ", *(pResponse + i));
                }
                printf("\n");
        }

        /* Check GPIO_15_8*/
        printf("Check GPIO_15_8\n");
        *(pCommand + 0) = CMD_WRITE_GPIO_15_8;
        *(pCommand + 1) = 0XFF;
        err = BulkOutAndIn(RSP_WRITE_GPIO_15_8, pBulkOut, pCommand, pBulkIn, 
pResponse);
        printf("0X%X = RSP_WRITE_GPIO_15_8\n", err);

        *(pCommand + 0) = CMD_READ_GPIO_15_8;
        err = BulkOutAndIn(RSP_READ_GPIO_15_8, pBulkOut, pCommand, pBulkIn, 
pResponse);
        if(ERR_ZERO == err)
        {
                printf("GPIO_7_0: 0X%2.2X\n", *(pResponse + OFFSET_RETURN_0));
                printf("Response Packet: ");
                for(int i = 0; i < SIZE_BULK_PACKET; i++)
                {
                        printf("%2.2X ", *(pResponse + i));
                }
                printf("\n");
        }

        /* Check GPIO_23_16*/
        printf("Check GPIO_23_16\n");
        *(pCommand + 0) = CMD_WRITE_GPIO_23_16;
        *(pCommand + 1) = 0XFF;
        err = BulkOutAndIn(RSP_WRITE_GPIO_23_16, pBulkOut, pCommand, pBulkIn, 
pResponse);
        printf("0X%X = RSP_WRITE_GPIO_23_16\n", err);

        *(pCommand + 0) = CMD_READ_GPIO_23_16;
        err = BulkOutAndIn(RSP_READ_GPIO_23_16, pBulkOut, pCommand, pBulkIn, 
pResponse);
        if(ERR_ZERO == err)
        {
                printf("GPIO_7_0: 0X%2.2X\n", *(pResponse + OFFSET_RETURN_0));
                printf("Response Packet: ");
                for(int i = 0; i < SIZE_BULK_PACKET; i++)
                {
                        printf("%2.2X ", *(pResponse + i));
                }
                printf("\n");
        }

        /* Check GPIO_31_24 */
        printf("Check GPIO_31_24\n");
        *(pCommand + 0) = CMD_WRITE_GPIO_31_24;
        *(pCommand + 1) = 0XFF;
        err = BulkOutAndIn(RSP_WRITE_GPIO_31_24, pBulkOut, pCommand, pBulkIn,
pResponse);
        printf("0X%X = RSP_WRITE_GPIO_31_24\n", err);

        *(pCommand + 0) = CMD_READ_GPIO_31_24;
        err = BulkOutAndIn(RSP_READ_GPIO_31_24, pBulkOut, pCommand, pBulkIn,
pResponse);
        if(ERR_ZERO == err)
        {
                printf("GPIO_31_24: 0X%2.2X\n", *(pResponse + OFFSET_RETURN_0));
                printf("Response Packet: ");
                for(int i = 0; i < SIZE_BULK_PACKET; i++)
                {
                        printf("%2.2X ", *(pResponse + i));
                }
                printf("\n");
        }

done:
        if(pCommand)
                delete [] pCommand;

        if(pResponse)
                delete [] pResponse;
} /* CheckPorts */
```

```
void CheckPins_7_0(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn)
{
        int err;
        unsigned char pin;
        unsigned char *pCommand;
        unsigned char *pResponse;

        pCommand = (unsigned char *) new unsigned char [SIZE_BULK_PACKET];
        pResponse = (unsigned char *) new unsigned char [SIZE_BULK_PACKET];

        if((NULL == pCommand) || (NULL == pResponse))
        {
                printf("Failed to allocate buffer\n");
                goto done;
        }

        /* Check pins of GPIO_7_0 */
        printf("Check pins of GPIO_7_0\n");
        pin = 0x01;
        for(int j = 0; j < 9; j++)
        {
                *(pCommand + 0) = CMD_WRITE_GPIO_7_0;
                *(pCommand + 1) = ~pin;
```

```
              err = BulkOutAndIn(RSP_WRITE_GPIO_7_0, pBulkOut, pCommand,
pBulkIn, pResponse);
              printf("0X%X = RSP_WRITE_GPIO_7_0\n", err);

              *(pCommand + 0) = CMD_READ_GPIO_7_0;
              err = BulkOutAndIn(RSP_READ_GPIO_7_0, pBulkOut, pCommand,
pBulkIn, pResponse);
              if(ERR_ZERO == err)
              {
                     printf("GPIO_7_0: 0X%2.2X\n", *(pResponse +
OFFSET_RETURN_0));
                     printf("Response Packet: ");
                     for(int i = 0; i < SIZE_BULK_PACKET; i++)
                     {
                            printf("%2.2X ", *(pResponse + i));
                     }
                     printf("\n");
              }
              pin <<= 1;
              Sleep(1000);
       }

done:
       if(pCommand)
              delete [] pCommand;

       if(pResponse)
              delete [] pResponse;
} /* CheckPins_7_0 */
```

```
void CheckPins_15_8(CCyBulkEndPoint *pBulkOut, CCyBulkEndPoint *pBulkIn)
{
       int err;
       unsigned char pin;
       unsigned char *pCommand;
       unsigned char *pResponse;

       pCommand = (unsigned char *) new unsigned char [SIZE_BULK_PACKET];
       pResponse = (unsigned char *) new unsigned char [SIZE_BULK_PACKET];

       if((NULL == pCommand) || (NULL == pResponse))
       {
              printf("Failed to allocate buffer\n");
              goto done;
       }

       /* Check pins of GPIO_7_0 */
       printf("Check pins of GPIO_15_8\n");
       pin = 0x01;
       for(int j = 0; j < 9; j++)
       {
              *(pCommand + 0) = CMD_WRITE_GPIO_15_8;
              *(pCommand + 1) = ~pin;
              err = BulkOutAndIn(RSP_WRITE_GPIO_15_8, pBulkOut, pCommand,
pBulkIn, pResponse);
```

```
		printf("0X%X = RSP_WRITE_GPIO_15_8\n", err);

		*(pCommand + 0) = CMD_READ_GPIO_15_8;
		err = BulkOutAndIn(RSP_READ_GPIO_15_8, pBulkOut, pCommand,
pBulkIn, pResponse);
		if(ERR_ZERO == err)
		{
			printf("GPIO_15_8: 0X%2.2X\n", *(pResponse +
OFFSET_RETURN_0));
			printf("Response Packet: ");
			for(int i = 0; i < SIZE_BULK_PACKET; i++)
			{
				printf("%2.2X ", *(pResponse + i));
			}
			printf("\n");
		}
		pin <<= 1;
		Sleep(1000);
	}

done:
	if(pCommand)
		delete [] pCommand;

	if(pResponse)
		delete [] pResponse;
} /* CheckPins_15_8 */
```

```
int BulkOutAndIn(unsigned char bResponseCode, CCyBulkEndPoint *pBulkOut,
unsigned char *pCommand, CCyBulkEndPoint *pBulkIn, unsigned char *pResponse)
{
	int err;
	LONG length;

	length = 16;
	err = pBulkOut->XferData(pCommand, length);
	if(!err)
		return ERR_XFER_TIMEOUT;

	err = pBulkIn->XferData(pResponse, length);
	if(!err)
		return ERR_XFER_TIMEOUT;

	err = CheckResponse(bResponseCode, pResponse);

	return err;
} /* BulkOutAndIn */
```

```
int CheckResponse(unsigned char bResponseCode, unsigned char *pResponse)
{
	int length;
	unsigned char checksum;
	if(bResponseCode != *(pResponse + OFFSET_RESPONSE_CODE))
		return ERR_WRONG_RESPONSE_CODE;

	checksum = 0;
	length = *(pResponse + OFFSET_LENGTH);
	if(length > SIZE_BULK_PACKET)
		return ERR_INVALID_PACKET;

	for(int i = 0; i < length; i++)
	{
		checksum += *(pResponse + i);
	}

	return((checksum)? ERR_CHECKSUM_ERROR : ERR_SUCCESS);
} /* CheckResponse */
```

5 References

[1] CY8C24x94 PSoC Programmable System-on-Chip Technical Reference Manual (TRM). Document No. 001-14463 Rev. *F.

[2] Cypress CyAPI Programmer's Reference, Cypress Suite USB 3.4.2.

[3] Cypress CyUSB .NET DLL Programmer's Reference, Cypress Suite USB 3.4.2.