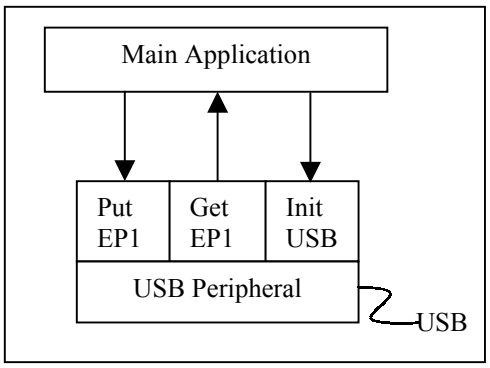
USB Firmware Users Guide – Assembly version

V1.21 5 August 2001

1. Introducing the USB software interface

Microchip provides a layer of software for the PIC16C745/65 that handles the lowest level interface so your application won't have to. This provides a simple Put/Get interface for communication. Most of the USB processing takes place in the background through the Interrupt Service Routine. From the application viewpoint, the enumeration process and data communication takes place without further interaction. However substantial setup is required in the form of generating appropriate descriptors.



2. Integrating USB into your application

The latest version of the USB interface software is available on Microchip's website. See http://www.microchip.com/

The interface to the application is packaged in 5 functions: **InitUSB, PutEP1, PutEP2, GetEP1** and **GetEP2**. **InitUSB** initializes the USB peripheral allowing the host to enumerate the device. Then for normal data communications, the **PutEPn** functions send data to the host and the **GetEPn** functions receive data from the host.

There's also a fair amount of setup work that must be completed. USB depends heavily on the descriptors. These are the software parameters that are communicated to the host to let it know what the device is, and how to communicate with it. See USB V1.1 spec section 9.5 for more details.

InitUSB enables the USB interrupt so enumeration can begin. The actual enumeration process occurs in the background, driven by the host and interrupt service routine. Macro **ConfiguredUSB** waits until the device is in the CONFIGURED state. The time required to enumerate is completely dependent on the host and bus loading.

3. Interrupt structure concerns

Processor Resources

Most of the USB processing occurs via the interrupt and thus is invisible to the application. However it still consumes processor resources. These include ROM, RAM, Common RAM, Stack Levels and processor cycles. This section attempts to quantify the impact on each of these resources, and shows ways to avoid conflicts.

These considerations should be taken into account if you write your own Interrupt Service Routine: Save W, STATUS, FSR and PCLATH which are the file registers that may be corrupted by servicing the USB interrupt.

The file *usb_main.asm* provides a skeleton ISR which does this for you, and includes tests for each of the possible ISR bits. This provides a good starting point if you haven't already written your own.

Stack Levels

The hardware stack on the PIC is only 8 levels deep. So the worst case call between the application and ISR can only be 8 levels. The enumeration process requires 4 levels, so it's best if the main application holds off on any processing until enumeration is complete. **ConfiguredUSB** is a macro that waits until the enumeration process is complete for exactly this purpose. This macro does this by testing the lower two bits of USWSTAT (0x197).

ROM

The code required to support the USB interrupt, including the chapter 9 interface calls, but not including the HID interface calls or descriptor tables is about 1kW. The HID specific functions and

descriptor tables take up additional memory. The location of these parts is not restricted, and the linker script may be edited to control the placement of each part. See the Strings and Descriptors sections in the linker script.

RAM
With the exception of Common RAM discussed below, servicing the USB interrupt requires approximately 40 bytes of RAM in Bank 2. That leaves all the General Purpose RAM in banks zero and one, plus half of Bank 2 available for your application to use.

Common RAM usage
The PIC16C745/765 has 16 bytes of common RAM. These are the last 16 addresses in each bank and all refer to the same 16 bytes of memory without regard to which register bank is currently addressed by the RP0, RP1 and IRP bits.

These are particularly useful when responding to interrupts. When an interrupt occurs, the ISR doesn't immediately know which bank is addressed. With devices that don't support common RAM, the W register must be provided for in each bank. The PIC16C745/765 can save the appropriate registers in Common RAM and not have to waste a byte in each bank for the W register.

Buffer allocation
The PIC16C745/765 has 64 bytes of Dual Port RAM. 24 are used for the Buffer Descriptor Table, (BDT) leaving 40 bytes for buffers.

Endpoint 0 (EP0) IN and OUT need dedicated buffers since a setup transaction can never be NAKed. That leaves three buffers for four possible endpoints. But the USB spec requires that low speed devices are only allowed 2 endpoints (USB V1.1 paragraph 5.3.1.2), where an endpoint is a simplex connection that is defined by the combination of endpoint number and direction.

The default configuration allocates individual buffers to EP0 OUT, EP0 IN, EP1 OUT, and EP1 IN. The last buffer is shared between EP2 IN and EP2 OUT. Again, the spec says low speed devices can only use 2 endpoints beyond EP0. This configuration supports most of the possible combinations of endpoints (EP1 OUT and EP1 IN, EP1 OUT and EP2 IN, EP1 OUT and EP2 OUT, EP1 IN and EP2 OUT, EP1 IN and EP2 IN). The only combination that is not supported by this configuration is EP2 IN and EP2 OUT. If your application needs both EP2 IN and EP2 OUT, the function **USBReset** will need to be edited to give each of these endpoints dedicated buffers at the expense of EP1.

4. File packaging

The software interface is packaged into five files, designed to simplify the integration with your application. These are:

1. usb_main.asm
2. usb_ch9.asm
3. hidclass.asm
4. descript.asm
5. usb_defs.inc

File *usb_main.asm* is useful as a starting point on a new application and as an example of how an existing application needs to service the USB interrupt and communicate with the core functions.

File *usb_ch9.asm* contains the interface and core functions needed to enumerate the bus. It also contains the functions that service the USB, send data to the host, and receive data from the host.

File *hidclass.asm* provides some HID Class specific functions. Currently, only **Get_Report_Descriptor** and **Set_Report** are supported. Other class specific functions can be implemented in a similar fashion. When a get token interrupt determines that it is a class specific command on the basis that ReportType bit 6 is set, control is passed to function **ClassSpecific**. If

you're working with a different class, this is your interface between the core functions and the class specific functions.

File *descript.asm* contains the device, config, interface, endpoint and string descriptors. It also includes the HID specific descriptors -- HID and Report.

File *usb_defs.inc* contains several macros including **ConfigUSB**, **PutEP1**, **PutEP2**, **GetEP1**, and **GetEP2**. As mentioned before, instances of the put and get macros have been created in *usb_ch9.asm* already.

5. Function Call reference

Interface between the Application and Protocol layer takes place in five main functions: **InitUSB**, **PutEP1, PutEP2, GetEP1** and **GetEP2**.

InitUSB should be called by the main program after the device is powered up. Be sure to precede the call with a 16 µs delay to give the SIE a chance to reset before beginning enumeration. **InitUSB** enables the USB peripheral and the USB reset interrupt. It also transitions the part to the powered state in order to prepare the device for enumeration. Before enumeration is initiated, the USB Reset is the only USB interrupt allowed, preventing the part from responding to anything on the bus until it's been reset. The USB Reset interrupt initializes the Buffer Descriptor Table (BDT) and transitions the part to the default state where it responds to commands on address zero. When it receives a SET ADDRESS command, the device transitions to the addressed state and now responds to commands on the new address.

PutEPn sends data to the host. Call this function with the following specified:

- The beginning address of the block of data to transmit in FSR/IRP.
- The number of bytes in the block of data in W.

If the IN buffer is available for that endpoint, the block of data is copied to the buffer, then the Data 0/1 bit is flipped and the owns bit is set. Should the buffer not be available, a failure code will be returned so the application can try again later (the Carry Flag will be cleared.) If the function is successful it will set the Carry Flag.

GetEPn returns data sent from the host. Call this function with the beginning address of the desired location data will be placed in FSR/IRP. If there is a buffer ready, meaning data has been received from the host, it is copied to the destination pointed to by FSR/IRP. The function will return the number of bytes copied in the W register. If no data is available, it returns a failure code (the Carry Flag is cleared.) If the function is successful it will set the Carry Flag.

ServiceUSBInt handles all interrupts generated by the USB peripheral. First it copies the active buffer to common RAM which provides a quick turn around on the buffer in dual port RAM and also to avoids having to switch banks during processing of the buffer.

StallUSBEP/UnstallUSBEP sets or clears the stall bit in the endpoint control register. The stall bit indicates to the host that user intervention is required and until such intervention is made, further attempts to communicate with the endpoint will not be successful. Once the user intervention has been made, **UnstallUSBEP** will clear the bit allowing communications to take place. These calls are used to signal to the host that user intervention is required. An example of this might be a printer out of paper.

SoftDetachUSB clears the DEV_ATT bit, electrically disconnecting the device from the bus, then reconnecting so it can be re-enumerated by the host. This process takes approximately 50 mS, to ensure that the host has seen the device disconnect and reattach to the bus.

CheckSleep tests the UCTRL.UIDLE bit which if set, indicates that there has been no activity on the bus for 3 mS. If set, the device can be put to sleep, which puts the part into a low power standby mode

until awakened by bus activity. This has to be handled outside the ISR because we need the interrupt to wake the PICmicro from sleep, and also because the application may not be ready to sleep when the interrupt occurs. Instead, the application should periodically call this function in order to poll the bit and find out when the device is in a good place to sleep.

Prior to putting the device to sleep, it enables the activity interrupt so the device will be awakened by the first transition on the bus. The PIC will immediately jump to the ISR which recognizes the activity interrupt and then disables the interrupt and resumes processing with the instruction following the **CheckSleep** call.

ConfiguredUSB (a macro) continuously polls the enumeration status bits and waits until the device has been configured by the host. This should be used after the call to **InitUSB** and prior to the first time your application attempts to communicate on the bus.

SetConfiguration is a callback function that allows your application to associate some meaning to a Set Configuration command from the host. The Ch9 software stores the value in USB_Curr_Config so it can be reported back on a **Get_Configuration** call. This function is also called, passing the new configuration into W. This function is called from within the ISR so it should be kept as short as possible.

6. Behind the scenes:

InitUSB clears the error counters and enables the 3.3 V regulator and the USB reset interrupt. This implements the requirement to prevent the PIC from responding to commands until the device has been reset.

The host sees the device, and resets the device to begin the enumeration process. The reset then initializes the Buffer Descriptor Table (BDT) and EndPoint Control Registers in addition to enabling the remaining USB interrupt sources.

The Interrupt transfers control to the interrupt vector (address 0x0004). Any interrupt service routine must preserve the processor state by saving the FSRs that might change during interrupt processing. We recommend saving W, STATUS, PCLATH and FSR. W can be stored in Common RAM to avoid banking issues. Then it starts polling the Interrupt flags to see what triggered the interrupt. The USB interrupts are serviced by calling **ServiceUSBInt** which further tests the USB interrupt sources to determine how to process the interrupt.

Then the host sends a setup token requesting the device descriptor. The USB Peripheral receives the Setup transaction, places the data portion in the EP0 out buffer, loads the USTAT register to indicate which endpoint received the data and triggers the Token Done (TOK_DNE) interrupt. The Chapter 9 commands then interpret the Setup token and sets up the data to respond to the request in the EP0 IN buffer. It then sets the UOWN bit to tell the SIE there is data available.

Next, the host sends an IN transaction to receive the data from the setup transaction. The SIE sends the data from the EP0 IN buffer and then sets the Token done interrupt to notify us that the data has been sent. If there is additional data, the next buffer full is set up in the EP0 IN buffer.

This token processing sequence holds true for the entire enumeration sequence, which walks through the flow chart starting chapter 9 of the USB spec. The device starts off in the powered state, transitions to RESET via the reset interrupt, transitions to ADDRESSED via the Set Address command, and transitions to CONFIGURED via a Set Configuration command.

The USB peripheral detects several different errors and handles most internally. The USB_ERR interrupt notifies the PIC that an error has occurred. No action is required by the PIC when an error occurs. Instead the errors are simply acknowledged and counted. There is no mechanism to pull the

device off the bus if there are too many errors. If this behavior is desired it must be implemented in the application.

The Activity interrupt is left disabled until the USB peripheral detects no bus activity for 3 mS. Then it suspends the USB peripheral and enables the activity interrupt. The activity interrupt then reactivates the USB peripheral when bus activity resumes so processing may continue.

CheckSleep is a separate call that takes the bus idle one step further and puts the PIC to sleep if the USB peripheral has detected no activity on the bus. This powers down most of the device to minimal current draw. This call should be made at a point in the main loop where all other processing is complete.

7. Examples

This example shows how the USB functions are used. This example first initializes the USB peripheral which allows the host to enumerate the device. The enumeration process occurs in the background, via an Interrupt service routine. This function waits until enumeration is complete, and then polls EP1 OUT to see if there is any data available. When a buffer is available, it is copied to the IN buffer. Presumably your application would do something more interesting with the data than this example.

```
; *****************************************************************
; Demo program that initializes the USB peripheral, allows the Host
;  to Enumerate, then copies buffers from EP1OUT to EP1IN.
; *****************************************************************
Main
   pagesel     InitUSB
   call        InitUSB     ; Set up everything so we can enumerate
   pagesel     Main
   ConfiguredUSB           ; wait here until we have enumerated.

CheckEP1       ; Check Endpoint 1 for an OUT transaction
   bankisel    BUFFER      ; point to lower banks
   pagesel     GetEP1
   movlw       BUFFER
   movwf       FSR         ; point FSR to our buffer
   call        GetEP1      ; If data is ready, it will be copied.
   pagesel     CheckEP1
   btfss       STATUS,C    ; was there any data for us?
   goto        CheckEP1    ; Nope, check again.

PutBuffer
   bankisel    BUFFER      ; point to lower banks
   pagesel     PutEP1
   movlw       BUFFER
   movwf       FSR         ; point FSR to our buffer
   movlw       0x08        ; send 8 bytes to endpoint 1
   call        PutEP1
   pagesel     PutBuffer
   btfss       STATUS,C    ; was it successful?
   goto        PutBuffer   ; No: try again until successful
   pagesel     CheckEP1
   goto        CheckEP1    ; Yes: restart loop

   end
```

8. Multiple Configuration or Report Descriptors

The Ch9 firmware makes allowances for the fact that more than one configuration or report descriptor may be desired. Allowances for multiple interface, HID, and endpoint descriptors were not made because they are not needed. These descriptors are all read in with the configuration descriptor regardless of how many there are.
The host requests the descriptors by specifying the type of descriptor it wants and an index value. If more than one configuration descriptor exists it will request the first one by specifying an index of zero and the second by specifying and index of one. To make this process as easy as possible for developers to deal with, the functions **Config_descr_index** and **Report_descr_index** have been created in *descript.asm*. These functions will need to be modified in your code has more than one configuration or more than one report descriptor. All that is needed is for you to specify the starting label for additional desriptors in the lookup table for these functions.
Note: String descriptors also use an index function, string_index (in descript.asm). This function will need to be modified in the same manner if your code has a number of strings other than six.

9. Optimizing the Firmware

This firmware has been created to provide developers with ready-made USB functions so they don't have to create these functions for themselves. Most developers will not utilize all of the functions in the Ch9 firmware. In order to optimize the program memory, unused functions can be taken out of the firmware. The following guidelines are a good place to start the optimization.

USB status on Port B.
The firmware outputs the status of USB communication on Port B. This feature is intended for use with the PICDEM USB circuit board which drives an LED with each Port B pin. The LEDs indicate the following USB status information: RB0 – powered, RB1 – default, RB2 – addressed, RB3 – configured, RB4 – sleeping, RB5 – EP0 active, RB6 – EP1 active, RB7 – EP2 active. Obviously, these USB status indicators will probably not be used in a finished product by a developer although they are very helpful during development. All code associated with the USB status LEDs can be eliminated from the program memory by ensuring that SHOW_ENUM_STATUS is not defined at the top of *usb_ch9.asm*.

Error counter
Similar to the USB status LEDs, code exists in the firmware that counts various errors for debugging purposes. To eliminate this excess code from the program memory simply make sure that COUNTERRORS is not defined.

GetEP1, GetEP2, PutEP1, and PutEP2
These functions are all macros defined in *usb_defs.inc*. Instances of each of these macros occur in *usb_ch9.asm*. If a developer does not utilize one or more of these functions, space can be saved by removing the instance(s) not needed from *usb_ch9.asm*.

HID class
The HID class is one of several classes suitable for low-speed USB. In addition to these other classes, a vendor-defined class can be specified. Should a developer use a class other than the HID class, any HID class specific code in the firmware would be wasting space. The HID class specific code is found in the file, *hidclass.asm*. In a case where the HID class is not being utilized by a developer, this file and any variables or labels associated with it should be removed from the project.

10. Cursor Demonstration

Microchip provides a working USB demonstration. This demonstration has the effect of moving the cursor in a small circle on the user's screen. The following steps will get the demonstration working with an actual part or the MPLAB ICE:

Getting the USB demonstration to work on a PIC16C745/65

USB Firmware Users Guide – Assembly version

V1.21 5 August 2001

1. Unzip **usbxxxasm.zip** to a project folder.
2. Build the project in MPLAB.
3. Program a PIC16C765 using a PICSTART Plus or a PRO MATE II. Make sure the configuration bits are set as follows:
   - Oscillator: H4
   - Watchdog Timer: Off
   - Power Up Timer: Off
   - Code Protect: Off

   Note: To program a PIC16C745 you will need to use the linker file and include files associated with the PIC16C745. Also, you will need to identify the part in the Development Mode dialog. See the MPLAB IDE User's Guide for details.
4. The figure at the bottom of this section details the circuit in which to implement the PIC. Attach the USB cable to your computer.
5. Provide power to the PIC. If you are running Windows 98(with the USB upgrade), Windows NT, or Windows 2000 then your operating system will detect a new device and install the necessary drivers automatically. After this occurs you should see the cursor rotating in a small circle on your screen. To stop the cursor from rotating, detach the USB cable.

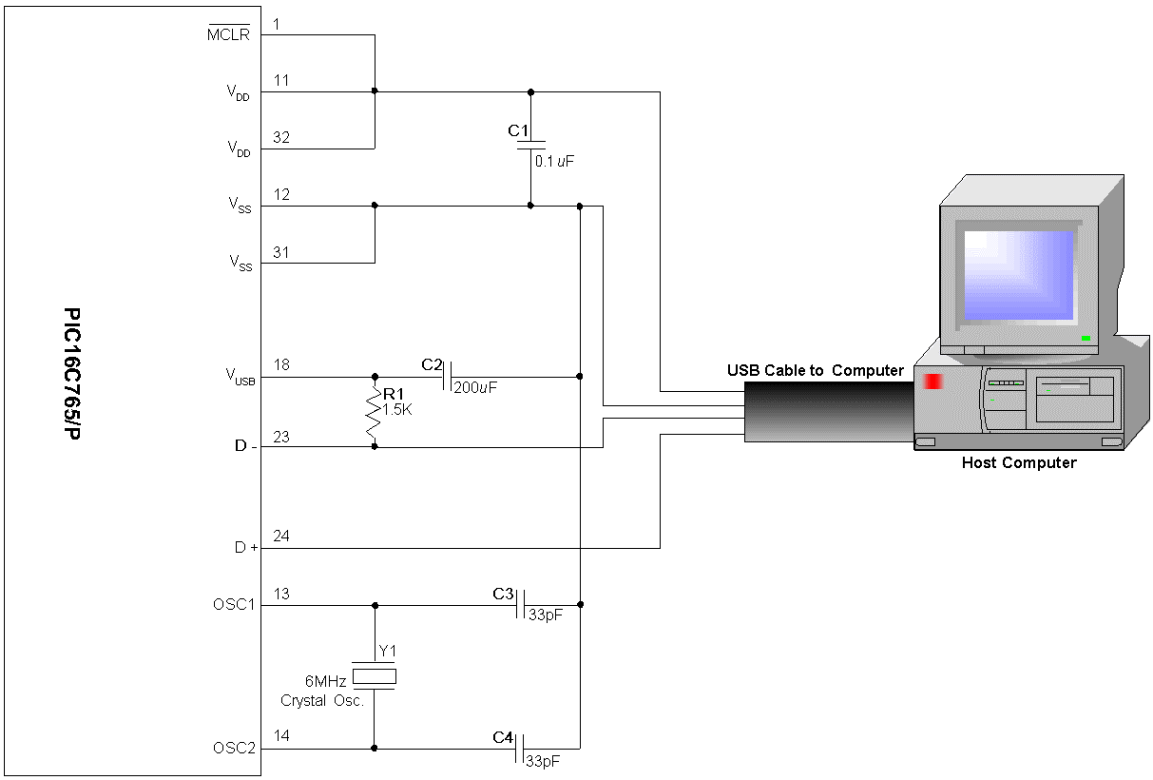
Getting the USB demonstration to work using the MPLAB ICE

1. Unzip **usbxxxasm.zip** to a project folder.
2. Build the project in MPLAB.
3. Make sure the emulator is set up as follows in the Development Mode dialog:
   - Tools: MPLAB-ICE Emulator
   - Clock: Desired Frequency: 24 MHz
   - Configuration: Watch Dog Timer: None
   - Power: Processor Power: From Target Board

   Note: The firmware is set up to run on a PIC15C765 as the default. To emulate a PIC16C745 you will need to use the linker file and include files associated with the PIC16C745. Also you will need to identify the part in the Development Mode dialog.
4. The figure below shows the circuit in which to implement the ICE. Connect the USB cable to your computer. (The external oscillator portion of the circuit is optional when using MPLAB ICE.)
5. Run the project on the emulator. If you are running Windows 98(with the USB upgrade), Windows NT, or Windows 2000 then your operating system will detect a new device and install the necessary drivers automatically. After this occurs you should see the cursor rotating in a small circle on your screen. To stop the cursor from rotating, press F5.

Emulation Tips

1. Turn on the emulator, then open the project to initialize the emulator. If the project is in a mode other than Emulation follow step 2.
2. If your project is already open before you turn the emulator on you can initialize it by first, turning the emulator on. Then go to the option menu and click Development Mode. Make sure that the MPLAB-ICE Emulator radio button is selected in the Tools folder of the Development Mode dialog. Click OK. The emulator will initialize.
3. Make sure the proper processor module and device adapter are installed.
4. Should you have other problems please refer to the MPLAB-ICE Users Guide as your first resource.



Note: Be sure to pull D- up to V_{USB} (via R1) *not* V_{DD}. For more on why this is done see the section **Universal Serial Bus:Transceiver:Regulator** in the PIC16C745/65 datasheet.