##

PIC32MZ HighSpeed USB HOST program
manual

July 2025 Suwa-Koubou

##

1. Introduction

The USB module built into the PIC32MZ is an excellent module that supports high speeds and is designed for use with a hub. However, the only way to use it is to use the Harmony library provided by Microchip. However, even moving a single mouse requires a large amount of source code.

I've always wanted to create my own host program to replace Harmony.
I couldn't understand anything from the documentation for the USB module available from Microchip, and I had given up on the idea of building one myself.

It seems there are others overseas who share the same thoughts. After searching the internet, I found several sample programs for creating device programs. As for host programs, I was able to find a USB keyboard host program created by aidanmoche.

https://www.aidanmocke.com
https://www.aidanmocke.com/blog/2024/05/16/usb-host-hid-keyboard-code/ He devoted himself to analyzing the Harmony source code and created the host program himself.
I'm grateful that you've made the source code public.

I downloaded it and tried it out straight away, and when I operated the keyboard, the characters I typed were displayed on the screen, confirming that the USB host function was working correctly. Unfortunately, there were some issues, such as it ceasing to work halfway through when I changed to a different keyboard, but what impressed me was that it gave me a great hint for investigating in detail the function and setting methods of the USB hardware module registers, something I had always wanted to know.

Although the host hardware module is designed to use a hub, a separate hub driver must be created. Fortunately, the hub driver for PIC32MX is available.

It's already been created, so it seems like it would be fine to use this.

In this way, we were able to create a USB host program with the following functions:
Ta.

- Built-in HUB driver
- Hubs ranging from USB 1.1 to USB 3.0 are available.
  (However, the maximum speed is limited to high speed.)
- The HUB supports up to four ports (can be changed to support six
ports). - Supports multiple interfaces and devices.
  (For a 4-port hub, the number of devices is limited to 5, including the hub.)
•Supported device types
  Mouse, keyboard, game controller, HID generic, MIDI,
  USB serial converter, USB memory (USB card reader)

* Things that are not realized in the current version
- UVC devices such as web cameras are not supported.
(I am considering porting the UVC driver for PIC32MX that I created in the past)
- A HUB connected to a HUB port cannot be used.
(This is done to avoid overly complex program design and to take into consideration the needs of the user.)
DMA is not used for data transfer with the USB module's FIFO. (This is due to my
lack of knowledge. I would like to try it someday.)

Below is a brief explanation of the source code for the host program we created.
Please use this as a reference when deciphering or modifying the code.

The source code consists of declarations followed by class drivers, core drivers, and hub drivers.
For class drivers, see PIC32MZ **HighSpeed USB HOST Application Interface**
Please refer to the following explanation.

2. Main declarations and macros

//#define DEBUG_MSG_PRINT Outputs a debug message using printf().
//#define DEBUG_DATA_PRINT Prints a hexadecimal representation of received data, including the DESCRIPTOR.
//#define DEBUG_DUMP_PRINT Prints the contents of DESCRIPTOR. It is
commented out, so please uncomment it if necessary.

```
#define MAXINTERFACE            4       // One device, 4 interfaces
#define HUBMAXPORT              4       // Supports up to 4 ports
```

(Note)

To support a 6-port HUB, set HUBMAXPORT to 6. However, if you connect up to 6 devices, the HOST side endpoints (described later) may be insufficient and the HUB may not function properly. There may not be.

```
typedef struct usb_device_t {
    uint16_t VId; //                  // vendor id copy from DEVICE DESCRIPTOR
    uint16_t PId; // product id copy from DEVICE DESCRIPTOR
    uint8_t Address; 1 = direct device or HUB,
                                     // 2- MAXPORT+1 = HUB port connected device
    uint8_t Speed;                   // 3=low, 2=full, 1=high
    uint8_t Attached;                // 0=detached, 1=attached
    uint8_t Ready;                   // 0=not ready, 1= ready
    uint8_t interfaceNums; // number of has interfaces
    struct interface_t {
        uint8_t Class;               // from Class to OUTinterval,
                                     // copy from CONFIGURATION_DESCRIPTOR
        uint8_t SubClass;
        uint8_t Protocol;
        uint8_t interfaceNumber;
        uint8_t INEpNum;
        uint16_t INEpSize;
        uint8_t INEpProto;
        uint8_t INinterval;
        uint8_t OUTEpNum;
        uint16_t OUTEpSize;
        uint8_t OUTEpProto;
        uint8_t OUTinterval;
        uint8_t HostEpNum; // assigned HOST side endpoint number
    } interface[MAXINTERFACE];
} USB_DEVICE;
```

Manages information about connected USB devices. The contents of each variable are as described in the comments. Information about unsupported interface classes and alternative interfaces is not recorded.

```
typedef struct {
    uint8_t nums;                                   // number of devices
    uint8_t port[HUBMAXPORT];                       // usbDevice[] index
    uint8_t interfaceNumber[HUBMAXPORT]; // index of usbDevice[port].interface[]
    uint8_t option[HUBMAXPORT];                     // this field used serial driver
```

```
                                                    // and msc driver
} DRIVER_INFO;
```

For each supported device, it contains a link to information about the USB device.

```
typedef struct product_t {
        uint16_t VId;
        uint16_t PId;
        uint8_t Class;
        uint8_t SubClass;
        uint8_t Protocol;
} PRODUCT;
```

Information about supported vendor class products. Converts
vendor class (0xFF) to its equivalent Class.
The list is generated using PRODUCT product_table[];

```
typedef struct support_class_t {
        int class;
        int subclass;
        int protocol;
} STANDARD;
```

The code for the standard class that is supported.
Generate a code listing with STANDARD class_table[];

3. Main global variables

All variables are static and cannot be accessed from application programs.

```
USB_DEVICE        usbDevice[HUBMAXPORT+1];
```

This is an array variable that manages information about connected USB devices.
- Information about the device connected to the USB connector is stored in usbDevice[0].
- Information about devices connected to the HUB port is stored in usbDevice[port number].

uint8_t cnctDeviceNums; This holds the
number of currently connected USB devices. The initial value is 0.
The HUB itself is not counted. If nothing is connected to the HUB port, the count is 0.
- When a supported USB device is connected to a USB connector or a HUB port, the count

I'll upload it.
- Countdown will start when the USB device is removed.

uint8_t usbDeviceAttach; •It becomes 1
when a hub or other USB device is connected to the USB connector. •It becomes 0 if nothing is
connected. •It becomes 0 when a connected
device is removed.
- Updated during the USB module interrupt process.

uint8_t hubAttached;
•When a hub is connected to the USB connector, this value becomes 1. The initial value is 0.

PRODUCT product_table[]; An
array containing the vendor IDs and product IDs of supported vendor class devices.
- ID information for YAMAHA and ROLAND MIDI devices, FTDI USB serial conversion devices, etc. is stored.
It is delivered.
This is used to determine whether a connected USB device is a supported device.

STANDARD class_table[];
Stores the class code, subclass code, and protocol code of the supported standard classes.
This is an
array that is referenced when determining whether a connected USB device is a supported device.

DRIVER_INFO mouse_driver, keubpard_driver, ... msc_driver - Driver
information for supported classes. - Each contains a
link to the usbDevice[] variable.

4. Application Interface Functions

void usb_host_init(void); Initializes
the USB host module. The application program
must call this function once before using USB.
- Work
variables are cleared, USB module registers are cleared, and USB interrupt processing is initialized.
I am.

int isUsbConnect(void);
- Returns the value of cnctDeviceNums.
- This function is called periodically to connect or remove USB devices. The application program must call this function from within the application loop.

5. USB Device Management

5-1. USB Device Connection Processing

The process from when a USB device is connected to a connector until it can be used is as follows: It will be.

When the host is initialized by the usb_host_init() function, it starts monitoring the connection and removal of USB devices to the USB connector, and if there is a change in the connector, a USB interrupt occurs. During the interrupt process, if a connection is detected, the usbDeviceAttach flag is set to 1 and the detach is If a removal is detected, it is set to 0.

On the other hand, the isUsbConnect() function performs the following process each time it is called by an application.

When the usbDeviceAttach flag is 1 and usbDevice[0].Ready is 0, the device enumeration process begins. If the enumeration process is successful, usbDevice[0].Ready becomes 1. If a hub connection is confirmed during the enumeration process, the hubAttached flag becomes 1.

When the usbDeviceAttach flag is 1, if usbDevice[0].Ready is already 1,
The installation process is now complete, so next we will check the HUB connection.
When the hubAttached flag is 0, a device other than a HUB is connected, so you can continue If not, the function will terminate and cnctDeviceNums will remain 1.
If the hubAttached flag is 1, the hub_loop() function is called.
If a device is detected, it will be enumerated.

5-2. USB device enumeration process

USB device enumeration is performed by calling the device_enumeration() function. The same function is used for devices connected to USB connectors and devices connected to hub ports.

First, read the DEVICE_DESCRIPTOR using the get_device_descriptor() function and set information such as VId and PId in the usbDevice[port] variable.

If the connected device is a HUB class device and the hubAttached flag is 0,
The hub_initialize() function is called to initialize the HUB.
The hubAttached flag is set to 1 and usbDevice[0].Ready is also set to 1.

If the hubAttached flag is already set to 1, cascading of HUBs is not supported.
Since the device is not connected, the enumeration process ends here. Since in this case ,
usbDevice[port].Ready remains 0, the enumeration fails.

If a device other than a HUB is connected, a SET_ADDRES request is issued to the device.
Set the address. The device address is 1 for the device connected to the USB connector,
For devices connected to a hub port, the port number is +1.

Next , a SET_CONFIG request is issued to activate the USB device .
Set usbDevice[port].Ready to 1 to end the enumeration process.

5-3. Processing after enumeration

If the enumeration process is successful, the analys_interface_class() function is called to
Determines whether the device is portable.

Issue a GET_CONFIG request to obtain the configuration descriptor and interpret its contents.
Analyzes
all interfaces in the configuration.
Determine whether the device corresponds to the supported class based on the crypter class code.

The class code is If it is 0xFF (vendor class), refer to product_table[], otherwise
is determined by referencing class_table[]. If the device is supported,
Reads endpoint information from an endpoint descriptor.
Store information such as interface number and endpoint number in the usbDevice[].interface[] variable.
Store it in

Next, the host side endpoints that communicate with each endpoint for each interface of the device
The endpoint number to be assigned is the endpoint number on the device side.
The number does not necessarily match the IN endpoint number of the newly connected mouse device.
Even if the value is 1, the endpoint number 1 on the host side is already assigned to communication with another device.
If the endpoint is already configured, it will be assigned the second endpoint.
After allocating the endpoint, call the set_port_device_endpoint() function to set the allocated
For the endpoint you have selected, specify the device information, such as the FIFO address and FIFO size of the USB module.

This sets the registers required for communication with the endpoint on the client side.

To manage input/output by device class, the device interface information is
Register it in the management variable of the class you want to use (such as mouse_driver or midi_driver).
Finally, cnctDeviceNums is incremented by 1.

Once the above process is complete, the application can communicate with the device via API functions.
It will be possible.

5-4. USB device removal process

When a device connected to a USB connector is removed, the USB interrupt is
The usbDeviceAttach flag is set to 0.

When the isUsbConnect() function is called, the usbDeviceAttach flag is 0.
If usbDevice[0].Ready is 1, the device removal process will be performed.
If usbDevice[0].Ready is 0, the device is not connected or the removal process has completed.
So I won't do anything.

When a device connected to a HUB port is removed, the HUB itself remains connected to the USB connector, so the usbDeviceAttach flag remains 1. In this case, the hub_loop() function is called, the device removal is detected, and the removal process is carried out.

The process of removing a device connected to a USB connector is performed by calling the usb_host_init() function.
The device is removed and cnctDeviceNums is set to 0.
When you remove a device connected to a HUB port, only the information related to that device is displayed.
cnctDeviceNums will count down by one.

6. HUB driver processing

6-1. HUB initialization

The hub_initialize() function performs the following processing:

First issue a SET_ADDRESS request to set the address, followed by a SET_CONFIG request.
Activate the HUB with est.
A CONFIGURATION DESCRIPTOR is obtained with a GET_CONFIG request to obtain interface information and endpoint information.
A GET_HUB_DESCRIPTOR request is then issued to obtain information about the ports on the HUB.
This will tell you the number of ports on the HUB.

Then issue a SET_PORT_FEATURE request to power on each port.
The HUB is a HighSpeed hub with class code (9), subclass code (0), and protocol code (2).
If you have a Multi Transaction Translator interface,
Issue a SET_INTERFACE request to use that interface.

Set the variables in usbDevice[0] appropriately.

The HUB notifies the IN endpoint when there is a change in the port status (connection or removal of a device).
The host is notified using the set_port_device_endpoint() To receive this notice ,
function to start the receiving process.

Finally, set the hubAttached flag to 1 to complete the initialization.

6-2. HUB port handling

The isUsbConnect() function calls the hub_loop() function if a HUB is connected.

The hub_loop() function first checks whether there has been a change in the status of the HUB
port. If there has been a change in the port, it calls the port_loop() function for the port that has changed.
I will.

The port_loop() function performs the necessary processing depending on the change in port state.
For example, if a connection is detected, reset the port.

The results of each process are reported to the isUsbConnect() function through the following variables:

hubPortNumber: Notifies the port number that has changed
hubPortChanged: Notifies of changes

The changes are as follows:
PORT_DEVICE_ATTACHED ... Device attached
PORT_DEVICE_DETACED ... Device removed
PORT_DEVICE_NONE ... No change

7. Detecting USB device connection speed

The connection speed of the device connected to the USB connector can be known from the USBOTG register.
When
a low-speed device is connected, USBOTGbits.LSDEV is set to 1.

When a standard or high-speed device is connected, USBOTGbits.LSDEV is 0.

To determine whether the device is a full-speed or high-speed device, perform the following after a bus reset:
You can tell by looking at the HSMODE bit in USBCSR0. If USBCSR0.HSMODE is 1, it is high speed.
Device connection.

On the other hand, for devices connected to a HUB port, the port status indicating the status of the HUB port is displayed.
This can be determined by the change in the status bit.

When a device is connected to the port, if PORT_LOW_SPEED_BIT is 1, it is a low-speed device.
If PORT_HIGH_SPEED_BIT is 1, it is a high-speed device connection.
If both bits are 0, a full speed device is connected.

8. Automatic reception by interrupt

For USB communications with a packet size of 64 bytes or less, data reception is automatic.
Received packets are stored in a ring buffer.

Received data read from the application (such as calling the usb_mouse_read() function) is
Reads data from the buffer.

Automatic reception is initiated within the set_port_device_endpoint() function.

The code is as follows, where hep is the host endpoint number.

```
// start receive auto request
USB_EPN_RX_IE |= (1<<hep);              // HOST endpoint enable RX interrupt
IEC4bits.USBIE = 0;
resetRxQueue(hep);                      // clear ring buffer
IEC4bits.USBIE = 1;
*((int32_t *)&USBE0CSR0 + hep * 4) &= ~(0x20000000);
                                        // USBIENCSR0bits.MODE = 0; Set RX mode
*((int32_t *)&USBE1CSR1 + (hep-1) * 4) |= 0x200000;
                                        // USBIENCSR1bits.REQPKT = 1; IN transaction start
```

When a packet is received, a USB interrupt occurs and the call_back_EPN_read() function is executed from the interrupt handler.
is called and the packets are stored in the ring buffer.

9. Control Transfer

Control transfer is usb_ctrl(uint8_t port, uint8_t addr, uint8_t * ctrl, uint8_t
*data).

port is the port number of the HUB to which the device is connected. addr is the address set for the device. ctrl is the request packet such as GET_DESCRIPTOR. data is the data to be sent or the receive buffer according to the request.

A control transfer consists of three stages: SETUP stage, DATA stage, and STATUS stage. Control transfers are performed on endpoint 0 for all devices.

In the SETUP stage, the usb_setup() function is called to send a SETUP packet on EP0.

In the DATA stage, depending on the contents of the request packet, if data is to be received, usb_ep0_read() is called to execute an IN transaction, and if data is to be sent, usb_ep0_write() is called to execute an OUT transaction. If there is no data to send or receive, there is no DATA stage.

In the STATUS stage, if the DATA stage was an IN transaction, the received response
The source code is as follows:

No data is set in the TXFIFO.
Turn on the STATUS and TXPKTRDY bits in the USBE0CSR0 register.
*((uint8_t *)&USBE0CSR0 + 0x2) = 0x42; // STATUS + TXPKTRDY

If the DATA stage is an OUT transaction, or if there is no DATA stage, null data is received from the device in response to the DATA packet or SETUP packet. The source code is as follows:

Turn on the STATUS and REQPKT bits in the USBE0CSR0 register
*((uint8_t *)&USBE0CSR0 + 0x2) = 0x60; // Set STATUS and REQPKT, no data stage. When an empty data packet is received, clear the STATUS and RXPKTRDY bits. *((uint8_t *)&USBE0CSR0 + 0x2) &= ~0x41; // Clear STATUS and RXPKTRDY
There is no process to read data from the RXFIFO.

10. Bulk Interrupt Transfer

An IN transaction is performed by calling the usb_read() function. The
endpoint numbers on the host side that communicate with the usb_read() function are 1 to 7.
By turning on USBIENCSR1bits.REQPKT, an IN token is sent and data is received from the device.
When this is received, a receive interrupt occurs.

After the received data is removed from the FIFO, the USBIENCSRD1bits.RXPKTRDY bit is cleared. The reception process is then completed.

An OUT transaction is performed by calling the usb_write() function. The endpoint numbers on the host side that communicate with the usb_write() function are 1 to 7. After storing the data to be transmitted in the FIFO, turning on the USBIENCSR0bits.TXPKTRDY bit starts the transmission. If the TXPKTRDY bit is cleared within a certain period of time, the transmission is successful.

If the amount of data to be sent is larger than the packet size, it will be divided into packets of the same size and sent.

The speed and address of the communication destination device for IN and OUT transactions are set by the set_port_device_endpoint() function call executed at the end of the enumeration process, so there is no need to set them in usb_read() or usb_write().

11. Isochronous Transfer

Isochronous transfers are not required for the USB devices currently supported.

Supports UVC devices such as webcams and audio streams.
If supported, isochronous transfers are required.

Coding the transfer process for isochronous transfers is almost the same as for bulk and interrupt transfers, but care must be taken with packet size. With isochronous transfers, the transfer packet size changes dynamically as the alternative interface is switched. The current set_port_device_endpoint() function does not support dynamic changes in packet size. Also, the usbDevice[] variable does not currently store any information about alternative interfaces. Some modifications are required to add isochronous transfer processing.

End