##

PIC32MZ HighSpeed USB HOST program
説明書

2025.7　Suwa-Koubou

##

I.はじめに

PIC32MZ に内蔵の USB モジュールは、ハイスピード対応で HUB の利用も考慮された優れたモジュールですが、使うにあたっては、Microchip 社が提供するライブラリー Harmony を使うしかありません。この Harmony ですが、マウス一つを動かすにも多量のソースコードを必要とします.

Harmony に取って代わるホストプログラムを自作してみたいと常々思っていたのですが、Microchip 社から入手できる USB モジュールのドキュメントだけでは、全く理解出来ず、自作はほぼ不可能と諦めていました。

海外でも同じような思いを持つ他人がいるようです。ネットを色々探してみるとデバイスのプログラム作成については、サンプルプログラムが幾つか見つかりました。ホストプログラムについては、aidanmoche さんが作られた USB キーボードのホストプログラムを見つけることが出来ました。

https://www.aidanmocke.com
https://www.aidanmocke.com/blog/2024/05/16/usb-host-hid-keyboard-code/
Harmony のソースコードの解析に尽力され自力でホストプログラムを作られたとのことです。ソースコードを公開して下さっていることに感謝します。

早速にダウンロードして試してみましたところ、キーボードを操作すると打鍵した文字が画面に表示され、USB のホスト機能が正しく動いていることが確認出来ました。残念ながらキーボードを他のものに変更すると、途中から動かなくなるなどの不備もありましたが、私が感動したのは、常々知りたかった USB ハードウェアモジュールのレジスタの働きや設定方法を詳しく調べるための大いなるヒントが得られたことです。

ホストのハードウエアモジュールは HUB の利用を考慮しているとはいうものの別途 HUB ドライバーを作成しなければなりません。幸い、HUB ドライバーについては PIC32MX のものを

作成済みですので、こちらを流用すればよさそうです。

　こうして、次に示すような機能を持つUSBホストプログラムを作り上げることが出来ました。

・HUBドライバーを内蔵
・HUBは、USB1.1仕様のものからUSB3.0仕様のものまで利用可能.
（ただし最大速度はハイスピードまでです）
・HUBは、４ポートまで対応.（６ポート対応へ変更可能）
・マルチインタフェース、マルチデバイスに対応
（デバイスの数は、4ポートHUBの場合、HUBを含めて5台までとなります）
・サポートするデバイスの種類
マウス、キーボード、ゲームコントローラー、HIDジェネリック、MIDI、
USBシリアル変換、USBメモリー（USBカードリーダー）

※ 現バージョンで実現していない事
・webカメラなどのUVCデバイスは未サポートです。
（過去に作成したPIC32MX用のUVCドライバーのポーティングを検討中です）
・HUBのポートに接続したHUBは使えません。
（プログラム設計を殊更複雑にしないためとニーズを考慮した結果です）
・USBモジュールのFIFOとのデータ転送にDMAは使っていません.
（私の知識不足です.いつか挑戦したいと思っています）

　以下、今回作成しましたホストプログラムのソースコードについて簡単な説明を記載しました。解読や改造される場合の参考にしてください.

　ソースコードは、宣言に続いてクラスドライバ、コアドライバ、ハブドライバで構成しています。クラスドライバーについては、PIC32MZ HighSpeed USB HOST Application Interfaceを参照してください.以下の説明には記載していません.

2.主な宣言やマクロ

```
//#define DEBUG_MSG_PRINT     printf()によるデバッグメッセージを出力します.
//#define DEBUG_DATA_PRINT    DESCRIPTORなどの受信データの16進表示を出力します.
//#define DEBUG_DUMP_PRINT    DESCRIPTORの内容を出力します.
```
　コメントアウトしていますので、必要に応じてアンコメントしてください.

```
#define   MAXINTERFACE       4       // One device 4 interface
#define   HUBMAXPORT         4       // support up to 4 port
```

（註）
　6ポートHUBに対応するには HUBMAXPORT を 6にします.
　ただし、最大6台のデバイスを接続するとHOST側のエンドポイント(後述)が不足して動かない場合があります.

```
typedef struct usb_device_t {
    uint16_t VId;            // vender 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;
```

　接続しているUSBデバイスの情報を管理します.各変数の内容はコメントの通りです.
　未サポートのクラスのインターフェースや代替インタフェースの情報は記録されません.

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

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

サポートするデバイス毎に、USB デバイスの情報へのリンクを保有します.

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

サポートするベンダークラスの製品についての情報です.
ベンダークラス(0xFF)を等価な Class クラスに変換します.
PRODUCT product_table[]; でリストを生成しています.

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

サポートする標準クラスのコードです.
STANDARD class_table[]; でコードリストを生成します.

3.主なグローバル変数

全て static 変数です. アプリケーションプログラムからはアクセス出来ません.

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

・接続している USB デバイスの情報を管理する配列変数です.
・ USB コネクタに接続したデバイスの情報は usbDevice[0] に格納されます.
・ HUB のポートに接続したデバイスの情報は usbDevice[ポート番号] に格納されます.

```
uint8_t       cnctDeviceNums;
```

・現在接続している USB デバイスの数を保持しています. 初期値は 0 です.
・ HUB 自体はカウントされません.HUB のポートに何も接続されていなければ 0 です.
・ USB コネクタや HUB のポートに、サポート可能な USB デバイスが接続されるとカウントア

ップします.
・USB デバイスが取り外されるとカウントダウンします.

uint8_t　　usbDeviceAttach;
・USB コネクタに HUB やその他の USB デバイスが接続されると 1 になります.
・何も接続されていなければ 0 です.
・接続されていたデバイスが取り外されると 0 になります.
・USB モジュールの割込み処理の中で更新されます.

uint8_t　　hubAttached;
・USB コネクタに HUB が接続されると 1 になります. 初期値は 0 です.

PRODUCT　　product_table[];
・サポートするベンダークラスデバイスのベンダー ID、プロダクト ID を格納した配列です.
・YAMAHA や ROLAND の MIDI デバイス、FTDI の USB シリアル変換デバイスなどの ID 情報が格納されています.
・接続された USB デバイスがサポート可能なデバイスかどうか判定する時に参照されます.

STANDARD　　class_table[];
・サポートする標準クラスのクラスコード、サブクラスコード、プロトコルコードを格納した配列です.
・接続された USB デバイスがサポート可能なデバイスかどうか判定する時に参照されます.

DRIVER_INFO　mouse_driver, keubpard_driver, ... msc_driver
・サポートするクラスのドライバー情報です.
・実体は、usbDevice[] 変数へのリンクを格納しています.

4.アプリケーションインタフェース関数

void usb_host_init(void);
・USB ホストモジュールを初期化します.
・アプリケーションプログラムは USB の使用に先立ってこの関数を一度呼び出さなければなりません.
・ワーク変数のクリア、USB モジュールのレジスタをクリア、USB 割込み処理の初期化を行います.

int isUsbConnect(void);
・cnctDeviceNums の値を返します.
・この関数が定期的に呼び出されることで、USB デバイスの接続や取り外しの処理を実行します. アプリケーションプログラムは、アプリケーションループの中からこの関数を呼び出さなければなりません.

5.USB デバイスの管理

5-1.USB デバイスの接続処理

USB デバイスがコネクタに接続されてから使えるようになるまでの処理の流れは次のようになります.

usb_host_init()関数によりホストが初期化されると、USB コネクタへの USB デバイスの接続・取り外しが監視されるようになり、コネクタに変化があれば USB の割込みが発生します. 割込み処理の中では、接続が検出された時は、usbDeviceAttach フラグに 1 を設定し、取り外しが検出された時には 0 を設定します.

一方、isUsbConnect()関数は アプリケーションから呼び出される度に次の処理を行います.

usbDeviceAttach フラグが 1 の時、 usbDevice[0].Ready が 0 であれば、デバイスのエヌメレーション処理を行います. エヌメレーション処理が成功すると usbDevice[0].Ready は 1 になります. エヌメレーション処理の中で HUB の接続が確認された場合には、更に hubAttached フラグが 1 になります.

usbDeviceAttach フラグが 1 の時、 usbDevice[0].Ready が既に 1 であれば、エヌメレーション処理は終了していますので、次に HUB の接続を確認します.
hubAttached フラグが 0 の時は、HUB 以外のデバイスが接続されているので、このまま何もしないで 関数を終了します. cnctDeviceNums は 1 のままです.
hubAttached フラグが 1 であれば、hub_loop()関数を呼び出します. HUB のポートに USB デバイスの接続が検出された場合には、そのデバイスのエヌメレーション処理を行います.

5-2.USB デバイスのエヌメレーション処理

USB デバイスのエヌメレーション処理は、 device_enumeration() 関数を呼び出して行われます.USB コネクタに接続したデバイスも HUB のポートに接続したデバイスも同じ関数で処理します.

最初に get_device_descriptor() 関数にて DEVICE_DESCRIPTOR を読み込み usbDevice[port] 変数の VId や PId などの情報を設定します.

接続されたデバイスがHUBクラスのデバイスで、hubAttached フラグが 0 であれば、hub_initialize()関数を呼び出して HUB の初期化を行います.HUB の初期化処理の中で hubAttached フラグが 1 に設定されるとともに usbDevice[0].Ready も 1 に設定されます.

既に hubAttached フラグが 1 になっていた場合には、HUB のカスケード接続はサポートしていませんのでエヌメレーション処理をここで終了します.この場合、usbDevice[port].Ready は 0 のままなのでエヌメレーションは失敗となります.

HUB 以外のデバイスが接続された場合には、SET_ADDRES リクエストを発行してデバイスにアドレスを設定します.デバイスのアドレスは、USB コネクタに接続したデバイスは 1 で、HUB のポートに接続したデバイスの場合は、ポート番号+1 になります.

次に SET_CONFIG リクエストを発行して USB デバイスを活性化します.usbDevice[port].Ready に 1 を設定してエヌメレーション処理を終了します.

5-3.エヌメレーション以後の処理

エヌメレーション処理が成功すると次に analys_interface_class()関数を呼び出してサポート可能なデバイスかどうか判定します.

GET_CONFIG リクエストを発行し、コンフィグレーションディスクリプタを取得し内容を解析します.
コンフィグレーションに含まれる全てのインタフェースについて、インタフェースディスクリプタのクラスコードを元にサポートするクラスに該当するデバイスかどうか判定します.
クラスコードが 0xFF(ベンダークラス)なら product_table[] を参照し、その他の場合には class_table[] を参照して判定します. 判定の結果、サポート可能なデバイスであれば、エンドポイントディスクリプタからエンドポイント情報を読み取ります.
インタフェース番号やエンドポイント番号などの情報を usbDevice[].interface[] 変数に格納します.

続いて、デバイスの各インタフェース毎の各エンドポイントと通信するホスト側のエンドポイントを割り当てます. 割り当てるエンドポイント番号は、デバイス側のエンドポイント番号とは必ずしも一致しません. 新たに接続したマウスデバイスの IN エンドポイント番号が 1 であっても、ホスト側のエンドポイント番号 1 が他のデバイスとの通信に既に割り当てられていると、2 番のエンドポイントが割り当てられたりします.
エンドポイントの割り当ての後、set_port_device_endpoint()関数を呼び出して、割り当てたエンドポイントについて、USB モジュールの FIFO アドレスや FIFO サイズなど、デバイ

ス側のエンドポイントとの通信に必要なレジスタの設定を行います.

デバイスのクラス別の入出力管理を行うために、デバイスのインタフェース情報を、該当するクラスの管理変数(mouse_driver や midi_driver など)に登録します.
最後に cnctDeviceNums を 1 カウントアップします.

以上までの処理が完了するとアプリケーションから API 関数を通してデバイスとの通信が可能になります.

5-4.USB デバイス取り外しの処理

USB コネクタに接続していたデバイスが取り外されると、USB の割込み処理の中で usbDeviceAttach フラグが 0 に設定されます.

isUsbConnect()関数が呼び出された時、usbDeviceAttach フラグが 0 であった時、usbDevice[0].Ready が 1 であればデバイス取り外しの処理を行います。
usbDevice[0].Ready が 0 であれば、未接続の状態であるか、取り外しの処理が完了していることになりますので何もしません.

HUB のポートに接続したデバイスが取り外された時は、HUB 自体は USB コネクタに接続したままですので usbDeviceAttach フラグは 1 のままです。この場合には、 hub_loop()関数が呼び出され、デバイスの取り外しが検出されて取り外しの処理が行われます。

USB コネクタに接続したデバイスの取り外し処理は、usb_host_init()関数を呼び出すことで取り外しの処理としています. cnctDeviceNums は 0 になります.
HUB のポートに接続したデバイスの取り外しでは、そのデバイスに関連する情報のみをクリアします. cnctDeviceNums は 1 カウントダウンします.

6.HUB ドライバーの処理
6-1.HUB の初期化

hub_initialize()関数は次のような処理となっています。

最初に SET_ADDRESS リクエストを発行して アドレスを設定し、続けて SET_CONFIG リクエストで HUB を活性化します.
GET_CONFIG リクエストで CONFIGURATION DESCRIPTOR を取得して、インタフェース情報、エンドポイント情報を取得します.
続けて GET_HUB_DESCRIPTOR リクエストを発行して、HUB の持つポートの情報を取得します。これにより HUB のポート数を知ることが出来ます。

次に SET_PORT_FEATURE リクエストを発行して、各ポートの電源をオンにします。
HUB がクラスコード(9)、 サブクラスコード(0)、プロトコルコード(2)の HighSpeed Multi Transaction Translator インタフェースを有していた場合には、
SET_INTERFACE リクエストを発行して、そのインタフェースを使うようにします.

usbDevice[0]の各変数を適切に設定します.

HUB はポートの状態に変化（デバイスの接続や取り外し）があった時には IN エンドポイントを使ってホストに通知してくれます。この通知が受け取れるように、set_port_device_endpoint()関数を呼び出して受信の処理を起動します.

最後に hubAttached フラグに 1 を設定して初期化を終了します.

6-2.HUB のポートの処理

isUsbConnect()関数は、HUB が接続されていると hub_loop()関数を呼び出します。

hub_loop()関数は、最初に HUB のポートに状態変化があったかどうかを確認します.
ポートに変化があった場合には、変化のあったポートに対して port_loop()関数を呼びだします。

port_loop()関数では、ポートの状態変化に応じて、それぞれ必要な処理を行います。
例えば、接続が検出された場合には、ポートをリセットする、などです.

それぞれの処理結果は、次の変数を通じて isUsbConnect()関数に通知されます.

hubPortNumber: 変化のあったポート番号を通知
hubPortChanged: 変化内容を通知

変化内容は次のとおりです.

```
PORT_DEVICE_ATTACHED  ... デバイス接続
PORT_DEVICE_DETACED   ... デバイス取り外し
PORT_DEVICE_NONE      ... 変化なし
```

7.USB デバイス接続速度の検出

USB コネクタに接続されたデバイスの接続速度は、USBOTG レジスタによって知ることが出来ます.
ロースピードデバイスが接続されると、USBOTGbits.LSDEV が 1 になります. フルスピー

ドもしくはハイスピードデバイスが接続されると USB0TGbits.LSDEV は 0 です.
フルスピードデバイスかハイスピードデバイスかの判定は、バスリセットの後に、USBCSR0 の HSMODE ビットを見ることで判ります. USBCSR0.HSMODE が 1 ならハイスピードデバイスの接続です.

一方、HUB のポートに接続したデバイスについては、HUB のポートの状態を示す ポートステータスビットの変化で知ることが出来ます.
ポートにデバイスが接続された時、PORT_LOW_SPEED_BIT が 1 なら ロースピードデバイスの接続です。 PORT_HIGH_SPEED_BIT が 1 なら ハイスピードデバイスの接続です。どちらの ビットも 0 なら フルスピードデバイスの接続です。

8.割込みによる自動受信

USB のパケットサイズが 64 バイト以下の通信については、データ受信は自動で行います. 受信したパケットはリングバッファに格納されます.
アプリケーションからの受信読み出し(usb_mouse_read()関数などの呼び出し)は リングバッファからデータを読み出します.

自動受信の起動は set_port_device_endpoint()関数の中で行っています。

コードは次の通りです. hep はホスト側のエンドポイント番号です.

```
// 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
```

、パケットを受信すると USB 割込みが発生し、割込み処理から call_back_EPN_read() 関数が呼び出されリングバッファにパケットが格納されます.

9. コントロール転送

コントロール転送は、usb_ctrl(uint8_t port, uint8_t addr, uint8_t * ctrl, uint8_t *data) を呼び出して行います.

port は、デバイスを接続しているHUBのポート番号です．addr は、そのデバイスに設定されているアドレスです。 ctrl は、GET_DESCRIPTOR などのリクエストパケットです．data はリクエストに応じた送信データ、もしくは受信バッファとなります．

コントロール転送は、SETUP ステージ，DATA ステージ，STATUS ステージの３ステージで構成されます． コントロール転送は、全てのデバイスに対してエンドポイント0で実行します．

SETUPステージでは、usb_setup()関数を呼び出して EP0 で SETUPパケットを送信します．

DATAステージでは、リクエストパケットの内容により、データを受信する場合には、usb_ep0_read()を呼び出して INトランザクションを、データを送信する場合にはusb_ep0_write()を呼び出して OUTトランザクションを実行します.送受信するデータが無い時は、DATAステージは有りません．

STATUSステージでは、DATAステージがINトランザクションであった場合には、受信応答として空データを送信します。そのソースコードは次のようになります．

```
TXFIFOにデータは設定しません．
USBE0CSR0 レジスタの STATUS ビットと TXPKTRDY ビットをオンにします．
 *((uint8_t *)&USBE0CSR0 + 0x2) = 0x42; // STATUS + TXPKTRDY
```

DATAステージがOUTトランザクション、あるいはDATAステージが無かった場合には、DATAパケットまたはSETUPパケットの送信応答としてデバイスから空データの受信を行います．そのソースコードは次のようになります．

```
USBE0CSR0 レジスタの STATUS ビットと REQPKT ビットをオンにします
 *((uint8_t *)&USBE0CSR0 + 0x2) = 0x60; // Set STATUS and REQPKT, no data stage
空のデータパケットを受信したら STATUSビットと RXPKTRDYビットをクリアします．
 *((uint8_t *)&USBE0CSR0 + 0x2) &= ~0x41; // Clear STATUS and RXPKTRDY
RXFIFOからデータを読む処理はありません．
```

10.バルク・インタラプト転送

INトランザクションは、 usb_read()関数を呼び出して行います．
usb_read()関数で通信するホスト側のエンドポイント番号は 1 から 7 になります。
USBIENCSR1bits.REQPKTをオンにすることでINトークンが送信され、デバイスからデータを受信すると受信割込みが発生します。

受信したデータを FIFO から取り出した後、USBIENCSRDIbits.RXPKTRDY ビットをクリアして受信処理終了となります。

OUT トランザクションは、usb_write()関数を呼び出して行います.
usb_write()関数で通信するホスト側のエンドポイント番号は　1 から 7 になります。
送信するデータを FIFO に格納した後、USBIENCSR0bits.TXPKTRDY ビットをオンにすると送信を開始します. 一定時間内に TXPKTRDY ビットがクリアされれば送信成功です.
送信するデータ数がパケットサイズより大きい時は、パケットサイズに分割して送信します.

IN トランザクションや OUT トランザクションの通信先デバイスの速度やアドレスなどの設定については、エヌメレーション処理の最後に実行した set_port_device_endpoint()関数の呼び出しによって済んでいますので、usb_read()や usb_write()の中では設定不要です.

II.イソクロナス転送

今回サポートしている USB デバイスの範疇ではイソクロナス転送は必要ありません。

web カメラなどの UVC デバイスをサポートする場合や Audio の音楽データのストリームをサポートする場合にはイソクロナス転送が必要になります。
イソクロナス転送の転送処理のコーディングは、バルクやインタラプトとほとんど変わらないと思いますが、パケットサイズに関して注意が必要となります。イソクロナス転送では、代替インタフェースの切替などに伴い、転送するパケットサイズが動的に変化します。現在の set_port_device_endpoint()関数では、パケットサイズの動的変更に対応出来ていません。また、現在は usbDevice[] 変数には 代替インタフェースの情報が一切格納されていません。
イソクロナス転送の処理を追加する場合には何らかの改造が必要です。

以上