

L'USB en bref

Donner un sens au standard USB

L'Énumération

L'énumération est la manière de déterminer l'appareil qui vient juste d'être branché au bus et les paramètres dont il a besoin, comme la consommation électrique, le nombre et le type de terminaison, la classe du produit, etc.... L'hôte attribuera donc à l'appareil une adresse et validera une configuration lui permettant de transférer des données sur le bus. Un assez bon processus d'énumération générique est détaillé en section 9.1.2 de la spécification USB. Toutefois lorsque l'on écrit un microprogramme USB pour la première fois, il est plus pratique de connaître la manière dont l'hôte répond pendant l'énumération, plutôt que le processus d'énumération général détaillé dans la spécification.

Une énumération sous Windows ordinaire implique les étapes suivantes :

- L'hôte ou Hub détecte la connexion d'un nouvel appareil via les résistances de rappel de l'appareil reliées sur les 2 fils de données. L'hôte attend au moins 100ms, le temps que la prise soit insérée complètement et que l'alimentation de l'appareil soit stabilisée.
- L'hôte émet un " reset " mettant l'appareil dans l'état par défaut. L'appareil peut maintenant répondre à l'adresse zéro par défaut.
- L'hôte (sous MS Windows) demande les 64 premiers octets du descripteur d'appareil.
- Après avoir reçu les 8 premiers octets du descripteur d'appareil, l'hôte émet immédiatement un autre reset sur le bus.
- L'hôte émet maintenant une commande *SetAdress*, mettant l'appareil dans l'état adressable.
- L'hôte demande la totalité des 18 octets du descripteur d'appareil.
- Puis il demande les 9 octets du descripteur de configuration pour déterminer la taille totale.
- L'hôte demande les 255 octets du descripteur de configuration.
- L'hôte demande l'un des descripteurs de chaînes s'ils étaient indiqués.

A la fin de l'étape 9, " Windows " demandera un driver (pilote logiciel) pour votre appareil. Il est alors courant de le voir redemander tous les descripteurs avant d'émettre une requête *SetConfiguration*.

Le processus d'énumération ci-dessus est courant dans Windows 2000, Windows XP et Windows 98 SE.

L'étape 4 embarrasse souvent les gens qui écrivent des microprogrammes pour la première fois. L'hôte demande les 64 premiers octets du descripteur d'appareil, aussi lorsque l'hôte met à zéro votre appareil après avoir reçu les 8 premiers octets, il est tout à fait naturel de penser qu'il y a un problème soit au niveau du descripteur d'appareil soit dans la façon dont votre microprogramme manipule la requête. Cependant, comme vous le diront beaucoup de gens, si vous insistez sur la mise en œuvre de la commande *SetAdress*, elle sera récompensée par la demande suivante de 18 octets pleins du descripteur d'appareil.

Généralement quand il y a un problème avec le descripteur ou sur la façon dont il est envoyé, l'hôte tentera de le lire 3 fois avec de longues pauses entre les requêtes. Après la troisième tentative, l'hôte abandonne signalant une erreur au niveau de l'appareil.

Microprogramme. Le PIC 16F876 pilotant le PDIUSB11

Microprogramme. Le PIC 16F876 pilotant le PDIUSB11 Nous commençons nos exemples avec le circuit USB PDIUSB11 à liaison série I2C de Philips relié au microcontrôleur de Microchip, le PIC 16F876 représenté ici ou bien le PIC 16F877 (circuit à 40 broches). Tandis que Microchip a 2 circuits USB basse vitesse sur le marché actuellement, le PIC 16C745 et le PIC 16C765, ils sont seulement OTP (One Time Programmable= programmable une seule fois) sans le soutien du circuit interne de débogage (ICD) qui n'apporte pas vraiment d'aide au débit de la réalisation. Ils possèdent 4 nouveaux circuits " Flash " pleine vitesse avec le support de l'ICD qui devraient sortir. En attendant, on trouve le PDIUSB11 de Philips relié au PIC 16F876 qui a l'avantage d'être Flash et d'avoir l'ICD .

Un schéma électronique de base est représenté ci-dessus.

L'exemple énumère et permet à des tensions analogiques d'être mesurées à partir des 5 entrées ADC multiplexées du microcontrôleur (MCU) PIC 16F876. Le code est compatible avec le PIC 16F877 permettant un maximum de 8 voies analogiques. Une LED reliée à la broche RB3 s'allume quand le circuit est configuré. Un régulateur de 3,3V n'est pas dessiné, mais il est nécessaire au PDIUSB11. Si vous alimentez le circuit exemple d'une alimentation externe, alors vous pouvez utiliser un vulgaire régulateur de tension 78L033 de 3,3V, cependant si vous voulez faire fonctionner le circuit en tant qu'appareil alimenté par le bus, un régulateur à faible chute de tension se révélera obligatoire.

Le débogage peut être fait en connectant TXD (broche 17) à un câble RS232 et branché à un PC configuré à 115 200 bauds. Des formulations "*printf*" ont été incorporées permettant l'affichage de la progression de l'énumération.

Le code a été écrit en C et compilé avec le compilateur [Hi-Tech PICC](#). Ils ont une [version de démonstration \(7.86 PL4\)](#) de PICC à télécharger utilisable pendant 30 jours. Un fichier Hex pré compilé inclus en archive a été compilé pour être utilisé avec (ou sans) l'ICD.

```
#include <pic.h>
#include <stdio.h>
#include <string.h>
#include "usbfull.h"

const USB_DEVICE_DESCRIPTOR DeviceDescriptor = {
    sizeof(USB_DEVICE_DESCRIPTOR), /* bLength */
    TYPE_DEVICE_DESCRIPTOR,       /* bDescriptorType */
    0x0110,                        /* bcdUSB USB Version 1.1 */
    0,                             /* bDeviceClass */
    0,                             /* bDeviceSubclass */
    0,                             /* bDeviceProtocol */
    8,                             /* bMaxPacketSize 8 Bytes */
    0x04B4,                        /* idVendor (Cypress Semi) */
    0x0002,                        /* idProduct (USB Thermometer Example) */
    0x0000,                        /* bcdDevice */
    1,                             /* iManufacturer String Index */
    0,                             /* iProduct String Index */
    0,                             /* iSerialNumber String Index */
    1                             /* bNumberConfigurations */
};
```

Les " structures " sont toutes définies dans les fichiers d'en tête. Nous avons basé cet exemple sur l'exemple du thermomètre USB de Cypress pour que vous puissiez utiliser [le driver USB du kit de démarrage de Cypress](#). Un nouveau driver générique qui prendrait ceci en charge est en phase d'écriture et d'autres exemples seront bientôt disponibles. Une seule chaîne de caractère est fournie pour afficher le nom du fabricant. Cela donne suffisamment d'informations sur la façon de mettre en œuvre des descripteurs de chaînes sans remplir le composant dans sa totalité avec du code. On peut trouver [ici](#) une description de descripteur d'appareil et de ses champs.

```

const USB_CONFIG_DATA ConfigurationDescriptor = {
    {
        /* configuration descriptor */
        sizeof(USB_CONFIGURATION_DESCRIPTOR), /* bLength */
        TYPE_CONFIGURATION_DESCRIPTOR, /* bDescriptorType */
        sizeof(USB_CONFIG_DATA), /* wTotalLength */
        1, /* bNumInterfaces */
        1, /* bConfigurationValue */
        0, /* iConfiguration String Index */
        0x80, /* bmAttributes Bus Powered, No Remote Wakeup */
        0x32 /* bMaxPower, 100mA */
    },
    {
        /* interface descriptor */
        sizeof(USB_INTERFACE_DESCRIPTOR), /* bLength */
        TYPE_INTERFACE_DESCRIPTOR, /* bDescriptorType */
        0, /* bInterface Number */
        0, /* bAlternateSetting */
        2, /* bNumEndpoints */
        0xFF, /* bInterfaceClass (Vendor specific) */
        0xFF, /* bInterfaceSubClass */
        0xFF, /* bInterfaceProtocol */
        0 /* iInterface String Index */
    },
    {
        /* endpoint descriptor */
        sizeof(USB_ENDPOINT_DESCRIPTOR), /* bLength */
        TYPE_ENDPOINT_DESCRIPTOR, /* bDescriptorType */
        0x01, /* bEndpoint Address EP1 OUT */
        0x02, /* bmAttributes - Interrupt */
        0x0008, /* wMaxPacketSize */
        0x00 /* bInterval */
    },
    {
        /* endpoint descriptor */
        sizeof(USB_ENDPOINT_DESCRIPTOR), /* bLength */
        TYPE_ENDPOINT_DESCRIPTOR, /* bDescriptorType */
        0x81, /* bEndpoint Address EP1 IN */
        0x02, /* bmAttributes - Interrupt */
        0x0008, /* wMaxPacketSize */
        0x00 /* bInterval */
    }
};

```

Une description du descripteur de configuration et de ses champs figure [ici](#). Nous fournissons 2 descripteurs de terminaison en plus du canal de communication par défaut. EP1 OUT est une terminaison OUT de Bloc de 8 octets maximum et EP1 IN OUT est une terminaison IN de Bloc de 8 octets maximum. Notre exemple lit les données de la terminaison OUT de Bloc et les met dans un tampon mémoire circulaire de 80 octets. L'envoi d'un paquet IN à EP1 permettra de lire un morceau de 8 octets du tampon mémoire circulaire.

```

LANGID_DESCRIPTOR LANGID_Descriptor = { /* LANGID String Descriptor Zero */
    sizeof(LANGID_DESCRIPTOR),          /* bLenght */
    TYPE_STRING_DESCRIPTOR,             /* bDescriptorType */
    0x0409                               /* LANGID US English */
};

const MANUFACTURER_DESCRIPTOR Manufacturer_Descriptor = { /* ManufacturerString 1 */
    sizeof(MANUFACTURER_DESCRIPTOR),    /* bLenght */
    TYPE_STRING_DESCRIPTOR,             /* bDescriptorType */
    "B\0e\0y\0o\0n\0d\0 \0L\0o\0g\0i\0c\0" /* ManufacturerString in
UNICODE */
};

```

Un descripteur de chaîne d'index zéro est formé pour supporter les besoins de LANGID des descripteurs de chaînes USB. Celui-ci indique que tous les descripteurs sont en anglais US. Le descripteur de constructeur peut être un peu décevant parce que la taille du tableau de caractères est fixée dans l'en tête et n'est pas dynamique.

```

#define MAX_BUFFER_SIZE 80

bank1 unsigned char circularbuffer[MAX_BUFFER_SIZE];
unsigned char inpointer;
unsigned char outpointer;

unsigned char *pSendBuffer;
unsigned char BytesToSend;
unsigned char CtlTransferInProgress;
unsigned char DeviceAddress;
unsigned char DeviceConfigured;

#define PROGRESS_IDLE          0
#define PROGRESS_ADDRESS      3

void main (void)
{
    TRISB = 0x03; /* Int & Suspend Inputs */
    RB3 = 1;     /* Device Not Configured (LED) */
    RB2 = 0;     /* Reset PDIUSB11 */

    InitUART();
    printf("Initialising\n\r");
    I2C_Init();

    RB2 = 1;     /* Bring PDIUSB11 out of reset */

    ADCON1 = 0x80; /* ADC Control - All 8 Channels Enabled, */
                /* supporting upgrade to 16F877 */

    USB_Init();
    printf("PDIUSB11 Ready for connection\n\r");
    while(1) D11GetIRQ();
}

```

La fonction principale dépend de l'exemple. Elle est responsable de l'initialisation de la direction des E/S des Ports, de l'initialisation de l'interface I2C, des Convertisseurs Analogiques Digitaux et du PDIUSB11. Une fois que tout ceci est configuré, elle continue d'interroger D11GetIRQ qui traite les requêtes d'interruption du PDIUSB11.

```

void USB_Init(void)
{
    unsigned char Buffer[2];

    /* Disable Hub Function in PDIUSBD11 */
    Buffer[0] = 0x00;
    D11CmdDataWrite(D11_SET_HUB_ADDRESS, Buffer, 1);

    /* Set Address to zero (default) and enable function */
    Buffer[0] = 0x80;
    D11CmdDataWrite(D11_SET_ADDRESS_ENABLE, Buffer, 1);

    /* Enable function generic endpoints */
    Buffer[0] = 0x02;
    D11CmdDataWrite(D11_SET_ENDPOINT_ENABLE, Buffer, 1);

    /* Set Mode - Enable SoftConnect */
    Buffer[0] = 0x97; /* Embedded Function, SoftConnect, Clk Run, No LazyClk, Remote
Wakeup */
    Buffer[1] = 0x0B; /* CLKOut = 4MHz */
    D11CmdDataWrite(D11_SET_MODE, Buffer, 2);
}

```

La fonction USB " init " initialise le PDIUSBD11. Cette procédure d'initialisation a été omise de la documentation technique du PDIUSBD11 de Philips mais elle est disponible de leur [FAQ](#) (Foire Aux Questions). La dernière commande valide la connexion logicielle de la résistance de rappel sur D+ signalant que c'est un appareil pleine vitesse mais aussi prévient de sa présence sur le Bus Série Universel.

```

void D11GetIRQ(void)
{
    unsigned short Irq;
    unsigned char Buffer[1];

    do {
        /* Read Interrupt Register to determine source of interrupt */

        D11CmdDataRead(D11_READ_INTERRUPT_REGISTER, (unsigned char *)&Irq, 2);

        if (Irq) printf("Irq = 0x%X: ", Irq);
    }
}

```

La fonction **main()** continue d'appeler D11GetIRQ en boucle. Cette fonction lit les registres d'interruption du PDIUSBD11 pour établir si des interruptions sont en suspens. Si c'est le cas, elle les traitera, sinon elle continuera sa boucle. D'autres appareils peuvent avoir plusieurs vecteurs d'interruptions relatif à chaque terminaison. Dans ce cas, chaque ISR (*Interrupt Service Routine je pense !*) traitera l'interruption appropriée enlevant les formulations "if".

```

    if (Irq & D11_INT_BUS_RESET) {
        printf("Bus Reset\n\r");
        USB_Init();
    }

    if (Irq & D11_INT_EP0_OUT) {
        printf("EP0_Out: ");
        Process_EP0_OUT_Interrupt();
    }
}

```

```

    if (Irq & D11_INT_EP0_IN) {
        printf("EP0_In: \n\r");
        if (CtlTransferInProgress == PROGRESS_ADDRESS) {
            D11CmdDataWrite(D11_SET_ADDRESS_ENABLE, &DeviceAddress, 1);
            D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_IN, Buffer,
1);

            CtlTransferInProgress = PROGRESS_IDLE;
        }
        else {
            D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_IN, Buffer,
1);

            WriteBufferToEndPoint();
        }
    }

```

Les formulations "*if*" apparaissent par ordre de priorité. L'interruption prioritaire la plus haute est le reset du bus. Elle appelle simplement USB_Init qui réinitialise la fonction USB. La priorité suivante est le canal de communication par défaut constitué de EP0 OUT et de EP1 IN. C'est ici que sont envoyées l'énumération et toutes les requêtes de commandes. Nous bifurquons sur une autre fonction pour manipuler les requêtes de EP0 OUT.

Quand une requête est faite par l'hôte et qu'il veut recevoir des données, le PIC 16F876 enverra au PDIUSB11 un paquet de 8 octets. Comme le bus USB est contrôlé par l'hôte, le PDIUSB11 ne peut écrire les données quand il le désire, aussi il les stocke et attend un jeton IN provenant de l'hôte. Quand le PDIUSB11 reçoit le jeton IN, il provoque une interruption. Cela laisse un temps suffisant pour recharger le prochain paquet de données à envoyer. Ceci est réalisé par une fonction supplémentaire : **WriteBufferToEndpoint()**.

La section incluse dans CtlTransferInProgress == PROGRESS_ADDRESS maintient le positionnement de l'adresse de l'appareil. Nous détaillerons ceci plus tard.

```

    if (Irq & D11_INT_EP1_OUT) {
        printf("EP1_OUT\n\r");
        D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP1_OUT, Buffer, 1);
        bytes = D11ReadEndpoint(D11_ENDPOINT_EP1_OUT, Buffer);
        for (count = 0; count < bytes; count++) {
            circularbuffer[inpointer++] = Buffer[count];
            if (inpointer >= MAX_BUFFER_SIZE) inpointer = 0;
        }
        loadfromcircularbuffer(); //Kick Start
    }

    if (Irq & D11_INT_EP1_IN) {
        printf("EP1_IN\n\r");
        D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP1_IN, Buffer, 1);
        loadfromcircularbuffer();
    }

```

EP1 OUT et EP1 In sont mis en place pour lire et écrire des données par Blocs à destination et en provenance d'un tampon mémoire circulaire. Cette configuration permet au code d'être utilisé en adéquation avec l'exemple BulkUSB provenant du DDK de Windows. Le tampon mémoire circulaire a été défini plus haut dans le code comme comportant 80 octets de long et prenant toute la banque1 de la RAM du 16F876.

```

if (Irq & D11_INT_EP2_OUT) {
    printf("EP2_OUT\n\r");
    D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP2_OUT, Buffer, 1);
    Buffer[0] = 0x01; /* Stall Endpoint */
    D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP2_OUT, Buffer, 1);
}

if (Irq & D11_INT_EP2_IN) {
    printf("EP2_IN\n\r");
    D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP2_IN, Buffer, 1);
    Buffer[0] = 0x01; /* Stall Endpoint */
    D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP2_IN, Buffer, 1);
}

if (Irq & D11_INT_EP3_OUT) {
    printf("EP3_OUT\n\r");
    D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP3_OUT, Buffer, 1);
    Buffer[0] = 0x01; /* Stall Endpoint */
    D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP3_OUT, Buffer, 1);
}

if (Irq & D11_INT_EP3_IN) {
    printf("EP3_IN\n\r");
    D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP3_IN, Buffer, 1);
    Buffer[0] = 0x01; /* Stall Endpoint */
    D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP3_IN, Buffer, 1);
}
} while (Irq);
}

```

Les terminaisons 2 et 3 ne sont pas utilisés pour l'instant, aussi nous les bloquons si elles reçoivent une donnée quelconque. Le PDIUSB11 a une commande SetEndpointEnable qui peut être utilisée pour valider ou dé valider la fonction de terminaison générique (n'importe qu'elle terminaison autre que le canal de communication par défaut). Nous pourrions utiliser cette commande pour dé valider les terminaisons de base, si nous envisageons de ne pas nous en servir plus tard. Toutefois pour l'instant, ce code fournit une base de travail.

```

void Process_EP0_OUT_Interrupt(void)
{
    unsigned long a;
    unsigned char Buffer[2];
    USB_SETUP_REQUEST SetupPacket;

    /* Check if packet received is Setup or Data - Also clears IRQ */
    D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_OUT, &SetupPacket, 1);

    if (SetupPacket.bmRequestType & D11_LAST_TRAN_SETUP) {

```

La première chose à faire est de déterminer si le paquet reçu sur EP0 est un paquet de données ou un paquet d'installation. Un paquet d'installation contient une requête telle que GetDescriptor, tandis qu'un paquet de données contient des données d'une requête précédente. Nous avons de la chance que la plupart des requêtes n'envoient pas des paquets de données provenant de l'hôte vers l'appareil. La seule requête qui le réalise est SET_DESCRIPTOR mais elle est rarement mise en œuvre.

```

/* This is a setup Packet - Read Packet */
D11ReadEndpoint(D11_ENDPOINT_EP0_OUT, &SetupPacket);

/* Acknowledge Setup Packet to EP0_OUT & Clear Buffer*/
D11CmdDataWrite(D11_ACK_SETUP, NULL, 0);
D11CmdDataWrite(D11_CLEAR_BUFFER, NULL, 0);

/* Acknowledge Setup Packet to EP0_IN */
D11CmdDataWrite(D11_ENDPOINT_EP0_IN, NULL, 0);
D11CmdDataWrite(D11_ACK_SETUP, NULL, 0);

/* Parse bmRequestType */
switch (SetupPacket.bmRequestType & 0x7F) {

```

Comme nous l'avons vu dans notre description de transferts de commande, un paquet d'installation ne peut pas être Non ACQuité (NAKed) ou Bloqué (STALLED). Quand le PDIUSB11 reçoit un paquet d'installation, il vide le tampon EPO IN et dé valide les commandes du tampon mémoire de validation et celles du tampon mémoire d'effacement. Cela garantit que le paquet d'installation est acquitté par le microcontrôleur en envoyant un acquittement de commande d'installation à EPO IN et EPO OUT avant qu'une commande de validation ou d'effacement de tampon mémoire ne soit effective. La réception d'un paquet d'installation déblocuera aussi une terminaison de commande bloquée (STALLED).

Une fois le paquet installé dans la mémoire et le paquet d'installation validé, nous allons analyser la requête en commençant par son type. A ce moment nous ne nous intéressons pas à la direction, aussi nous masquons ce bit. Les 3 requêtes que tout appareil doit traiter sont la requête standard d'appareil, la requête standard d'interface et les requêtes standard de terminaisons. Nous fournissons notre fonctionnalité (lecture des entrées analogiques) par la requête constructeur, aussi nous ajoutons un type d'instruction pour les requêtes standard de constructeur. Si votre appareil supporte une spécification de classe USB, alors vous pouvez aussi avoir besoin d'ajouter des cas pour la requête de classe d'appareil, la requête de classe d'interface et/ou de la requête de classe de terminaison.

```

case STANDARD_DEVICE_REQUEST:
    printf("Standard Device Request ");
    switch (SetupPacket.bRequest) {
        case GET_STATUS:
            /* Get Status Request to Device should return */
            /* Remote Wakeup and Self Powered Status */
            Buffer[0] = 0x01;
            Buffer[1] = 0x00;
            D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
            break;

        case CLEAR_FEATURE:
        case SET_FEATURE:
            /* We don't support DEVICE_REMOTE_WAKEUP or TEST_MODE */
            ErrorStallControlEndPoint();
            break;
    }

```

La requête GetStatus est utilisée pour rapporter l'état de l'appareil et dire par exemple si l'appareil est alimenté par le bus ou auto alimenté et s'il supporte le réveil à distance. Dans notre appareil, nous rapportons qu'il est auto alimenté et qu'il ne prend pas en charge le réveil à distance.

D'après la requête caractéristique d'appareil, l'appareil ne supporte ni DEVICE_REMOTE_WAKEUP ni TEST_MODE et retourne comme résultat une erreur de requête USB.

```
case SET_ADDRESS:
    printf("Set Address\n\r");
    DeviceAddress = SetupPacket.wValue | 0x80;
    DllWriteEndpoint(Dll_ENDPOINT_EP0_IN, NULL, 0);
    CtlTransferInProgress = PROGRESS_ADDRESS;
    break;
```

La commande SetAddress est la seule commande qui continue d'être traitée après l'étape d'état. Toutes les autres commandes doivent finir le traitement avant l'étape d'état. L'adresse de l'appareil est lue puis combinée par un OU logique avec 0x80 et ensuite stockée dans une variable DeviceAddress. La combinaison par un OU logique avec 0x80 avec le bit le plus significatif indiquant si l'appareil est validé ou non est spécifique au PDIUSB11. Un paquet de longueur nul est renvoyé comme état à l'hôte, indiquant que la commande est achevée. Toutefois l'hôte doit envoyer un jeton IN, récupérer le paquet de longueur nul et délivrer un ACK avant que l'on puisse changer d'adresse. Autrement l'appareil ne constatera peut être jamais que le jeton IN a été envoyé à l'adresse par défaut.

L'achèvement de l'étape d'état est signalé par une interruption sur EPO IN. Afin de différencier une réponse à SetAddress et une interruption normale EPO_IN, nous positionnons une variable, CtlTransferInProgress à PROGRESS_ADDRESS. Le programme de traitement de EPO IN prévoit une vérification de CtlTransferInProgress. Si elle équivaut à PROGRESS_ADDRESS la commande Set Address Enable est alors délivrée au PDIUSB11 et CtlTransferInProgress est positionné à PROGRESS_IDLE. L'hôte donne 2ms à l'appareil pour changer d'adresse avant que la prochaine commande ne soit envoyée.

```
case GET_DESCRIPTOR:
    GetDescriptor(&SetupPacket);
    break;

case GET_CONFIGURATION:
    DllWriteEndpoint(Dll_ENDPOINT_EP0_IN, &DeviceConfigured, 1);
    break;

case SET_CONFIGURATION:
    printf("Set Configuration\n\r");
    DeviceConfigured = SetupPacket.wValue & 0xFF;
    DllWriteEndpoint(Dll_ENDPOINT_EP0_IN, NULL, 0);
    if (DeviceConfigured) {
        RB3 = 0;
        printf("\n\r *** Device Configured *** \n\r");
    }
    else {
        RB3 = 1; /* Device Not Configured */
        printf("\n\r ** Device Not Configured *** \n\r");
    }
    break;

//case SET_DESCRIPTOR:
default:
    /* Unsupported - Request Error - Stall */
    ErrorStallControlEndPoint();
    break;
}
break;
```

GetConfiguration et SetConfiguration sont utilisés pour " valider " l'appareil USB permettant aux données d'être transférées sur les terminaisons autres que la terminaison zéro. SetConfiguration devrait être délivré avec wValue égal à bConfigurationValue de la configuration que vous voulez valider. Dans notre cas, nous n'avons qu'une configuration, la configuration 1. Une configuration de valeur zéro signifie que l'appareil n'est pas configuré tandis qu'une configuration de valeur différente de zéro indique que l'appareil est configuré. Le code ne recopie pas complètement la valeur de configuration, il la recopie seulement dans une variable locale (de stockage) appelée DeviceConfigured. Si la valeur dans wValue ne correspond pas à bConfigurationValue d'une configuration, elle devrait renvoyer le message : Erreur de Requête USB.

```

case STANDARD_INTERFACE_REQUEST:
    printf("Standard Interface Request\n\r");
    switch (SetupPacket.bRequest) {

        case GET_STATUS:
            /* Get Status Request to Interface should return */
            /* Zero, Zero (Reserved for future use) */
            Buffer[0] = 0x00;
            Buffer[1] = 0x00;
            DllWriteEndpoint(Dll_ENDPOINT_EP0_IN, Buffer, 2);
            break;

        case SET_INTERFACE:
            /* Device Only supports default setting, Stall may be */
            /* returned in the status stage of the request */
            if (SetupPacket.wIndex == 0 && SetupPacket.wValue == 0)
                /* Interface Zero, Alternative Setting = 0 */
                DllWriteEndpoint(Dll_ENDPOINT_EP0_IN, NULL, 0);
            else ErrorStallControlEndPoint();
            break;

        case GET_INTERFACE:
            if (SetupPacket.wIndex == 0) { /* Interface Zero */
                Buffer[0] = 0; /* Alternative Setting */
                DllWriteEndpoint(Dll_ENDPOINT_EP0_IN, Buffer, 1);
                break;
            } /* else fall through as RequestError */

        //case CLEAR_FEATURE:
        //case SET_FEATURE:
            /* Interface has no defined features. Return RequestError */
        default:
            ErrorStallControlEndPoint();
            break;
    }
    break;

```

Parmi les requêtes standard d'interface, aucune n'exécute vraiment de fonction. La requête GetStatus doit retourner un message composé de zéros et est réservée pour un usage ultérieur. Les requêtes SetInterface et GetInterface sont utilisées avec les descripteurs alternatifs d'interfaces. Jusqu'ici nous n'avons défini aucun descripteur alternatif d'interface, donc GetInterface renvoie un zéro et toutes les requêtes pour mettre à 1 une interface autre que l'interface zéro avec un positionnement alternatif de zéro se traduiront par une Erreur de Requête.

```

case STANDARD_ENDPOINT_REQUEST:
    printf("Standard Endpoint Request\n\r");
    switch (SetupPacket.bRequest) {

        case CLEAR_FEATURE:
        case SET_FEATURE:
            /* Halt(Stall) feature required to be implemented on all
Interrupt and */
            /* Bulk Endpoints. It is not required nor recommended on the
Default Pipe */

            if (SetupPacket.wValue == ENDPOINT_HALT)
            {
                if (SetupPacket.bRequest == CLEAR_FEATURE) Buffer[0] = 0x00;
                else Buffer[0] = 0x01;
                switch (SetupPacket.wIndex & 0xFF) {
                    case 0x01 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
D11_ENDPOINT_EP1_OUT, Buffer, 1);
                    break;
                    case 0x81 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
D11_ENDPOINT_EP1_IN, Buffer, 1);
                    break;
                    case 0x02 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
D11_ENDPOINT_EP2_OUT, Buffer, 1);
                    break;
                    case 0x82 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
D11_ENDPOINT_EP2_IN, Buffer, 1);
                    break;
                    case 0x03 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
D11_ENDPOINT_EP3_OUT, Buffer, 1);
                    break;
                    case 0x83 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
D11_ENDPOINT_EP3_IN, Buffer, 1);
                    break;
                    default : /* Invalid Endpoint - RequestError */
ErrorStallControlEndPoint();
                    break;
                }
                D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
            } else {
                /* No other Features for Endpoint - Request Error */
                ErrorStallControlEndPoint();
            }
            break;

```

Les requêtes SetFeature et ClearFeature sont utilisés pour positionner des fonctions spécifiques de terminaison. Le standard définit un sélecteur de fonction de terminaison : ENDPOINT_HALT. Nous vérifierons vers quelle terminaison la requête est dirigée et positionnerons/effacerons le bit STALL en conséquence. La fonction HALT n'est pas exigée sur les terminaisons par défaut.

```

case GET_STATUS:
    /* Get Status Request to Endpoint should return */
    /* Halt Status in D0 for Interrupt and Bulk */
    switch (SetupPacket.wIndex & 0xFF) {
        case 0x01 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
D11_ENDPOINT_EP1_OUT, Buffer, 1);
        break;

```

```

        case 0x81 : DllCmdDataRead(D11_READ_ENDPOINT_STATUS + \
            D11_ENDPOINT_EP1_IN, Buffer, 1);
            break;

        case 0x02 : DllCmdDataRead(D11_READ_ENDPOINT_STATUS + \
            D11_ENDPOINT_EP2_OUT, Buffer, 1);
            break;
        case 0x82 : DllCmdDataRead(D11_READ_ENDPOINT_STATUS + \
            D11_ENDPOINT_EP2_IN, Buffer, 1);
            break;
        case 0x03 : DllCmdDataRead(D11_READ_ENDPOINT_STATUS + \
            D11_ENDPOINT_EP3_OUT, Buffer, 1);
            break;
        case 0x83 : DllCmdDataRead(D11_READ_ENDPOINT_STATUS + \
            D11_ENDPOINT_EP3_IN, Buffer, 1);
            break;
        default : /* Invalid Endpoint - RequestError */
            ErrorStallControlEndPoint();
            break;
    }
    if (Buffer[0] & 0x08) Buffer[0] = 0x01;
    else Buffer[0] = 0x00;
    Buffer[1] = 0x00;
    DllWriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
    break;

default:
    /* Unsupported - Request Error - Stall */
    ErrorStallControlEndPoint();
    break;
}
break;

```

Quand la requête GetStatus est dirigée sur la terminaison, elle retourne l'état de terminaison, c'est à dire, si elle est arrêtée ou non. De même que pour la requête de fonction de positionnement/effacement ENDPOINT_HALT, nous avons juste besoin de rapporter l'état des terminaisons génériques (standards).

Toutes les requêtes standard de terminaison non définies peuvent être manipulées par une erreur de requête USB.

```

case VENDOR_DEVICE_REQUEST:
case VENDOR_ENDPOINT_REQUEST:
    printf("Vendor Device bRequest = 0x%X, wValue = 0x%X, wIndex = 0x%X\n\r",
        \
        SetupPacket.bRequest, SetupPacket.wValue, SetupPacket.wIndex);
    switch (SetupPacket.bRequest) {

        case VENDOR_GET_ANALOG_VALUE:
            printf("Get Analog Value, Channel %x :", SetupPacket.wIndex &
0x07);

            ADCON0 = 0xC1 | (SetupPacket.wIndex & 0x07) << 3;
            /* Wait Acquisition time of Sample and Hold */
            for (a = 0; a <= 255; a++);
            ADGO = 1;
            while(ADGO);
            Buffer[0] = ADRESL;
            Buffer[1] = ADRESH;

```

```

        a = (Buffer[1] << 8) + Buffer[0];
        a = (a * 500) / 1024;
        printf(" Value = %d.%02d\n\r", (unsigned int)a/100, (unsigned
int)a%100);

        DllWriteEndpoint(Dll_ENDPOINT_EP0_IN, Buffer, 2);
        break;

```

Venons-en maintenant aux parties fonctionnelles de l'appareil USB. Les requêtes constructeur peuvent être imaginées par le concepteur. Nous avons imaginé 2 requêtes : `VENDOR_GET_ANALOG_VALUE` et `VENDOR_SET_RB_HIGH_NIBBLE`. `VENDOR_GET_ANALOG_VALUE` lit la valeur analogique de 10 bits du canal x imposé par `wIndex`. Elle est masquée avec `0x07` pour permettre 8 canaux possibles, supportant le PIC 16F877 plus grand si nécessaire. La valeur analogique est renvoyée dans un paquet de données de 2 octets.

```

        case VENDOR_SET_RB_HIGH_NIBBLE:
            printf("Write High Nibble of PORTB\n\r");
            PORTB = (PORTB & 0x0F) | (SetupPacket.wIndex & 0xF0);
            DllWriteEndpoint(Dll_ENDPOINT_EP0_IN, NULL, 0);
            break;

        default:
            ErrorStallControlEndPoint();
            break;
    }
    break;

```

`VENDOR_SET_RB_HIGH_NIBBLE` peut être utilisé pour positionner les bits factices de poids forts du `PORTB[3:7]`.

```

        default:
            printf("Unsupported Request Type 0x%X\n\r", SetupPacket.bmRequestType);
            ErrorStallControlEndPoint();
            break;
    }
} else {
    printf("Data Packet?\n\r");
    /* This is a Data Packet */
}
}

```

Tous les types de requêtes non supportés tels que la requête de classe d'appareil, la requête de classe d'interface, etc. se traduiront par une erreur de requête USB.

```

void GetDescriptor(PUSB_SETUP_REQUEST SetupPacket)
{
    switch((SetupPacket->wValue & 0xFF00) >> 8) {

        case TYPE_DEVICE_DESCRIPTOR:
            printf("\n\rDevice Descriptor: Bytes Asked For %d, Size of Descriptor
%d\n\r", \
                SetupPacket->wLength, DeviceDescriptor.bLength);
            pSendBuffer = (const unsigned char *)&DeviceDescriptor;
            BytesToSend = DeviceDescriptor.bLength;

```

```

        if (BytesToSend > SetupPacket->wLength)
            BytesToSend = SetupPacket->wLength;
        WriteBufferToEndPoint();
        break;

    case TYPE_CONFIGURATION_DESCRIPTOR:
        printf("\n\rConfiguration Descriptor: Bytes Asked For %d, Size of Descriptor
%d\n\r", \
            SetupPacket->wLength, sizeof(ConfigurationDescriptor));
        pSendBuffer = (const unsigned char *)&ConfigurationDescriptor;
        BytesToSend = sizeof(ConfigurationDescriptor);
        if (BytesToSend > SetupPacket->wLength)
            BytesToSend = SetupPacket->wLength;
        WriteBufferToEndPoint();
        break;

```

Les requêtes GetDescriptor impliquent des réponses plus grandes que la taille maximale limite du paquet de 8 octets de la terminaison. Par conséquent elles doivent être scindées en morceaux de 8 octets. Les 2 requêtes d'appareil et de configuration chargent l'adresse des descripteurs pertinents dans pSendBuffer et positionnent BytesToSend à la longueur du descripteur. La requête précisera aussi une longueur de descripteur dans wLength indiquant le maximum de données à envoyer. Dans chaque cas nous vérifions la longueur réelle par rapport à celle demandée par l'hôte et ajustons la taille si nécessaire. Ensuite nous appellerons WriteBufferToEndpoint qui charge les 8 premiers octets dans le tampon de terminaison et incrémenterons le pointeur qui sera prêt pour le prochain paquet de 8 octets.

```

    case TYPE_STRING_DESCRIPTOR:
        printf("\n\rString Descriptor: LANGID = 0x%04x, Index %d\n\r", \
            SetupPacket->wIndex, SetupPacket->wValue & 0xFF);
        switch (SetupPacket->wValue & 0xFF){

            case 0 : pSendBuffer = (const unsigned char *)&LANGID_Descriptor;
                    BytesToSend = sizeof(LANGID_Descriptor);
                    break;

            case 1 : pSendBuffer = (const unsigned char *)&Manufacturer_Descriptor;
                    BytesToSend = sizeof(Manufacturer_Descriptor);
                    break;

            default : pSendBuffer = NULL;
                    BytesToSend = 0;
        }
        if (BytesToSend > SetupPacket->wLength)
            BytesToSend = SetupPacket->wLength;
        WriteBufferToEndPoint();
        break;

```

Si des descripteurs de chaînes sont inclus, il doit y avoir la présence d'un descripteur de chaîne zéro qui détaille quelles langues sont supportées par l'appareil. N'importe quelle requête de chaînes différente de zéro a un langage ID spécifié dans wIndex indiquant quelle langue est supportée. Dans notre cas nous trichons quelque peu et ignorons la valeur de wIndex (LANGID) en renvoyant la chaîne, quelle que soit la langue demandée.

```

        default:
            ErrorStallControlEndPoint();
            break;
    }
}

void ErrorStallControlEndPoint(void)
{
    unsigned char Buffer[] = { 0x01 };
    /* 9.2.7 RequestError - return STALL PID in response to next DATA Stage Transaction
*/
    DllCmdDataWrite(Dll_SET_ENDPOINT_STATUS + Dll_ENDPOINT_EP0_IN, Buffer, 1);
    /* or in the status stage of the message. */
    DllCmdDataWrite(Dll_SET_ENDPOINT_STATUS + Dll_ENDPOINT_EP0_OUT, Buffer, 1);
}

```

Quand nous rencontrons une requête invalide, un paramètre invalide ou une requête que l'appareil ne prend pas en charge, nous devons prendre compte d'une erreur de requête. Ceci est défini dans la spécification 9.2.7. Une erreur de requête renverra un STALL PID en réponse à la prochaine transaction d'étape de données ou au cours de l'étape d'état du message.. Toutefois elle note que pour empêcher un trafic inutile sur le bus, l'erreur doit être reportée à la prochaine étape de données plutôt que d'attendre l'étape d'état.

```

unsigned char DllReadEndpoint(unsigned char Endpoint, unsigned char *Buffer)
{
    unsigned char DllHeader[2];
    unsigned char BufferStatus = 0;

    /* Select Endpoint */
    DllCmdDataRead(Endpoint, &BufferStatus, 1);

    /* Check if Buffer is Full */
    if(BufferStatus & 0x01)
    {
        /* Read dummy header - Dll buffer pointer is incremented on each read */
        /* and is only reset by a Select Endpoint Command */
        DllCmdDataRead(Dll_READ_BUFFER, DllHeader, 2);
        if(DllHeader[1]) DllCmdDataRead(Dll_READ_BUFFER, Buffer, DllHeader[1]);
        /* Allow new packets to be accepted */
        DllCmdDataWrite(Dll_CLEAR_BUFFER, NULL, 0);
    }
    return DllHeader[1];
}

void DllWriteEndpoint(unsigned char Endpoint, const unsigned char *Buffer, unsigned char Bytes)
{
    unsigned char DllHeader[2];
    unsigned char BufferStatus = 0;
    DllHeader[0] = 0x00;
    DllHeader[1] = Bytes;

    /* Select Endpoint */
    DllCmdDataRead(Endpoint, &BufferStatus, 1);
    /* Write Header */
    DllCmdDataWrite(Dll_WRITE_BUFFER, DllHeader, 2);
    /* Write Packet */
    if (Bytes) DllCmdDataWrite(Dll_WRITE_BUFFER, Buffer, Bytes);
    /* Validate Buffer */
    DllCmdDataWrite(Dll_VALIDATE_BUFFER, NULL, 0);
}

```

D11ReadEndpoint et D11WriteEndpoint sont des fonctions spécifiques au PDIUSB11. Le PDIUSB11 a 2 octets factices préfixant toute opération de lecture ou écriture de données. Le premier octet est réservé, tandis que le second octet indique le nombre d'octets reçus ou à transmettre. Ces 2 fonctions tiennent compte de cet en-tête.

```
void WriteBufferToEndPoint(void)
{
    if (BytesToSend == 0) {
        /* If BytesToSend is Zero and we get called again, assume buffer is smaller */
        /* than Setup Request Size and indicate end by sending Zero Length packet */
        D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
    } else if (BytesToSend >= 8) {
        /* Write another 8 Bytes to buffer and send */
        D11WriteEndpoint(D11_ENDPOINT_EP0_IN, pSendBuffer, 8);
        pSendBuffer += 8;
        BytesToSend -= 8;
    } else {
        /* Buffer must have less than 8 bytes left */
        D11WriteEndpoint(D11_ENDPOINT_EP0_IN, pSendBuffer, BytesToSend);
        BytesToSend = 0;
    }
}
```

Comme nous l'avons mentionné précédemment, WriteBufferToEndpoint est responsable du chargement des données dans PDIUSB11 en morceaux de 8 octets et de l'ajustement de la préparation des pointeurs pour le prochain paquet. Elle est appelée une fois par le programme de gestion de la requête pour charger les 8 premiers octets dans le tampon mémoire de la terminaison. L'hôte enverra donc un jeton IN, lira cette donnée et le PDIUSB11 générera une interruption. Le programme de gestion de EPO IN appellera donc WriteBufferToEndpoint pour charger le prochain paquet qui sera prêt pour le prochain jeton IN provenant de l'hôte.

Un transfert est considéré comme complet si tous les octets requis ont été lus, si le paquet est reçu avec une charge utile inférieure à bMaxPacketSize ou si un paquet de longueur nul est renvoyé. Par conséquent si le compteur BytesToSend atteint zéro, nous pouvons présumer que les données à envoyer représentaient un multiple de 8 octets et que nous enverrons un paquet de longueur nul pour indiquer les dernières données. Toutefois, s'il nous reste moins de 8 octets à envoyer, nous enverrons seulement les octets restants. Il n'est pas nécessaire de rembourrer les données avec des zéros.

```
void loadfromcircularbuffer(void)
{
    unsigned char Buffer[10];
    unsigned char count;

    // Read Buffer Full Status
    D11CmdDataRead(D11_ENDPOINT_EP1_IN, Buffer, 1);

    if (Buffer[0] == 0){
        // Buffer Empty
        if (inpointer != outpointer){
            // We have bytes to send
            count = 0;
            do {
                Buffer[count++] = circularbuffer[outpointer++];
                if (outpointer >= MAX_BUFFER_SIZE) outpointer = 0;
                if (outpointer == inpointer) break; // No more data
            } while (count < 8); // Maximum Buffer Size
        }
    }
}
```

```

        // Now load it into EP1_In
        DllWriteEndpoint(Dll_ENDPOINT_EP1_IN, Buffer, count);
    }
}

```

La routine `loadfromcircularbuffer()` gère le chargement de données dans le tampon mémoire de la terminaison EPO IN. On y fait normalement appel après une interruption EPO IN pour recharger le tampon mémoire afin d'être prêt pour le prochain jeton IN sur EP1. Cependant afin d'envoyer le premier paquet, nous avons besoin de charger les données avant la réception de l'interruption EP1 IN. Par conséquent on fait aussi appel à la routine après que les données soient reçues sur EP1 OUT.

En appelant aussi la routine du programme de gestion de EPO OUT, nous pouvons facilement écraser les données dans le tampon IN sans se préoccuper si elles ont été envoyées ou pas. Pour empêcher ceci, nous déterminons si le tampon mémoire EP1 IN est vide, avant de tenter de le recharger avec de nouvelles données.

```

void DllCmdDataWrite(unsigned char Command, const unsigned char *Buffer, unsigned char
Count)
{
    I2C_Write(Dll_CMD_ADDR, &Command, 1);
    if(Count) I2C_Write(Dll_DATA_ADDR_WRITE, Buffer, Count);
}

```

```

void DllCmdDataRead(unsigned char Command, unsigned char Buffer[], unsigned char Count)
{
    I2C_Write(Dll_CMD_ADDR, &Command, 1);
    if(Count) I2C_Read(Dll_DATA_ADDR_READ, Buffer, Count);
}

```

`DllCmdDataWrite` et `DllCmdDataRead` sont 2 fonctions spécifiques au `PDIUSB11` qui sont responsables de l'envoi de l'adresse/commande I2C en premier puis de l'envoi ou de la réception des données sur le bus I2C. Des fonctions bas niveau supplémentaires sont incluses dans le code source mais ne sont pas reproduites ici du fait de l'intention de se focaliser sur les détails spécifiques de l'USB.

Cet exemple peut être utilisé avec l'exemple `bulkUSB.sys` comme faisant partie de la DDK de windows. Pour charger le driver `bulkUSB.sys`, soit vous changez le code pour l'identifier avec un VID de `0x045E` et un PID de `0x930A` ou bien vous changez le fichier `bulkUSB.inf` accompagnant `bulkUSB.sys` pour correspondre à la combinaison VID/PID que vous utilisez dans cet exemple.

Il est alors possible d'utiliser le programme utilisateur en mode console, `rwbulk.exe` pour envoyer et recevoir des paquets du tampon mémoire circulaire. Utiliser:

```
rwbulk -r 80 -w 80 -c 1 -i 1 -o 0
```

pour envoyer des morceaux de données de 80 octets au `PIC16F876`. L'utilisation de charges utiles supérieures à 80 octets fera déborder le tampon mémoire circulaire de banque1 du PIC.

Cet exemple a été codé pour une lisibilité au détriment de la taille du code. La compilation vaut 3250 mots de la Flash (39% de la capacité du `PIC 16F876`).

Décharger le code source.

Version 1.2, 15ko.

Historique des révisions:

- 6 Avril 2002 - Version 1.2 - Accroissement de la vitesse I2C pour correspondre au commentaire. Amélioration du traitement de l'IRQ PDIUSB11.
- Le programme de gestion des routines en Bloc de EP1 IN et EP1 OUT et la possibilité de charger des descripteurs à partir de la Flash. révision... a été ajouté.
- 31 Decembre 2001 - Version 1.0.

Reconnaissance:

Une reconnaissance speciale à Michael DeVault de DeVasys Embedded Systems. Cet exemple a été basé sur le code de Michael et développé sans effort sur la carte de développement USB de DeVasys: USBLPT-PD11 avant d'être porté sur le PIC.