

**MINF**  
**Mise en œuvre**  
**des microcontrôleurs PIC32MX**

# **Chapitre 9**

## **Gestion du bus USB**

**✂ T.P. PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.71 mars 2019**



## CONTENU DU CHAPITRE 9

<b>9. Gestion du bus USB</b>	<b>9-1</b>
<b>9.1. Hardware USB sur le KIT PIC32MX95F512L</b>	<b>9-1</b>
<b>9.2. Utilisation d'un exemple</b>	<b>9-2</b>
9.2.1. Ouverture du projet et adaptation	9-2
9.2.1.1. Ouverture du projet	9-2
9.2.1.2. Renommage du projet	9-2
9.2.1.3. Choix de la configuration	9-3
9.2.2. Réduction à une seule configuration	9-4
9.2.3. Sélection du bsp du kit	9-5
9.2.4. Adaptation du device configuration pour le Kit	9-6
9.2.4.1. Section DEVCFG3	9-6
9.2.4.2. Section DEVCFG2	9-6
9.2.4.3. Section DEVCFG1	9-6
9.2.4.4. Section DEVCFG0	9-6
9.2.5. Génération du code	9-7
9.2.6. Test de compilation	9-7
9.2.7. Nettoyage du projet	9-7
9.2.8. Test de la communication USB	9-8
9.2.8.1. Test avec Putty	9-8
9.2.9. Détails de fonctionnement	9-10
9.2.9.1. Code	9-10
9.2.9.2. Machine d'état	9-13
9.2.9.3. Déroulement d'une lecture	9-14
9.2.9.4. Déroulement d'une écriture	9-14
9.2.9.5. Etat de l'USB	9-15
<b>9.3. Ajout de traitement particulier</b>	<b>9-16</b>
<b>9.4. Exemple d'application</b>	<b>9-17</b>
9.4.1. Initialisation	9-17
9.4.2. Affichage des messages reçus	9-17
9.4.3. Envoi des messages	9-18
9.4.3.1. Contenu initial APP_STATE_SCHEDULE_WRITE	9-18
9.4.3.2. Contenu modifié de APP_STATE_SCHEDULE_WRITE	9-19
9.4.4. Situation avec application Windows	9-20
<b>9.5. Conclusion</b>	<b>9-21</b>
<b>9.6. Historique des versions</b>	<b>9-21</b>
9.6.1. Version 1.5 avril 2015	9-21
9.6.2. Version 1.6 avril 2016	9-21
9.6.3. Version 1.7 mars 2017	9-21
9.6.1. Version 1.71 mars 2019	9-21



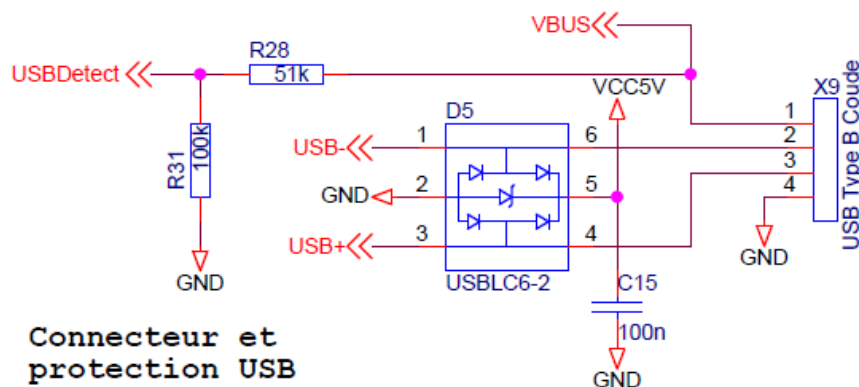
## 9. GESTION DU BUS USB

Ce chapitre a pour objectif de découvrir la gestion du bus USB fourni par Harmony et le MHC.

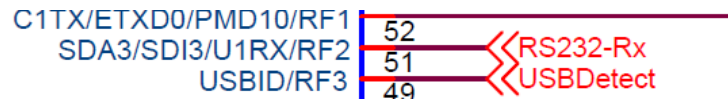
L'objectif est de réaliser un USB device de type cdc (Communication Device Class) qui permettra finalement de gérer l'USB comme un port de communication virtuel.

### 9.1. HARDWARE USB SUR LE KIT PIC32MX95F512L

Voici la circuiterie USB du kit PIC32MX795F512L :



L'entrée USBDetect est une E/S digitale qui correspond à RF3 broche 51.



## 9.2. UTILISATION D'UN EXEMPLE

Bien qu'il soit possible avec le MHC d'utiliser la USB Library et que l'on obtienne les drivers USB, il reste encore passablement de travail pour obtenir une application complète.

Pour découvrir l'USB, nous allons utiliser l'exemple qui se trouve sous :  
<Répertoire Harmony>\v<n>\apps\usb\device\cdc\_com\_port\_single.

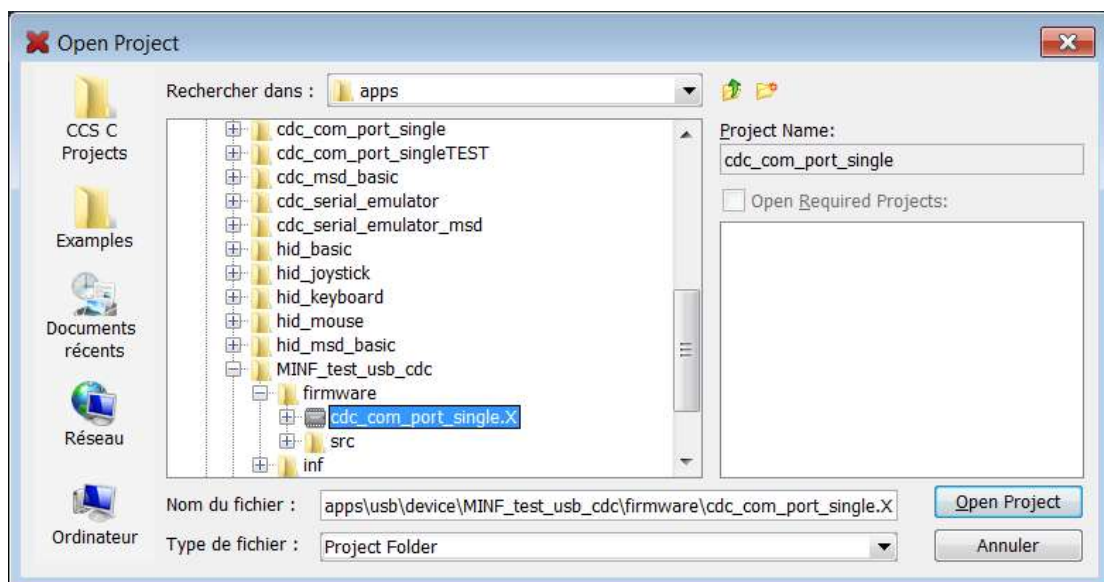
Pour ne pas modifier le contenu de l'exemple, nous copions le répertoire cdc\_com\_port\_single au même niveau et nous renommons la copie en MINF\_test\_usb\_cdc.

Le portage sur le kit SKES du projet suivant a été réalisé avec les logiciels suivants :

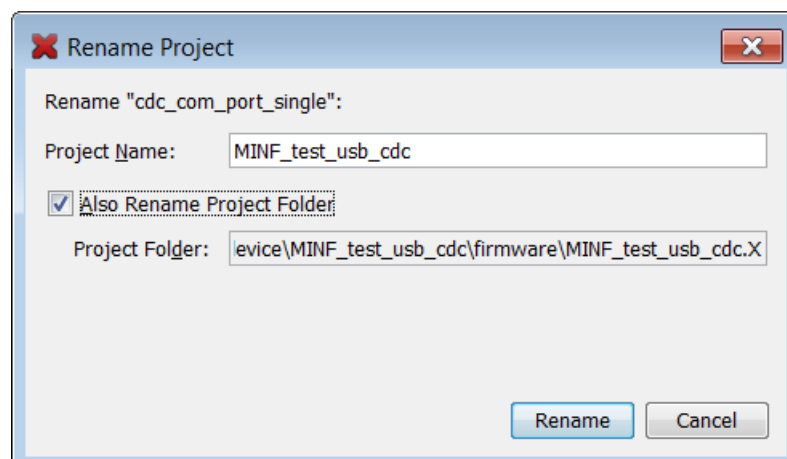
- Harmony v1\_06
- MPLABX IDE v3.10
- XC32 v1.40

### 9.2.1. OUVERTURE DU PROJET ET ADAPTATION

#### 9.2.1.1. OUVERTURE DU PROJET



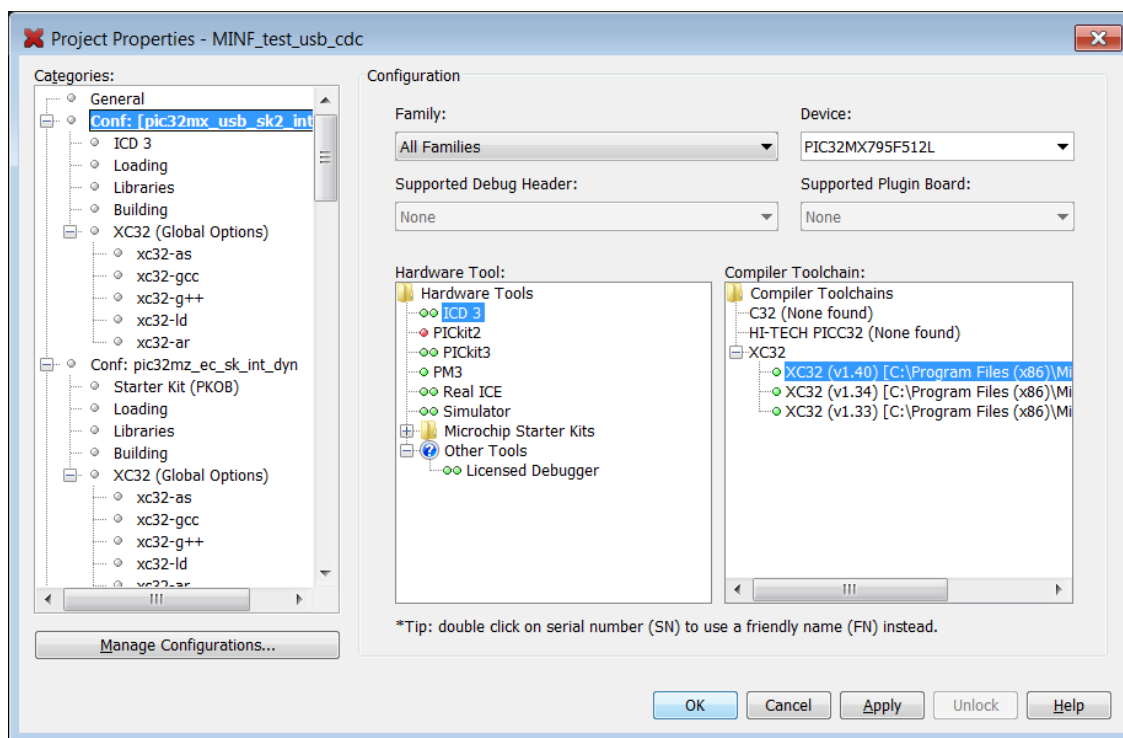
#### 9.2.1.2. RENOMMAGE DU PROJET



### 9.2.1.3. CHOIX DE LA CONFIGURATION

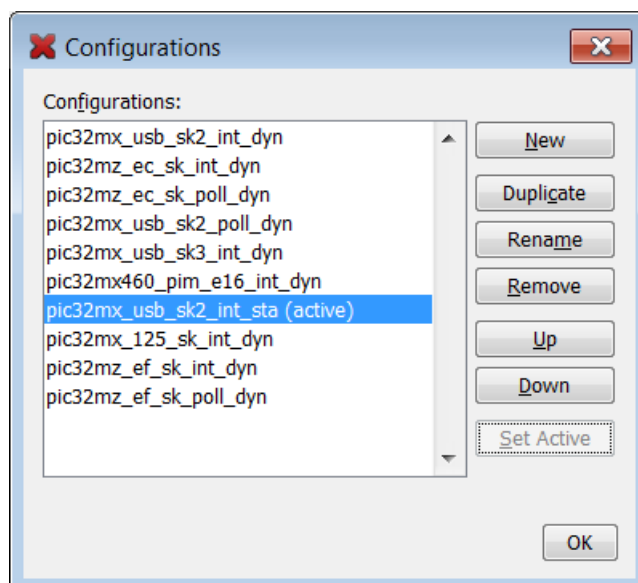
La particularité du projet fourni est qu'il s'agit d'un projet multi configuration et multi bsp.

Dans les propriétés du projet on découvre les multiples configurations.



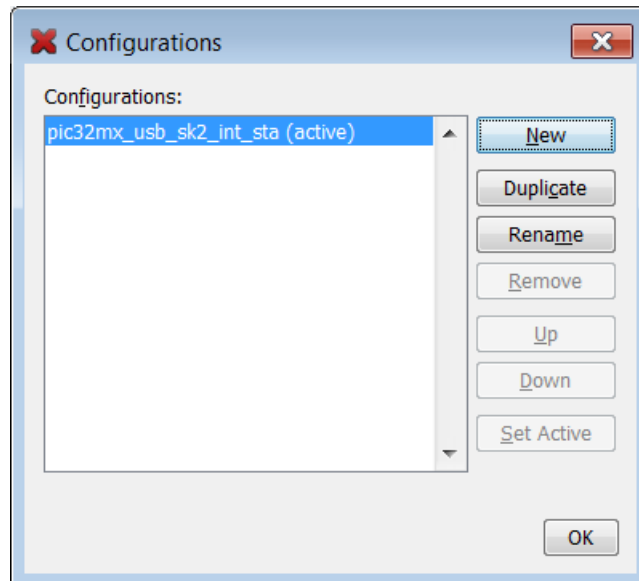
Avec  on peut choisir la configuration active.

Choix de **pic32mx\_usb\_sk2\_int\_sta** en utilisant le bouton "Set Active" :

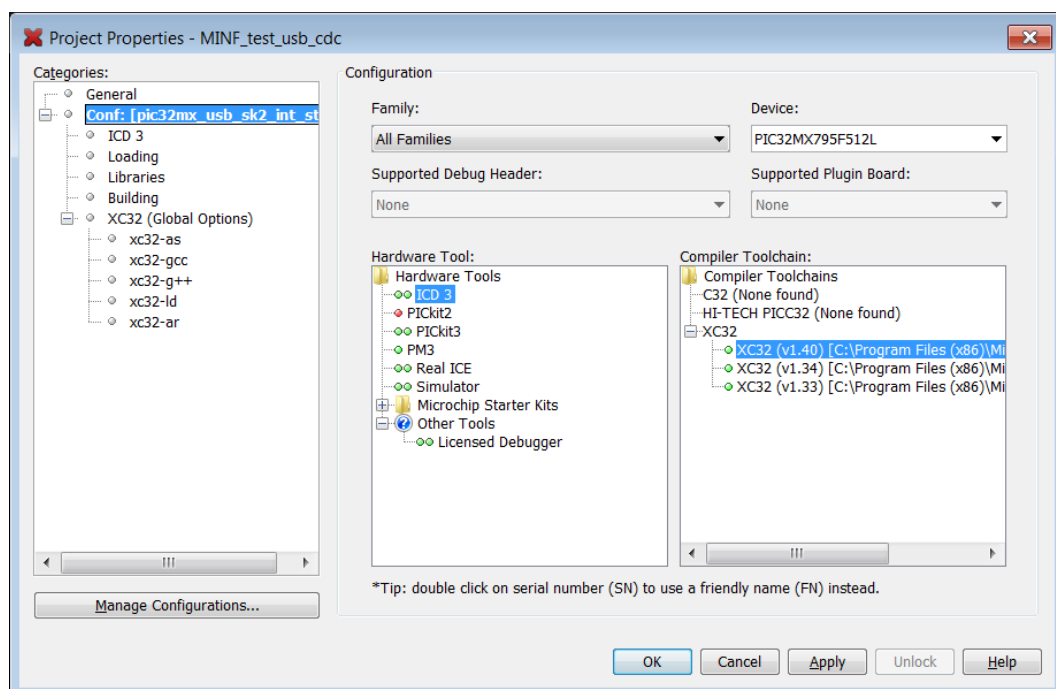


### 9.2.2. REDUCTION A UNE SEULE CONFIGURATION

En utilisant le bouton Remove, on enlève toutes les autres configurations.

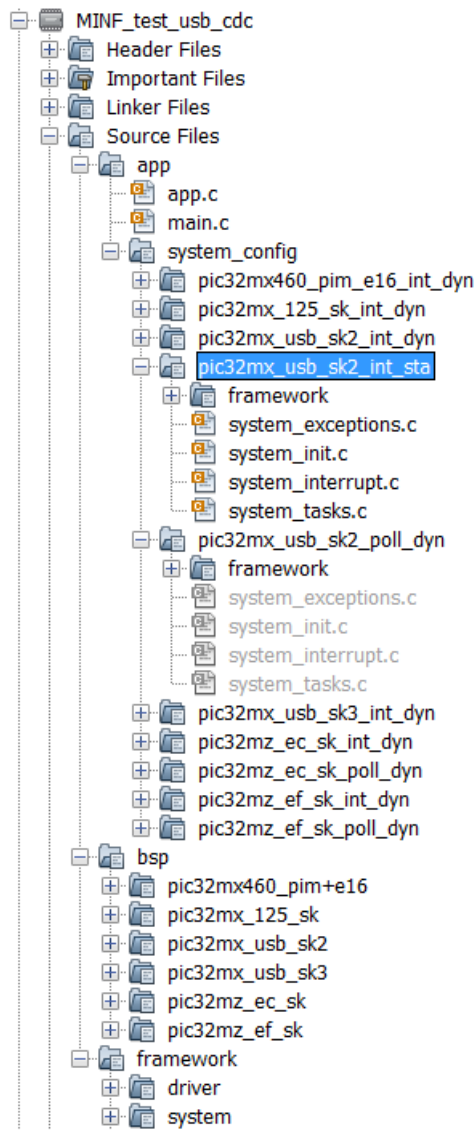


Ce qui nous ramène à un projet à configuration unique :





Par contre, cela ne supprime pas les répertoires. Pour les autres configurations les fichiers sont en grisé.



### 9.2.3. SELECTION DU BSP DU KIT

Afin de pouvoir sélectionner le bsp du kit, il est nécessaire d'ouvrir le MHC :



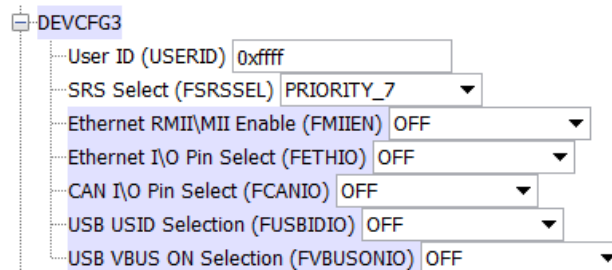
On coche le BSP du Kit ETML-ES.

### 9.2.4. ADAPTATION DU DEVICE CONFIGURATION POUR LE KIT

Les adaptations sont très modestes, le kit usb sk2 de Microchip utilise le même PIC32MX.

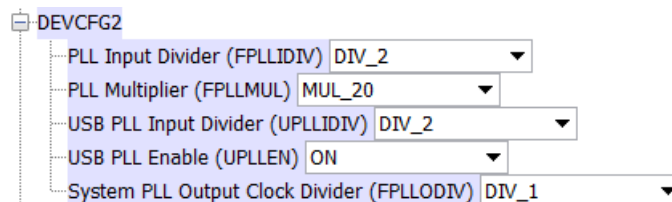
#### 9.2.4.1. SECTION DEVCFG3

Pas de modification nécessaire.



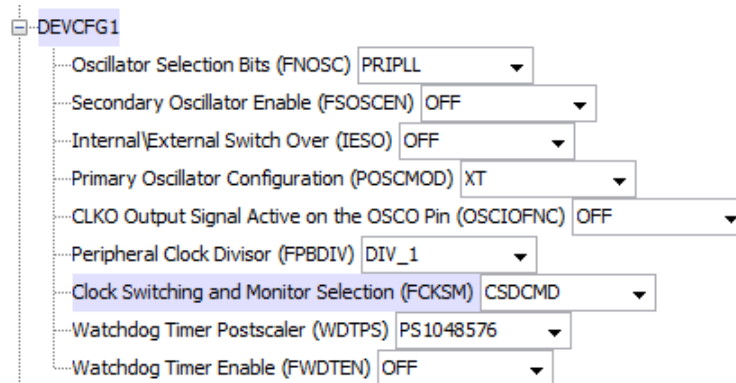
#### 9.2.4.2. SECTION DEVCFG2

Pas de modification nécessaire.



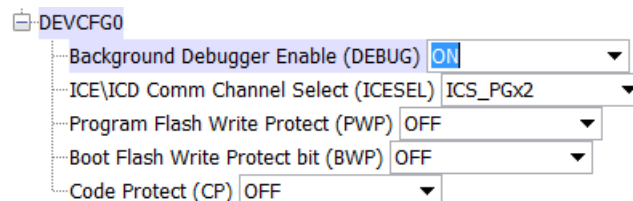
#### 9.2.4.3. SECTION DEVCFG1

Pas de modification nécessaire.



#### 9.2.4.4. SECTION DEVCFG0

On met le Debugger sur ON.

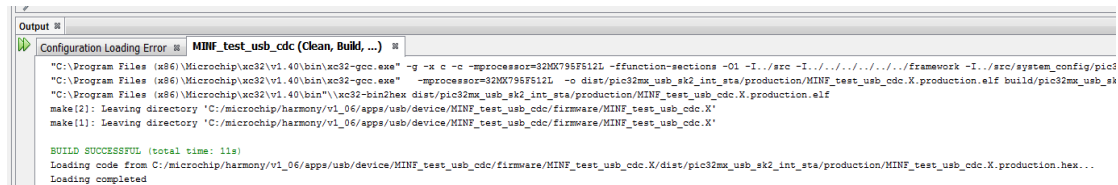


### 9.2.5. GENERATION DU CODE

Lors de la génération du code, il faut ajouter les éléments qui apparaissent dans la fenêtre de gauche. Par contre, ne pas supprimer les éléments en plus dans la partie de droite.

### 9.2.6. TEST DE COMPILATION

Le résultat du clean and build est OK.



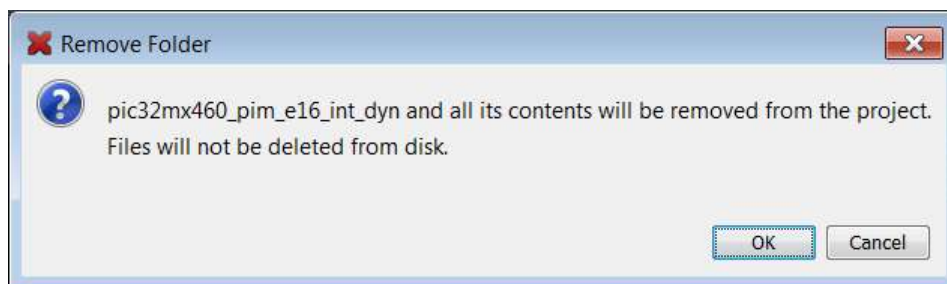
```
Output
Configuration Loading Error: MINF_test_usb_cdc (Clean, Build, ...)
"C:\Program Files (x86)\Microchip\pic32\v1.40\bin\pic32-gcc.exe" -g -x c -c -mprocessor=32MX795F512L -ffunction-sections -O1 -I. -I../src -I../framework -I../src/system_config/pic32
"C:\Program Files (x86)\Microchip\pic32\v1.40\bin\pic32-gcc.exe" -mprocessor=32MX795F512L -o dist/pic32mx_usb_sk2_int_sta/production/MINF_test_usb_cdc.X.production.elf build/pic32mx_usb_sk2
"C:\Program Files (x86)\Microchip\pic32\v1.40\bin\pic32-gcc.exe" -mprocessor=32MX795F512L -o dist/pic32mx_usb_sk2_int_sta/production/MINF_test_usb_cdc.X.production.elf build/pic32mx_usb_sk2
make[2]: Leaving directory 'C:/microchip/harmony/v1_06/apps/usb/device/MINF_test_usb_cdc/firmware/MINF_test_usb_cdc.X'
make[1]: Leaving directory 'C:/microchip/harmony/v1_06/apps/usb/device/MINF_test_usb_cdc/firmware/MINF_test_usb_cdc.X'

BUILD SUCCESSFUL (total time: 11s)
Loading code from C:/microchip/harmony/v1_06/apps/usb/device/MINF_test_usb_cdc/firmware/MINF_test_usb_cdc.X/dist/pic32mx_usb_sk2_int_sta/production/MINF_test_usb_cdc.X.production.hex...
Loading completed
```

### 9.2.7. NETTOYAGE DU PROJET

Pour éviter une arborescence encombrée, il est possible de faire de l'ordre en utilisant "Remove from project".

Pour un élément on obtient le message suivant :



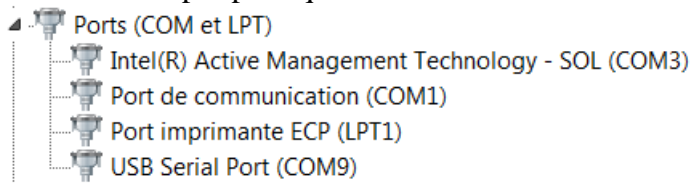
On procèdera de même pour les sections :

Header Files : system\_config et bsp

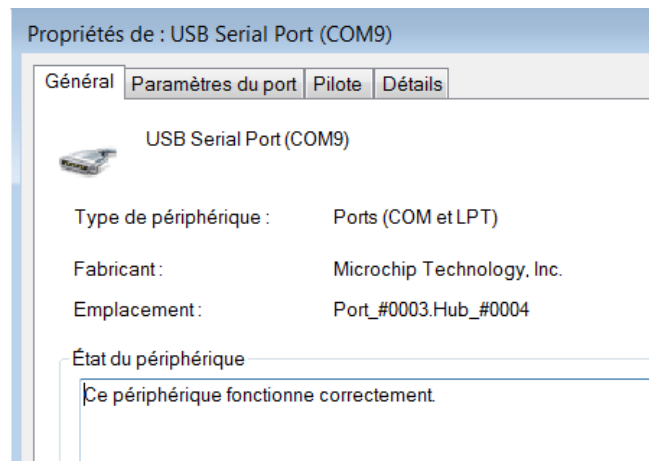
Sources Files : system\_config et bsp

### 9.2.8. TEST DE LA COMMUNICATION USB

En utilisant un port USB 2.0 sur le PC, on obtient l'installation d'un pilote et dans le gestionnaire de périphériques on trouve un USB Serial Port.

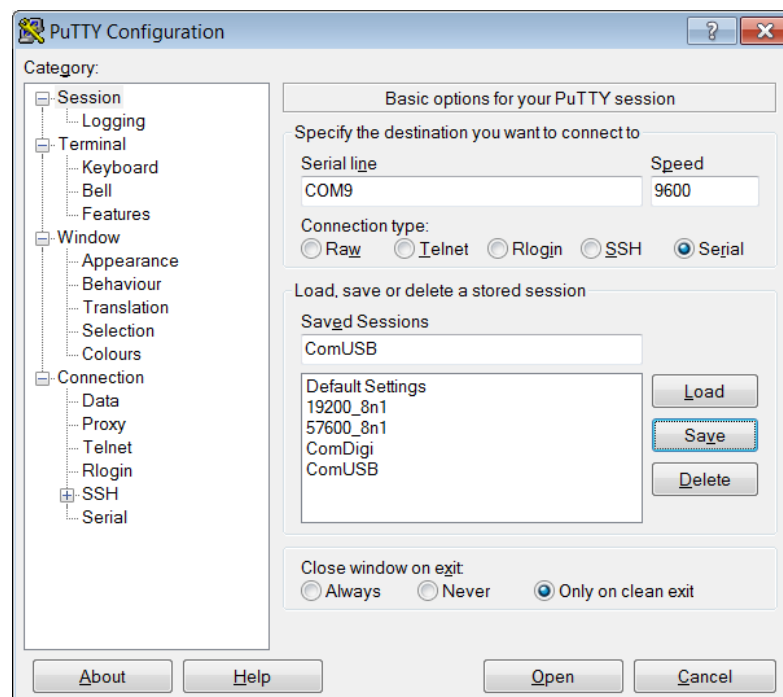


Et c'est bien un élément fourni par Microchip.

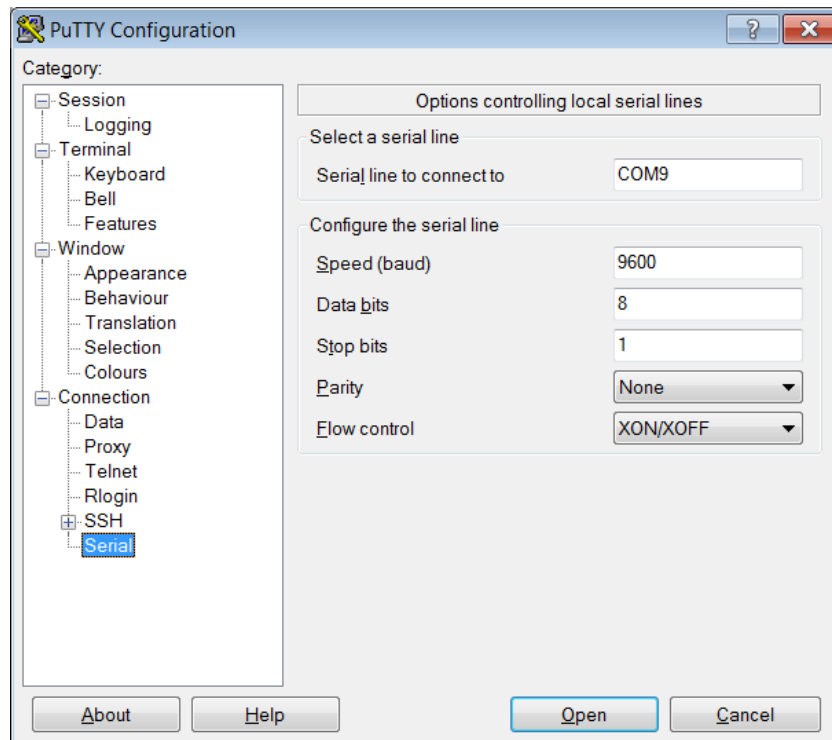


#### 9.2.8.1. TEST AVEC PUTTY

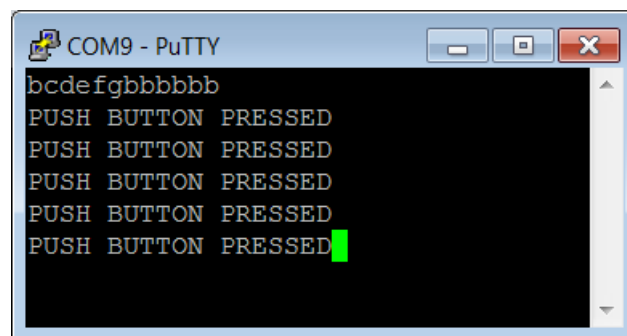
Dans cet exemple, le port série virtuel est le COM9 :



La configuration par défaut fonctionne !



Lorsque l'on presse sur le BSP\_SWITCH\_1 (touche OK du Kit) le message PUSH\_BUTTON\_PRESSED s'affiche. Lors d'une frappe au clavier on reçoit le caractère + 1, donc la touche 'a' affiche 'b'.



Cela montre que le système fonctionne.

### 9.2.9. DETAILS DE FONCTIONNEMENT

Les exemples de codes suivants sont tirés du projet testé avec les logiciels suivants :

- Harmony v1\_08
- MPLABX IDE v3.40
- XC32 v1.42

#### 9.2.9.1. CODE

L'USB a une machine d'état dédiée, comme une application standard. En voici le code :

```
void APP_Tasks (void )
{
    /* Update the application state machine based
     * on the current state */

    switch(appData.state)
    {
        case APP_STATE_INIT:

            /* Open the device layer */
            appData.deviceHandle = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                DRV_IO_INTENT_READWRITE );

            if(appData.deviceHandle != USB_DEVICE_HANDLE_INVALID)
            {
                /* Register a callback with device layer to get event
                 notification (for end point 0) */
                USB_DEVICE_EventHandlerSet(appData.deviceHandle,
                    APP_USBDeviceEventHandler, 0);

                appData.state = APP_STATE_WAIT_FOR_CONFIGURATION;
            }
            else
            {
                /* The Device Layer is not ready to be opened. We should try
                 * again later. */
            }
            break;

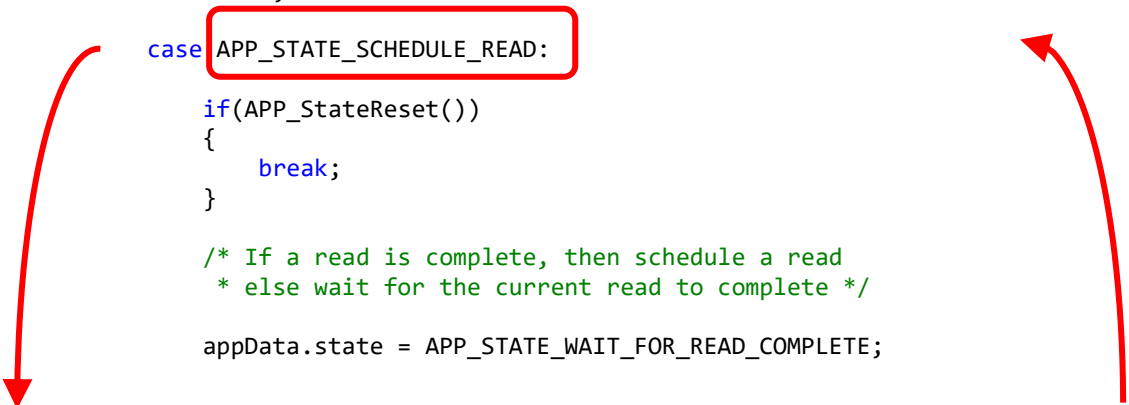
        case APP_STATE_WAIT_FOR_CONFIGURATION:

            /* Check if the device was configured */
            if(appData.isConfigured)
            {
                /* If the device is configured then lets start reading */
                appData.state = APP_STATE_SCHEDULE_READ;
            }
            break;

        case APP_STATE_SCHEDULE_READ:
            if(APP_StateReset())
            {
                break;
            }

            /* If a read is complete, then schedule a read
             * else wait for the current read to complete */

            appData.state = APP_STATE_WAIT_FOR_READ_COMPLETE;
```



```
if(appData.isReadComplete == true)
{
    appData.isReadComplete = false;
    appData.readTransferHandle =
        USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID;

    USB_DEVICE_CDC_Read (USB_DEVICE_CDC_INDEX_0,
        &appData.readTransferHandle, appData.readBuffer,
        APP_READ_BUFFER_SIZE);

    if(appData.readTransferHandle ==
        USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID)
    {
        appData.state = APP_STATE_ERROR;
        break;
    }
}
break;

case APP_STATE_WAIT_FOR_READ_COMPLETE:
case APP_STATE_CHECK_SWITCH_PRESSED:

    if(APP_StateReset())
    {
        break;
    }

    APP_ProcessSwitchPress();

    /* Check if a character was received or a switch was pressed.
     * The isReadComplete flag gets updated in the CDC event handler. */

    if(appData.isReadComplete || appData.isSwitchPressed)
    {
        appData.state = APP_STATE_SCHEDULE_WRITE;
    }
    break;

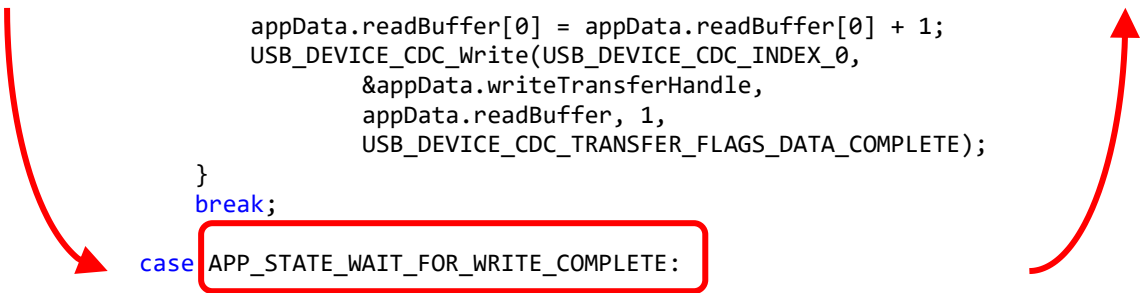
case APP_STATE_SCHEDULE_WRITE:

    if(APP_StateReset())
    {
        break;
    }

    /* Setup the write */

    appData.writeTransferHandle = USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID;
    appData.isWriteComplete = false;
    appData.state = APP_STATE_WAIT_FOR_WRITE_COMPLETE;

    if(appData.isSwitchPressed)
    {
        /* If the switch was pressed, then send the switch prompt*/
        appData.isSwitchPressed = false;
        USB_DEVICE_CDC_Write(USB_DEVICE_CDC_INDEX_0,
            &appData.writeTransferHandle, switchPromptUSB, 23,
            USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);
    }
    else
    {
        /* Else echo the received character + 1*/
    }
}
```



```
        appData.readBuffer[0] = appData.readBuffer[0] + 1;
        USB_DEVICE_CDC_Write(USB_DEVICE_CDC_INDEX_0,
                              &appData.writeTransferHandle,
                              appData.readBuffer, 1,
                              USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);
    }
    break;
case APP_STATE_WAIT_FOR_WRITE_COMPLETE:
    if(APP_StateReset())
    {
        break;
    }

    /* Check if a character was sent. The isWriteComplete
     * flag gets updated in the CDC event handler */

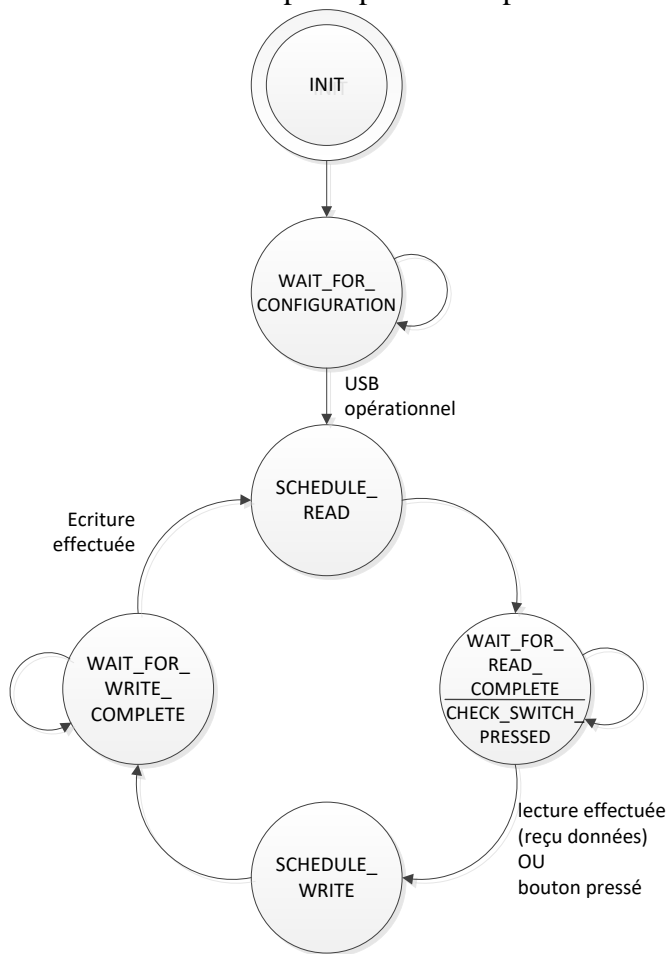
    if(appData.isWriteComplete == true)
    {
        appData.state = APP_STATE_SCHEDULE_READ;
    }
    break;

case APP_STATE_ERROR:
    break;
default:
    break;
    }
}
```



### 9.2.9.2. MACHINE D'ETAT

Son fonctionnement simplifié peut être représenté ainsi :



Par soucis de simplification, les transitions de retour depuis chaque état de lecture ou écriture vers l'état WAIT\_FOR\_CONFIGURATION n'ont pas été représentées. Ces transitions ont lieu en cas de réinitialisation de l'USB.

Les noms des états étant judicieusement choisis, on peut comprendre que l'application passe son temps à faire des lectures, puis dès qu'une donnée est reçue ou que le bouton a été appuyé, elle procède à une écriture. Et la boucle recommence. Cela correspond effectivement au comportement observé précédemment.

Partant de cet exemple, un fonctionnement de réception d'une trame, puis de renvoi d'une réponse peut facilement être implémenté. A la manière du bouton pressé, on peut également transmettre spontanément une trame (sans besoin de réception préalable).

### 9.2.9.3. DEROULEMENT D'UNE LECTURE

Pour effectuer une réception de données via USB, la fonction suivante est utilisée :

```
USB_DEVICE_CDC_Read (USB_DEVICE_CDC_INDEX_0,  
                      &appData.readTransferHandle,  
                      appData.readBuffer,  
                      APP_READ_BUFFER_SIZE) ;
```

Paramètres :

- Les deux premiers paramètres sont à conserver tels quels.
- Le paramètre n°3 est un pointeur sur un buffer où peuvent être stockées les données reçues.
- Le paramètre n°4 représente le nombre d'octets maximum à lire.

Cette fonction fait une requête de lecture au driver USB. Elle n'effectue pas la lecture ! Dans le fonctionnement USB, le master est le PC, le périphérique ne peut pas initier des communications.

On doit ensuite contrôler si la lecture a été effectuée (et donc on pourrait lire le contenu du buffer) ainsi :

```
if(appData.isReadComplete == true)
```

### 9.2.9.4. DEROULEMENT D'UNE ECRITURE

De la même manière, pour effectuer une requête d'écriture de données :

```
USB_DEVICE_CDC_Write(USB_DEVICE_CDC_INDEX_0,  
                     &appData.writeTransferHandle,  
                     switchPromptUSB,  
                     23,  
                     USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_  
                     COMPLETE) ;
```

Paramètres :

- Les deux premiers paramètres sont à conserver tels quels.
- Le paramètre n°3 est un pointeur sur un buffer où sont stockées les données à envoyer.
- Le paramètre n°4 représente le nombre d'octets à envoyer.
- Le paramètre n°5 indique qu'on ne désire pas compléter l'envoi par des données supplémentaires, et donc que l'envoi des données peut être effectué.

Comme dans le cas de la lecture, cette fonction fait uniquement une requête d'écriture au driver USB. Elle n'effectue pas l'écriture !

On doit ensuite contrôler si l'écriture a été effectuée :

```
if(appData.isWriteComplete == true)
```

La documentation concernant ces fonctions de lecture et écriture est tirée du l'aide Harmony disponible sous :

<Répertoire Harmony>\v<n>\doc\help\_harmony.chm

A la rubrique

MPLAB Harmony Framework Reference > USB Libraries Help > USB Device Library  
> USB CDC Device Library > Library Interface > a) Functions

#### 9.2.9.5. ETAT DE L'USB

L'application principale devra très certainement pouvoir connaître l'état de l'USB. Cet état peut être :

- 1) Reset (déconnecté),
- 2) Configured (en fonction).
- 3) Suspended (périphérique en mode basse consommation),

L'état "configuré" est le seul où des transferts peuvent avoir lieu.

En allant voir le code du gestionnaire d'événements USB (USBDeviceEventHandler() - dans le même fichier), on trouve 3 endroits :

```
/* *****  
 * Application USB Device Layer Event Handler.  
 * ***** */  
void APP_USBDeviceEventHandler ( USB_DEVICE_EVENT event, void * eventData,  
uintptr_t context )  
{  
    USB_DEVICE_EVENT_DATA_CONFIGURED *configuredEventData;  
  
    switch ( event )  
    {  
        case USB_DEVICE_EVENT_SOF:  
  
            /* This event is used for switch debounce. This flag is reset  
             * by the switch process routine. */  
            appData.sofEventHasOccurred = true;  
            break;  
  
        case USB_DEVICE_EVENT_RESET:  
  
            /* Update LED to show reset state */  
            BSP_LEDOn ( APP_USB_LED_1 );  
            BSP_LEDOn ( APP_USB_LED_2 );  
            BSP_LEDOff ( APP_USB_LED_3 );  
  
            appData.isConfigured = false;  
  
            break;  
  
        case USB_DEVICE_EVENT_CONFIGURED:  
  
            /* Check the configuratio. We only support configuration 1 */  
            configuredEventData = (USB_DEVICE_EVENT_DATA_CONFIGURED*)eventData;  
            if ( configuredEventData->configurationValue == 1 )  
            {  
                /* Update LED to show configured state */  
                BSP_LEDOff ( APP_USB_LED_1 );  
                BSP_LEDOff ( APP_USB_LED_2 );  
                BSP_LEDOn ( APP_USB_LED_3 );  
            }  
        }  
    }
```

1

2

```

        /* Register the CDC Device application event handler here.
        * Note how the appData object pointer is passed as the
        * user data */

        USB_DEVICE_CDC_EventHandlerSet(USB_DEVICE_CDC_INDEX_0,
        APP_USBDeviceCDCEventHandler, (uintptr_t)&appData);

        /* Mark that the device is now configured */
        appData.isConfigured = true;

    }
    break;

case USB_DEVICE_EVENT_POWER_DETECTED:

    /* VBUS was detected. We can attach the device */
    USB_DEVICE_Attach(appData.deviceHandle);
    break;

case USB_DEVICE_EVENT_POWER_REMOVED:

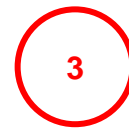
    /* VBUS is not available any more. Detach the device. */
    USB_DEVICE_Detach(appData.deviceHandle);
    break;

case USB_DEVICE_EVENT_SUSPENDED:

    /* Switch LED to show suspended state */
    BSP_LEDOff ( APP_USB_LED_1 );
    BSP_LEDOn ( APP_USB_LED_2 );
    BSP_LEDOn ( APP_USB_LED_3 );
    break;

case USB_DEVICE_EVENT_RESUMED:
case USB_DEVICE_EVENT_ERROR:
default:
    break;
}
}

```



De ces portions de code (ne pas oublier l'initialisation également), on peut facilement déduire l'état de l'USB et renseigner notre application principale.

### 9.3. AJOUT DE TRAITEMENT PARTICULIER

Dans la perspective d'intégrer une application existante à la communication USB, il est nécessaire d'introduire la possibilité de déclencher cycliquement une action d'application et d'introduire des interruptions supplémentaires.

La machine d'état de l'application issue de l'exemple USB a des états spécifiques au fonctionnement USB. Pour permettre le traitement habituel avec les 3 états (INIT, WAIT, SERVICE\_TASKS), la solution consiste à ajouter une deuxième application distincte avec sa machine d'état. Il faudra prévoir des variables ou fonctions pour se passer les informations entre les 2 machines d'état.

Ceci va permettre, en relation avec `system_interrupt.c`, d'activer cycliquement notre application et d'y ajouter les traitements spécifiques.

## 9.4. EXEMPLE D'APPLICATION

Il s'agit maintenant d'étudier l'application et de déterminer comment récupérer les données reçues par USB et en envoyer.

Nous allons réutiliser une application prévue pour le test de l'UART. Cette application envoie des messages de type texte, formatés de la manière suivante :

```
strMess = "!Ch0 " & NudCanal0.Value & " Ch1 " & NudCanal1.Value & "#"
```

### 9.4.1. INITIALISATION

Au début du case APP\_STATE\_INIT: nous ajoutons :

```
// Init pour Kit pic32mx_skes
lcd_init();
lcd_bl_on();

// Init AD
BSP_InitADC10();

printf_lcd("MINF_test USB CDC");
lcd_gotoxy(1,2);
printf_lcd("CHR 21.04.2016    ");
```

Ceci avant :

```
/* Open the device layer */
```

### 9.4.2. AFFICHAGE DES MESSAGES REÇUS

Dans un 1<sup>er</sup> temps, nous allons afficher le message reçu sur la 4<sup>ème</sup> ligne de l'afficheur LCD.

C'est dans le case APP\_STATE\_WAIT\_FOR\_READ\_COMPLETE que nous pouvons récupérer le message entrant.

```
case APP_STATE_WAIT_FOR_READ_COMPLETE:
case APP_STATE_CHECK_SWITCH_PRESSED:
    if (APP_StateReset())
    {
        break;
    }

    APP_ProcessSwitchPress();

    /* Check if a character was received or a switch
       was pressed. The isReadComplete flag gets updated in
       the CDC event handler. */

    if (appData.isReadComplete || appData.isSwitchPressed)
    {
        appData.state = APP_STATE_SCHEDULE_WRITE;
        lcd_gotoxy(1,4);
        printf_lcd("%s", appData.readBuffer );
    }
    break;
```

☹ Il n'y a pas d'élément pour connaître le nombre d'octets reçus. En principe la trame USB est de 64 octets au maximum.

### 9.4.3. ENVOI DES MESSAGES

Création d'un message contenant la valeur des 2 canaux de l'AD.

C'est dans le case APP\_STATE\_SCHEDULE\_WRITE que nous plaçons l'émission du message.

#### 9.4.3.1. CONTENU INITIAL APP\_STATE\_SCHEDULE\_WRITE

Voici le contenu non modifié du case APP\_STATE\_SCHEDULE\_WRITE:

```
case APP_STATE_SCHEDULE_WRITE:
    if (APP_StateReset())
    {
        break;
    }

    /* Setup the write */
    appData.writeTransferHandle =
        USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID;
    appData.isWriteComplete = false;
    appData.state = APP_STATE_WAIT_FOR_WRITE_COMPLETE;

    if (appData.isSwitchPressed)
    {
        /* If the switch was pressed,
           then send the switch prompt*/
        appData.isSwitchPressed = false;
        USB_DEVICE_CDC_Write(USB_DEVICE_CDC_INDEX_0,
                             &appData.writeTransferHandle,
                             switchPromptUSB, 23,
                             USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);
    }
    else
    {
        /* Else echo the received character + 1 */
        appData.readBuffer[0] = appData.readBuffer[0] + 1;
        USB_DEVICE_CDC_Write(USB_DEVICE_CDC_INDEX_0,
                             &appData.writeTransferHandle,
                             appData.readBuffer, 1,
                             USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);
    }
    break;
```

#### 9.4.3.2. CONTENU MODIFIE DE APP\_STATE\_SCHEDULE\_WRITE

Voici le contenu modifié du case **APP\_STATE\_SCHEDULE\_WRITE**. Suppression de la gestion du switch et traitement pour envoyer un message lorsqu'on en reçoit un.

Ajout des éléments suivants au niveau des données de l'application.

```
APP_DATA appData;
```

```
S_ADCResults ValAD;
```

```
char SendBuffer[64];
```

```
case APP_STATE_SCHEDULE_WRITE:
```

```
    if (APP_StateReset())
```

```
    {
```

```
        break;
```

```
    }
```

```
    /* Setup the write */
```

```
    appData.writeTransferHandle =
```

```
        USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID;
```

```
    appData.isWriteComplete = false;
```

```
    appData.state = APP_STATE_WAIT_FOR_WRITE_COMPLETE;
```

```
    // Lecture AD et composition du message
```

```
    ValAD = BSP_ReadAllADC();
```

```
    sprintf(SendBuffer, "!Ch0 %04d Ch1 %04d#",
```

```
        ValAD.Chan0, ValAD.Chan1);
```

```
    // Emission du message
```

```
    USB_DEVICE_CDC_Write(USB_DEVICE_CDC_INDEX_0,
```

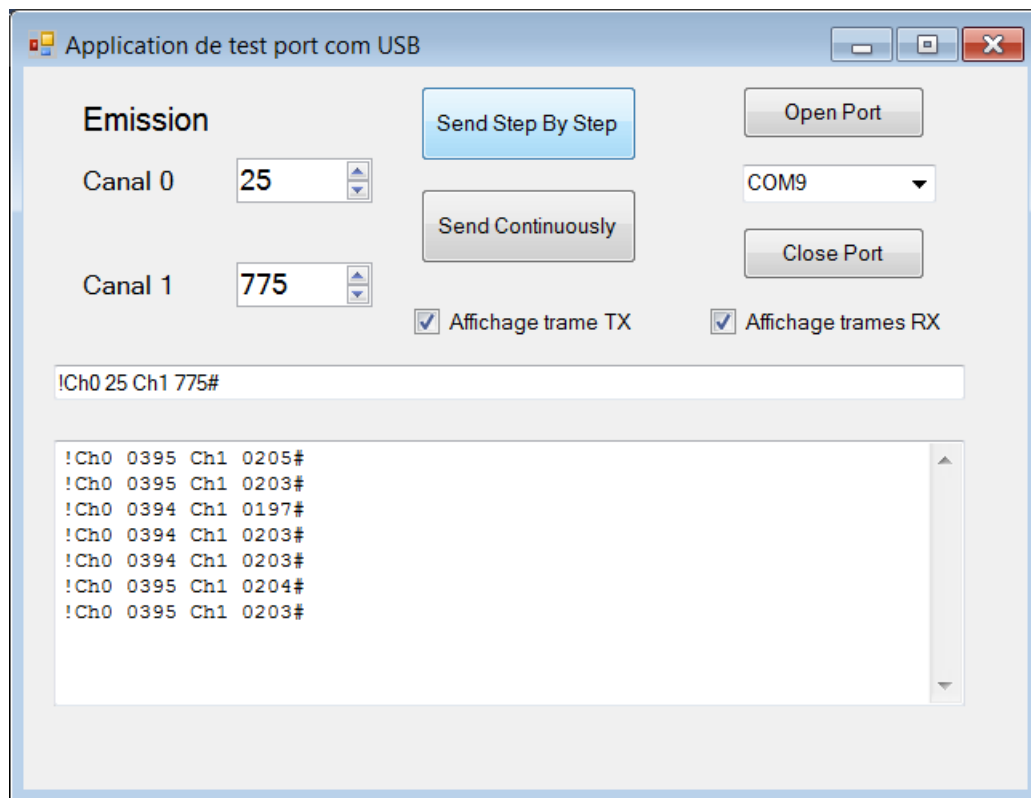
```
        &appData.writeTransferHandle,
```

```
        SendBuffer, strlen(SendBuffer),
```

```
        USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);
```

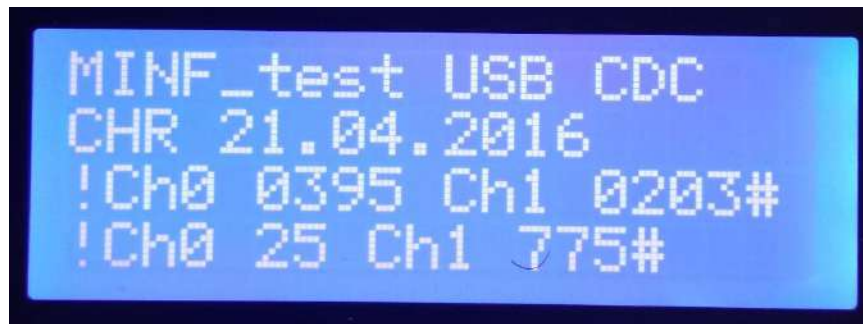
```
break;
```

#### 9.4.4. SITUATION AVEC APPLICATION WINDOWS



On obtient l'affichage des messages émis par le kit, et sur le kit on obtient l'affichage du message émis par l'application Windows.

Ligne 3 = valeurs envoyées / ligne 4 = valeurs reçues.





## 9.5. CONCLUSION

Ce document devrait permettre, en s'inspirant de la démarche d'adaptation d'un projet USB Microchip, de procéder de même pour d'autres applications et projets particuliers. Dans le cas d'une application finale plus complexe, il peut être très utile de séparer le fonctionnement en plusieurs "app", comme le MHC le permet, chacune ayant sa machine d'état et gérant sa partie.

## 9.6. HISTORIQUE DES VERSIONS

### 9.6.1. VERSION 1.5 AVRIL 2015

Création du document. Version 1.5 pour indiquer l'utilisation de Harmony.

### 9.6.2. VERSION 1.6 AVRIL 2016

Reprise du document avec la situation Harmony 1.06 et MPLABX 3.10. Le fait d'avoir le bsp sélectionnable dans la liste modifie passablement le contenu du document.

### 9.6.3. VERSION 1.7 MARS 2017

Reprise du document par SCA. Relecture. Compléments fonctionnement et reprise projet "cdc\_com\_port\_single".

### 9.6.1. VERSION 1.71 MARS 2019

Compléments mineurs.