**Instituto Tecnológico y de Estudios Superiores de Monterrey**

**Interfaces computacionales**

**Práctica #4: Implementación de una interfaz serial.**

**Mario Ortega A01730557**

**Stephanie Benitez A00322030**

**Héctor Pequeño A01246364**

**Objetivo:** Utilizar un protocolo de comunicación serial entre un programa de cómputo y un dispositivo periférico con alguna aplicación específica.

1. **Explicación de funcionamiento:**

Una vez cargado al PIC32 el carácter ya sea del abecedario o numérico que ingreses en el PUTTY el microcontrolador lo procesa le suma uno (+1) a su valor en ASCII y lo muestra en la terminal PUTTY con su valor sumado.

1. **Código modificado:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MPLAB Harmony Application Source File

Company:

Microchip Technology Inc.

File Name:

app.c

Summary:

This file contains the source code for the MPLAB Harmony application.

Description:

This file contains the source code for the MPLAB Harmony application. It

implements the logic of the application's state machine and it may call

API routines of other MPLAB Harmony modules in the system, such as drivers,

system services, and middleware. However, it does not call any of the

system interfaces (such as the "Initialize" and "Tasks" functions) of any of

the modules in the system or make any assumptions about when those functions

are called. That is the responsibility of the configuration-specific system

files.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// DOM-IGNORE-BEGIN

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Copyright (c) 2013-2014 released Microchip Technology Inc. All rights reserved.

Microchip licenses to you the right to use, modify, copy and distribute

Software only when embedded on a Microchip microcontroller or digital signal

controller that is integrated into your product or third party product

(pursuant to the sublicense terms in the accompanying license agreement).

You should refer to the license agreement accompanying this Software for

additional information regarding your rights and obligations.

SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,

EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF

MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER

CONTRACT, NEGLIGENCE, STRICT LIABILITY, CONTRIBUTION, BREACH OF WARRANTY, OR

OTHER LEGAL EQUITABLE THEORY ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES

INCLUDING BUT NOT LIMITED TO ANY INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR

CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF

SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES

(INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// DOM-IGNORE-END

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Section: Included Files

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include "app.h"

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Section: Global Data Definitions

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

const uint8\_t \_attribute\_((aligned(16))) switchPromptUSB[] = "\r\nPUSH BUTTON PRESSED";

uint8\_t APP\_MAKE\_BUFFER\_DMA\_READY readBuffer[APP\_READ\_BUFFER\_SIZE];

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\* Application Data

Summary:

Holds application data

Description:

This structure holds the application's data.

Remarks:

This structure should be initialized by the APP\_Initialize function.

Application strings and buffers are be defined outside this structure.

\*/

APP\_DATA appData;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Section: Application Callback Functions

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* USB CDC Device Events - Application Event Handler

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

USB\_DEVICE\_CDC\_EVENT\_RESPONSE APP\_USBDeviceCDCEventHandler

(

USB\_DEVICE\_CDC\_INDEX index ,

USB\_DEVICE\_CDC\_EVENT event ,

void \* pData,

uintptr\_t userData

)

{

APP\_DATA \* appDataObject;

appDataObject = (APP\_DATA \*)userData;

USB\_CDC\_CONTROL\_LINE\_STATE \* controlLineStateData;

USB\_DEVICE\_CDC\_EVENT\_DATA\_READ\_COMPLETE \* eventDataRead;

switch ( event )

{

case USB\_DEVICE\_CDC\_EVENT\_GET\_LINE\_CODING:

/\* This means the host wants to know the current line

\* coding. This is a control transfer request. Use the

\* USB\_DEVICE\_ControlSend() function to send the data to

\* host. \*/

USB\_DEVICE\_ControlSend(appDataObject->deviceHandle,

&appDataObject->getLineCodingData, sizeof(USB\_CDC\_LINE\_CODING));

break;

case USB\_DEVICE\_CDC\_EVENT\_SET\_LINE\_CODING:

/\* This means the host wants to set the line coding.

\* This is a control transfer request. Use the

\* USB\_DEVICE\_ControlReceive() function to receive the

\* data from the host \*/

USB\_DEVICE\_ControlReceive(appDataObject->deviceHandle,

&appDataObject->setLineCodingData, sizeof(USB\_CDC\_LINE\_CODING));

break;

case USB\_DEVICE\_CDC\_EVENT\_SET\_CONTROL\_LINE\_STATE:

/\* This means the host is setting the control line state.

\* Read the control line state. We will accept this request

\* for now. \*/

controlLineStateData = (USB\_CDC\_CONTROL\_LINE\_STATE \*)pData;

appDataObject->controlLineStateData.dtr = controlLineStateData->dtr;

appDataObject->controlLineStateData.carrier = controlLineStateData->carrier;

USB\_DEVICE\_ControlStatus(appDataObject->deviceHandle, USB\_DEVICE\_CONTROL\_STATUS\_OK);

break;

case USB\_DEVICE\_CDC\_EVENT\_SEND\_BREAK:

/\* This means that the host is requesting that a break of the

\* specified duration be sent. Read the break duration \*/

appDataObject->breakData = ((USB\_DEVICE\_CDC\_EVENT\_DATA\_SEND\_BREAK \*)pData)->breakDuration;

/\* Complete the control transfer by sending a ZLP \*/

USB\_DEVICE\_ControlStatus(appDataObject->deviceHandle, USB\_DEVICE\_CONTROL\_STATUS\_OK);

break;

case USB\_DEVICE\_CDC\_EVENT\_READ\_COMPLETE:

/\* This means that the host has sent some data\*/

eventDataRead = (USB\_DEVICE\_CDC\_EVENT\_DATA\_READ\_COMPLETE \*)pData;

appDataObject->isReadComplete = true;

appDataObject->numBytesRead = eventDataRead->length;

break;

case USB\_DEVICE\_CDC\_EVENT\_CONTROL\_TRANSFER\_DATA\_RECEIVED:

/\* The data stage of the last control transfer is

\* complete. For now we accept all the data \*/

USB\_DEVICE\_ControlStatus(appDataObject->deviceHandle, USB\_DEVICE\_CONTROL\_STATUS\_OK);

break;

case USB\_DEVICE\_CDC\_EVENT\_CONTROL\_TRANSFER\_DATA\_SENT:

/\* This means the GET LINE CODING function data is valid. We dont

\* do much with this data in this demo. \*/

break;

case USB\_DEVICE\_CDC\_EVENT\_WRITE\_COMPLETE:

/\* This means that the data write got completed. We can schedule

\* the next read. \*/

appDataObject->isWriteComplete = true;

break;

default:

break;

}

return USB\_DEVICE\_CDC\_EVENT\_RESPONSE\_NONE;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 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 );

/\* 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;

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Section: Application Local Functions

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void APP\_ProcessSwitchPress()

{

/\* This function checks if the switch is pressed and then

\* debounces the switch press\*/

if(BSP\_SWITCH\_STATE\_PRESSED == (BSP\_SwitchStateGet(APP\_USB\_SWITCH\_1)))

{

if(appData.ignoreSwitchPress)

{

/\* This measn the key press is in progress \*/

if(appData.sofEventHasOccurred)

{

/\* A timer event has occurred. Update the debounce timer \*/

appData.switchDebounceTimer ++;

appData.sofEventHasOccurred = false;

if (USB\_DEVICE\_ActiveSpeedGet(appData.deviceHandle) == USB\_SPEED\_FULL)

{

appData.debounceCount = APP\_USB\_SWITCH\_DEBOUNCE\_COUNT\_FS;

}

else if (USB\_DEVICE\_ActiveSpeedGet(appData.deviceHandle) == USB\_SPEED\_HIGH)

{

appData.debounceCount = APP\_USB\_SWITCH\_DEBOUNCE\_COUNT\_HS;

}

if(appData.switchDebounceTimer == appData.debounceCount)

{

/\* Indicate that we have valid switch press. The switch is

\* pressed flag will be cleared by the application tasks

\* routine. We should be ready for the next key press.\*/

appData.isSwitchPressed = true;

appData.switchDebounceTimer = 0;

appData.ignoreSwitchPress = false;

}

}

}

else

{

/\* We have a fresh key press \*/

appData.ignoreSwitchPress = true;

appData.switchDebounceTimer = 0;

}

}

else

{

/\* No key press. Reset all the indicators. \*/

appData.ignoreSwitchPress = false;

appData.switchDebounceTimer = 0;

appData.sofEventHasOccurred = false;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* This function is called in every step of the

\* application state machine.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool APP\_StateReset(void)

{

/\* This function returns true if the device

\* was reset \*/

bool retVal;

if(appData.isConfigured == false)

{

appData.state = APP\_STATE\_WAIT\_FOR\_CONFIGURATION;

appData.readTransferHandle = USB\_DEVICE\_CDC\_TRANSFER\_HANDLE\_INVALID;

appData.writeTransferHandle = USB\_DEVICE\_CDC\_TRANSFER\_HANDLE\_INVALID;

appData.isReadComplete = true;

appData.isWriteComplete = true;

retVal = true;

}

else

{

retVal = false;

}

return(retVal);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Section: Application Initialization and State Machine Functions

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function:

void APP\_Initialize ( void )

Remarks:

See prototype in app.h.

\*/

void APP\_Initialize ( void )

{

/\* Place the App state machine in its initial state. \*/

appData.state = APP\_STATE\_INIT;

/\* Device Layer Handle \*/

appData.deviceHandle = USB\_DEVICE\_HANDLE\_INVALID ;

/\* Device configured status \*/

appData.isConfigured = false;

/\* Initial get line coding state \*/

appData.getLineCodingData.dwDTERate = 9600;

appData.getLineCodingData.bParityType = 0;

appData.getLineCodingData.bParityType = 0;

appData.getLineCodingData.bDataBits = 8;

/\* Read Transfer Handle \*/

appData.readTransferHandle = USB\_DEVICE\_CDC\_TRANSFER\_HANDLE\_INVALID;

/\* Write Transfer Handle \*/

appData.writeTransferHandle = USB\_DEVICE\_CDC\_TRANSFER\_HANDLE\_INVALID;

/\* Intialize the read complete flag \*/

appData.isReadComplete = true;

/Initialize the write complete flag/

appData.isWriteComplete = true;

/\* Initialize Ignore switch flag \*/

appData.ignoreSwitchPress = false;

/\* Reset the switch debounce counter \*/

appData.switchDebounceTimer = 0;

/\* Reset other flags \*/

appData.sofEventHasOccurred = false;

appData.isSwitchPressed = false;

/\* Set up the read buffer \*/

appData.readBuffer = &readBuffer[0];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function:

void APP\_Tasks ( void )

Remarks:

See prototype in app.h.

\*/

uint8\_t printFlag = 0;

uint32\_t res = 0;

uint32\_t auxRes = 0;

int digitosCont=0;

int miStringCont=0;

uint8\_t APP\_MAKE\_BUFFER\_DMA\_READY miString[] = " ";

char auxString[] = " ";

char chr=0;

int j=0;

void miPrintf(char\* s, int cont) {

int i;

for (i=0;i<cont;i++)

miString[i]=s[i];

printFlag=1;

miStringCont=cont;

}

void APP\_Tasks (void )

{

/\* Update the application state machine based

\* on the current state \*/

int i;

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 each received character by adding 1 \*/

for(i=0; i<appData.numBytesRead; i++)

{

if((appData.readBuffer[i] != 0x0A) && (appData.readBuffer[i] != 0x0D))

{

//printFlag |= 1;

//miPrintf();

if((appData.readBuffer[i] >= '0') && (appData.readBuffer[i] <= '9'))

{

//appData.readBuffer[i] = appData.readBuffer[i] + 1;

if (digitosCont < 2){

chr=appData.readBuffer[i];

miPrintf(&chr,1);

res \*= 10;

res += (chr-'0');

digitosCont++;

}

//numReceived \*= 10;

//numReceived += (appData.readBuffer[i]-'0');

}

if((appData.readBuffer[i] == ' '))

{

//appData.readBuffer[i] = appData.readBuffer[i] + 1;

//chr=appData.readBuffer[i];

//miPrintf(&chr,1);

res +=1;

auxRes=res;

digitosCont=0;

do {

auxRes/=10;

digitosCont++;

} while(auxRes);

//if(digitosCont > 2) {

// digitosCont = 2;

//}

j=digitosCont+1;

auxString[j]=' ';

j--;

do {

auxString[j]='0'+(res%10);

res/=10;

} while(--j>=0);

auxString[0]=' ';

miPrintf(&auxString[0],digitosCont+1);

digitosCont=0;

res=0;

//numReceived \*= 10;

//numReceived += (appData.readBuffer[i]-'0');

}

}

}

if (printFlag) {

USB\_DEVICE\_CDC\_Write(USB\_DEVICE\_CDC\_INDEX\_0, &appData.writeTransferHandle,

miString, miStringCont, USB\_DEVICE\_CDC\_TRANSFER\_FLAGS\_DATA\_COMPLETE);

printFlag=0;

miStringCont=0;

}

}

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;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

End of File

\*/