/\*

\* main.c //

\*

\* Created on: Jun 11, 2018

\* Author:Nirojan N

\*/

#include "xparameters.h"

#include "xil\_printf.h"

#include "sleep.h"

#include "xiic.h"

#include "xintc.h"

#include "xil\_exception.h"

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include "PmodESP32.h"

#ifdef \_\_MICROBLAZE\_\_

#define HOST\_UART\_DEVICE\_ID XPAR\_AXI\_UARTLITE\_0\_BASEADDR

#define HostUart XUartLite

#define HostUart\_Config XUartLite\_Config

#define HostUart\_CfgInitialize XUartLite\_CfgInitialize

#define HostUart\_LookupConfig XUartLite\_LookupConfig

#define HostUart\_Recv XUartLite\_Recv

#define HostUartConfig\_GetBaseAddr(CfgPtr) (CfgPtr->RegBaseAddr)

#include "xuartlite.h"

#include "xil\_cache.h"

#else

#define HOST\_UART\_DEVICE\_ID XPAR\_PS7\_UART\_1\_DEVICE\_ID

#define HostUart XUartPs

#define HostUart\_Config XUartPs\_Config

#define HostUart\_CfgInitialize XUartPs\_CfgInitialize

#define HostUart\_LookupConfig XUartPs\_LookupConfig

#define HostUart\_Recv XUartPs\_Recv

#define HostUartConfig\_GetBaseAddr(CfgPtr) (CfgPtr->BaseAddress)

#include "xuartps.h"

#endif

//----------------------OLED---------------------------------

#include "bitmap.h"

#include "PmodOLEDrgb.h"

#include "sleep.h"

#include "xil\_cache.h"

#include "xparameters.h"

void Oled\_Initialize();

void Oled\_Run();

void DemoCleanup();

void EnableCaches();

void DisableCaches();

PmodOLEDrgb oledrgb;

u8 rgbUserFont[] = {

0x00, 0x04, 0x02, 0x1F, 0x02, 0x04, 0x00, 0x00, // 0x00

0x0E, 0x1F, 0x15, 0x1F, 0x17, 0x10, 0x1F, 0x0E, // 0x01

0x00, 0x1F, 0x11, 0x00, 0x00, 0x11, 0x1F, 0x00, // 0x02

0x00, 0x0A, 0x15, 0x11, 0x0A, 0x04, 0x00, 0x00, // 0x03

0x07, 0x0C, 0xFA, 0x2F, 0x2F, 0xFA, 0x0C, 0x07 // 0x04

}; // This table defines 5 user characters, although only one is used

//-------------------------------------------------------

#define PMODESP32\_UART\_BASEADDR XPAR\_PMODESP32\_0\_AXI\_LITE\_UART\_BASEADDR

#define PMODESP32\_GPIO\_BASEADDR XPAR\_PMODESP32\_0\_AXI\_LITE\_GPIO\_BASEADDR

void EnableCaches();

void DisableCaches();

void DemoInitialize();

void DemoRun();

void DemoCleanup();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constant Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* The following constants map to the XPAR parameters created in the

\* xparameters.h file. They are defined here such that a user can easily

\* change all the needed parameters in one place.

\*/

#define IIC\_DEVICE\_ID XPAR\_IIC\_0\_DEVICE\_ID

#define INTC\_DEVICE\_ID XPAR\_INTC\_0\_DEVICE\_ID

#define INTC\_IIC\_INTERRUPT\_ID XPAR\_INTC\_0\_IIC\_0\_VEC\_ID

/\*s

\* The following constant defines the address of the IIC

\* temperature sensor device on the IIC bus. Note that since

\* the address is only 7 bits, this constant is the address divided by 2.

\*/

#define TEMP\_SENSOR\_ADDRESS 0x4B /\* The actual address is 0x30 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Type Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Macros (Inline Functions) Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function Prototypes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int TempSensorExample(u16 IicDeviceId, u8 TempSensorAddress,

u8 \*TemperaturePtr);

static int SetupInterruptSystem(XIic \*IicPtr);

static void RecvHandler(void \*CallbackRef, int ByteCount);

static void StatusHandler(void \*CallbackRef, int Status);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Variable Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

XIic Iic; /\* The instance of the IIC device \*/

XIntc InterruptController; /\* The instance of the Interrupt controller \*/

/\*

\* The following structure contains fields that are used with the callbacks

\* (handlers) of the IIC driver. The driver asynchronously calls handlers

\* when abnormal events occur or when data has been sent or received. This

\* structure must be volatile to work when the code is optimized.

\*/

volatile struct {

int EventStatus;

int RemainingRecvBytes;

int EventStatusUpdated;

int RecvBytesUpdated;

} HandlerInfo;

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

/\*\*

\*

\* The purpose of this function is to illustrate how to use the IIC driver to

\* read the temperature.

\*

\*

\* @return XST\_SUCCESS if successful, XST\_FAILURE if unsuccessful

\*

\* @note None

\*

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

int temp\_init(){

int Status;

u8 TemperaturePtr;

/\*

\* Call the TempSensorExample.

\*/

Status = TempSensorExample(IIC\_DEVICE\_ID, TEMP\_SENSOR\_ADDRESS,

&TemperaturePtr);

xil\_printf("Temp :%d °C \r\n",TemperaturePtr);

if (Status != XST\_SUCCESS) {

xil\_printf("IIC tempsensor connection Failed\r\n");

return XST\_FAILURE;

}

xil\_printf("Successfully ran IIC tempsensor on board\r\n");

return XST\_SUCCESS;

}

int global\_temp; // declaring a global variable

int print\_temp\_reading(){

int Status;

u8 TemperaturePtr;

/\*

\* Call the TempSensor.

\*/

Status = TempSensorExample(IIC\_DEVICE\_ID, TEMP\_SENSOR\_ADDRESS,

&TemperaturePtr);

xil\_printf("Temp :%d °C \r\n",TemperaturePtr);

global\_temp=TemperaturePtr; // assigning temp to a gloabal variable to be ablt to access in the esp sending

if (Status != XST\_SUCCESS) {

xil\_printf("IIC tempsensor Failed\r\n");

return XST\_FAILURE;

}

//xil\_printf("Successfully ran IIC tempsensor \r\n");

// original return XST\_SUCCESS;

return TemperaturePtr;

}

PmodESP32 myESP32;

HostUart myHostUart;

void DemoInitialize () {

HostUart\_Config \*CfgPtr;

EnableCaches();

ESP32\_Initialize(&myESP32, PMODESP32\_UART\_BASEADDR, PMODESP32\_GPIO\_BASEADDR);

CfgPtr = HostUart\_LookupConfig(HOST\_UART\_DEVICE\_ID);

HostUart\_CfgInitialize(&myHostUart, CfgPtr, HostUartConfig\_GetBaseAddr(CfgPtr));

}

void DemoRun() {

u8 recv\_buffer;

u32 num\_received;

u8 buffer[] = "AT+CIPSTART=\"TCP\",\"192.168.173.222\",80\r\n";

u8 buffer\_09[] = "AT+CIPSEND=8\r\n";

xil\_printf("Requesting if ESp32 is ready to send Data\r\n");

ESP32\_SendBuffer(&myESP32, buffer, 45);

xil\_printf("Connecting to TCp Server Rostock\r\n");

int a = 39; // testing with a dummy value

char msg[5];

itoa(a, msg, 10); // converting the recieved temperature value to string to send over the TCP IP protocol to server

int j = 0;

int i = 0;

while (1) {

// TODO: add exit functionality (ctrl-Z?)

num\_received = ESP32\_Recv(&myESP32, &recv\_buffer, 1);

if (num\_received > 0) {

xil\_printf("%c", recv\_buffer);

}

num\_received = HostUart\_Recv(&myHostUart, &recv\_buffer, 1);

if (num\_received > 0) {

xil\_printf("%c", recv\_buffer);

while (0 == ESP32\_Send(&myESP32, &recv\_buffer, 1));

}

i++;

j++;

if (j == 100000) {

// print\_temp\_reading();

// int store=TemperaturePtr;

j = 0;

if (i == 100000) {

ESP32\_SendBuffer(&myESP32, buffer\_09, 14);

}

if (i == 200000) {

print\_temp\_reading();

itoa(global\_temp, msg, 10);

char temp\_data\_string[] = "\r\n ";

char msg\_temp[] = " ^C\r\n";

char nxt\_line[] = "\r\n";

strcat(msg, msg\_temp);

ESP32\_SendBuffer(&myESP32, msg, 25);

ESP32\_SendBuffer(&myESP32, nxt\_line, 4);

i = 0;

}

}

}

}

void DemoCleanup() {

DisableCaches();

}

int main() {

DemoInitialize();

// Oled\_Initialize();

temp\_init();

DemoRun();

DemoCleanup();

return 0;

}

void EnableCaches() {

#ifdef \_\_MICROBLAZE\_\_

#ifdef XPAR\_MICROBLAZE\_USE\_DCACHE

Xil\_DCacheEnable();

#endif

#ifdef XPAR\_MICROBLAZE\_USE\_ICACHE

Xil\_ICacheEnable();

#endif

#endif

}

void DisableCaches() {

#ifdef \_\_MICROBLAZE\_\_

#ifdef XPAR\_MICROBLAZE\_USE\_DCACHE

Xil\_DCacheDisable();

#endif

#ifdef XPAR\_MICROBLAZE\_USE\_ICACHE

Xil\_ICacheDisable();

#endif

#endif

}

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

/\*\*

\* The function reads the temperature of the IIC temperature sensor on the

\* IIC bus. It initializes the IIC device driver and sets it up to communicate

\* with the temperature sensor. This function does contain a loop that polls

\* for completion of the IIC processing such that it may not return if

\* interrupts or the hardware are not working.

\*

\* @param IicDeviceId is the XPAR\_<IIC\_instance>\_DEVICE\_ID value from

\* xparameters.h for the IIC Device

\* @param TempSensorAddress is the address of the Temperature Sensor device

\* on the IIC bus

\* @param TemperaturePtr is the data byte read from the temperature sensor

\*

\* @return XST\_SUCCESS to indicate success, else XST\_FAILURE to indicate

\* a Failure.

\*

\* @note None.

\*

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

int TempSensorExample(u16 IicDeviceId, u8 TempSensorAddress, u8 \*TemperaturePtr)

{

int Status;

static int Initialized = FALSE;

XIic\_Config \*ConfigPtr; /\* Pointer to configuration data \*/

if (!Initialized) {

Initialized = TRUE;

/\*

\* Initialize the IIC driver so that it is ready to use.

\*/

ConfigPtr = XIic\_LookupConfig(IicDeviceId);

if (ConfigPtr == NULL) {

return XST\_FAILURE;

}

Status = XIic\_CfgInitialize(&Iic, ConfigPtr,

ConfigPtr->BaseAddress);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

/\*

\* Setup handler to process the asynchronous events which occur,

\* the driver is only interrupt driven such that this must be

\* done prior to starting the device.

\*/

XIic\_SetRecvHandler(&Iic, (void \*)&HandlerInfo, RecvHandler);

XIic\_SetStatusHandler(&Iic, (void \*)&HandlerInfo,

StatusHandler);

/\*

\* Connect the ISR to the interrupt and enable interrupts.

\*/

Status = SetupInterruptSystem(&Iic);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

/\*

\* Start the IIC driver such that it is ready to send and

\* receive messages on the IIC interface, set the address

\* to send to which is the temperature sensor address

\*/

XIic\_Start(&Iic);

XIic\_SetAddress(&Iic, XII\_ADDR\_TO\_SEND\_TYPE, TempSensorAddress);

}

/\*

\* Clear updated flags such that they can be polled to indicate

\* when the handler information has changed asynchronously and

\* initialize the status which will be returned to a default value

\*/

HandlerInfo.EventStatusUpdated = FALSE;

HandlerInfo.RecvBytesUpdated = FALSE;

Status = XST\_FAILURE;

/\*

\* Attempt to receive a byte of data from the temperature sensor

\* on the IIC interface, ignore the return value since this example is

\* a single master system such that the IIC bus should not ever be busy

\*/

(void)XIic\_MasterRecv(&Iic, TemperaturePtr, 1);

/\*

\* The message is being received from the temperature sensor,

\* wait for it to complete by polling the information that is

\* updated asynchronously by interrupt processing

\*/

while(1) {

if(HandlerInfo.RecvBytesUpdated == TRUE) {

/\*

\* The device information has been updated for receive

\* processing,if all bytes received (1), indicate

\* success

\*/

if (HandlerInfo.RemainingRecvBytes == 0) {

Status = XST\_SUCCESS;

}

break;

}

/\*

\* Any event status which occurs indicates there was an error,

\* so return unsuccessful, for this example there should be no

\* status events since there is a single master on the bus

\*/

if (HandlerInfo.EventStatusUpdated == TRUE) {

break;

}

}

return Status;

}

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

/\*\*

\*

\* This function setups the interrupt system such that interrupts can occur

\* for IIC. This function is application specific since the actual system may

\* or may not have an interrupt controller. The IIC device could be directly

\* connected to a processor without an interrupt controller. The user should

\* modify this function to fit the application.

\*

\* @param IicPtr contains a pointer to the instance of the IIC component

\* which is going to be connected to the interrupt controller.

\*

\* @return XST\_SUCCESS if successful, otherwise XST\_FAILURE.

\*

\* @notes None.

\*

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

int SetupInterruptSystem(XIic \*IicPtr) ///// static was removed from before INT

{

int Status;

/\*

\* Initialize the interrupt controller driver so that it's ready to use,

\* specify the device ID that is generated in xparameters.h

\*/

Status = XIntc\_Initialize(&InterruptController, INTC\_DEVICE\_ID);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

/\*

\* Connect a device driver handler that will be called when an interrupt

\* for the device occurs, the device driver handler performs the

\* specific interrupt processing for the device

\*/

Status = XIntc\_Connect(&InterruptController, INTC\_IIC\_INTERRUPT\_ID,

XIic\_InterruptHandler, IicPtr);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

/\*

\* Start the interrupt controller such that interrupts are recognized

\* and handled by the processor.

\*/

Status = XIntc\_Start(&InterruptController, XIN\_REAL\_MODE);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

/\*

\* Enable the interrupts for the IIC device.

\*/

XIntc\_Enable(&InterruptController, INTC\_IIC\_INTERRUPT\_ID);

/\*

\* Initialize the exception table.

\*/

Xil\_ExceptionInit();

/\*

\* Register the interrupt controller handler with the exception table.

\*/

Xil\_ExceptionRegisterHandler(XIL\_EXCEPTION\_ID\_INT,

(Xil\_ExceptionHandler) XIntc\_InterruptHandler,

&InterruptController);

/\*

\* Enable non-critical exceptions.

\*/

Xil\_ExceptionEnable();

return XST\_SUCCESS;

}

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

/\*\*

\* This receive handler is called asynchronously from an interrupt context and

\* and indicates that data in the specified buffer was received. The byte count

\* should equal the byte count of the buffer if all the buffer was filled.

\*

\* @param CallBackRef is a pointer to the IIC device driver instance which

\* the handler is being called for.

\* @param ByteCount indicates the number of bytes remaining to be received of

\* the requested byte count. A value of zero indicates all requested

\* bytes were received.

\*

\* @return None.

\*

\* @notes None.

\*

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

void RecvHandler(void \*CallbackRef, int ByteCount) ///// static was removed from before INT

{

HandlerInfo.RemainingRecvBytes = ByteCount;

HandlerInfo.RecvBytesUpdated = TRUE;

}

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

/\*\*

\* This status handler is called asynchronously from an interrupt context and

\* indicates that the conditions of the IIC bus changed. This handler should

\* not be called for the application unless an error occurs.

\*

\* @param CallBackRef is a pointer to the IIC device driver instance which the

\* handler is being called for.

\* @param Status contains the status of the IIC bus which changed.

\*

\* @return None.

\*

\* @notes None.

\*

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

void StatusHandler(void \*CallbackRef, int Status) ///// static was removed from before INT

{

HandlerInfo.EventStatus |= Status;

HandlerInfo.EventStatusUpdated = TRUE;

}

//-----------------------OLED----------------------------

void Oled\_Initialize() {

EnableCaches();

OLEDrgb\_begin(&oledrgb, XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_GPIO\_BASEADDR,

XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_SPI\_BASEADDR);

}

void Oled\_Run() {

char ch;

// Define the user definable characters

for (ch = 0; ch < 5; ch++) {

OLEDrgb\_DefUserChar(&oledrgb, ch, &rgbUserFont[ch \* 8]);

}

OLEDrgb\_SetCursor(&oledrgb, 2, 1);

OLEDrgb\_PutString(&oledrgb, "Digilent"); // Default color (green)

OLEDrgb\_SetCursor(&oledrgb, 4, 4);

OLEDrgb\_SetFontColor(&oledrgb, OLEDrgb\_BuildRGB(0, 0, 255)); // Blue font

OLEDrgb\_PutString(&oledrgb, "OledRGB");

OLEDrgb\_SetFontColor(&oledrgb, OLEDrgb\_BuildRGB(200, 200, 44));

OLEDrgb\_SetCursor(&oledrgb, 1, 6);

OLEDrgb\_PutChar(&oledrgb, 4);

OLEDrgb\_SetFontColor(&oledrgb, OLEDrgb\_BuildRGB(200, 12, 44));

OLEDrgb\_SetCursor(&oledrgb, 5, 6);

OLEDrgb\_PutString(&oledrgb, "Demo");

OLEDrgb\_PutChar(&oledrgb, 0);

sleep(5); // Wait 5 seconds

OLEDrgb\_DrawBitmap(&oledrgb, 0, 0, 95, 63, (u8\*) tommy);

}

//------------------------------------------------------