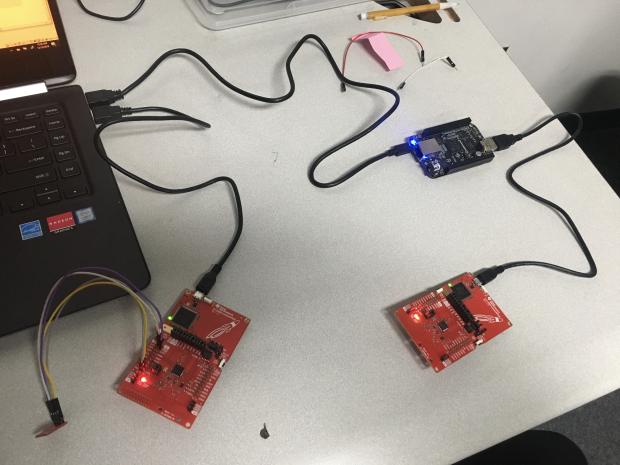
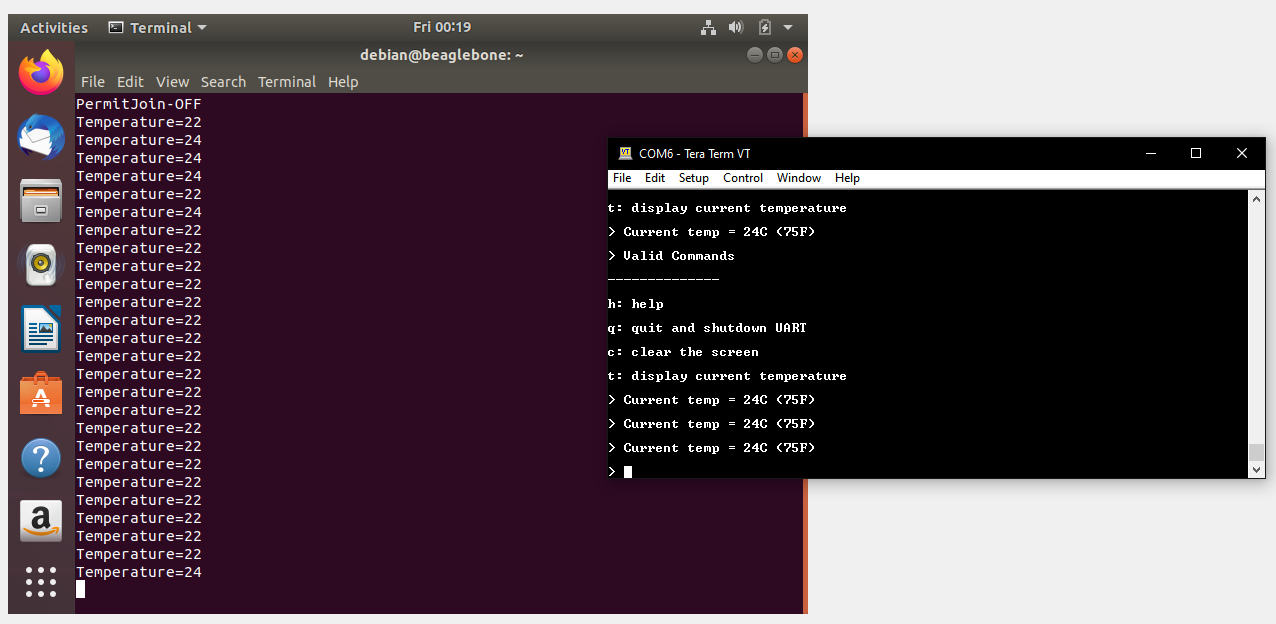
Partner: Darryl Derico

Video Link: <https://www.youtube.com/watch?v=HAaFMS9DRKM>

Code Editor/Compiler: Code Composer Studio  
Serial Connection: PuTTy, Tera Term  
Software: VM Virtual Box, Ubuntu

Final Setup  


Final Result  


Code:

/\*

\* ======== console.c ========

\*/

#include <stdint.h>

#include <string.h>

#include <stdbool.h>

/\* POSIX Header files \*/

#include <pthread.h>

#include <semaphore.h>

/\* Driver Header files \*/

#include "smsgs.h"

#include "mac\_util.h"

#include "api\_mac.h"

#include "sensor.h"

extern Smsgs\_tempSensorField\_t tempSensor;

#include <ti/drivers/GPIO.h>

#include <ti/drivers/UART.h>

#ifdef CC32XX

#include <ti/drivers/Power.h>

#include <ti/drivers/power/PowerCC32XX.h>

#endif

/\* Example/Board Header files \*/

#include "Board.h"

/\* Console display strings \*/

const char consoleDisplay[] = "\fConsole (h for help)\r\n";

const char helpPrompt[] = "Valid Commands\r\n" \

"--------------\r\n" \

"h: help\r\n" \

"q: quit and shutdown UART\r\n" \

"c: clear the screen\r\n" \

"t: display current temperature\r\n";

const char byeDisplay[] = "Bye! Hit button1 to start UART again\r\n";

const char tempStartDisplay[] = "Current temp = ";

const char tempMidDisplay[] = "C (";

const char tempEndDisplay[] = "F)\r\n";

const char cleanDisplay[] = "\f";

const char userPrompt[] = "> ";

const char readErrDisplay[] = "Problem read UART.\r\n";

/\* Used to determine whether to have the thread block \*/

volatile bool uartEnabled = true;

sem\_t semConsole;

/\* Temperature written by the temperature thread and read by console thread \*/

extern volatile float temperature;

extern volatile float temperaturef;

/\* Mutex to protect the reading/writing of the float temperature \*/

extern pthread\_mutex\_t temperatureMutex;

/\* Used itoa instead of sprintf to help minimize the size of the stack \*/

static void itoa(int n, char s[]);

/\*

\* ======== gpioButtonFxn ========

\* Callback function for the GPIO interrupt on Board\_GPIO\_BUTTON1.

\* There is no debounce logic here since we are just looking for

\* a button push. The uartEnabled variable protects use against any

\* additional interrupts cased by the bouncing of the button.

\*/

void gpioButtonFxn(uint\_least8\_t index)

{

/\* If disabled, enable and post the semaphore \*/

if (uartEnabled == false) {

uartEnabled = true;

sem\_post(&semConsole);

}

}

/\*

\* ======== simpleConsole ========

\* Handle the user input. Currently this console does not handle

\* user back-spaces or other "hard" characters.

\*/

void simpleConsole(UART\_Handle uart)

{

char cmd;

int status;

char tempStr[8];

int localTemperatureC;

int localTemperatureF;

UART\_write(uart, consoleDisplay, sizeof(consoleDisplay));

/\* Loop until read fails or user quits \*/

while (1) {

UART\_write(uart, userPrompt, sizeof(userPrompt));

status = UART\_read(uart, &cmd, sizeof(cmd));

if (status == 0) {

UART\_write(uart, readErrDisplay, sizeof(readErrDisplay));

cmd = 'q';

}

switch (cmd) {

case 't':

tempSensor.objectTemp = localTemperatureC;

tempSensor.ambienceTemp = localTemperatureC;

Util\_setEvent(&Sensor\_events, EXT\_SENSOR\_READING\_TIMEOUT\_EVT);

UART\_write(uart, tempStartDisplay, sizeof(tempStartDisplay));

/\*

\* Make sure we are accessing the global float temperature variables

\* in a thread-safe manner.

\*/

pthread\_mutex\_lock(&temperatureMutex);

localTemperatureC = (int)temperature;

localTemperatureF = (int)temperaturef;

pthread\_mutex\_unlock(&temperatureMutex);

itoa((int)localTemperatureC, tempStr);

UART\_write(uart, tempStr, strlen(tempStr));

UART\_write(uart, tempMidDisplay, sizeof(tempMidDisplay));

itoa((int)localTemperatureF, tempStr);

UART\_write(uart, tempStr, strlen(tempStr));

UART\_write(uart, tempEndDisplay, sizeof(tempEndDisplay));

break;

case 'c':

UART\_write(uart, cleanDisplay, sizeof(cleanDisplay));

break;

case 'q':

UART\_write(uart, byeDisplay, sizeof(byeDisplay));

return;

case 'h':

default:

UART\_write(uart, helpPrompt, sizeof(helpPrompt));

break;

}

}

}

/\*

\* ======== consoleThread ========

\*/

void \*consoleThread(void \*arg0)

{

UART\_Params uartParams;

UART\_Handle uart;

int retc;

#ifdef CC32XX

/\*

\* The CC3220 examples by default do not have power management enabled.

\* This allows a better debug experience. With the power management

\* enabled, if the device goes into a low power mode the emulation

\* session is lost.

\* Let's enable it and also configure the button to wake us up.

\*/

PowerCC32XX\_Wakeup wakeup;

PowerCC32XX\_getWakeup(&wakeup);

wakeup.wakeupGPIOFxnLPDS = gpioButtonFxn;

PowerCC32XX\_configureWakeup(&wakeup);

Power\_enablePolicy();

#endif

/\* Configure the button pin \*/

GPIO\_setConfig(Board\_GPIO\_BUTTON1, GPIO\_CFG\_IN\_PU | GPIO\_CFG\_IN\_INT\_FALLING);

/\* install Button callback and enable it \*/

GPIO\_setCallback(Board\_GPIO\_BUTTON1, gpioButtonFxn);

GPIO\_enableInt(Board\_GPIO\_BUTTON1);

retc = sem\_init(&semConsole, 0, 0);

if (retc == -1) {

while (1);

}

UART\_init();

/\*

\* Initialize the UART parameters outside the loop. Let's keep

\* most of the defaults (e.g. baudrate = 115200) and only change the

\* following.

\*/

UART\_Params\_init(&uartParams);

uartParams.writeDataMode = UART\_DATA\_BINARY;

uartParams.readDataMode = UART\_DATA\_BINARY;

uartParams.readReturnMode = UART\_RETURN\_FULL;

/\* Loop forever to start the console \*/

while (1) {

if (uartEnabled == false) {

retc = sem\_wait(&semConsole);

if (retc == -1) {

while (1);

}

}

/\* Create a UART for the console \*/

uart = UART\_open(Board\_UART0, &uartParams);

if (uart == NULL) {

while (1);

}

simpleConsole(uart);

/\*

\* Since we returned from the console, we need to close the UART.

\* The Power Manager will go into a lower power mode when the UART

\* is closed.

\*/

UART\_close(uart);

uartEnabled = false;

}

}

/\*

\* The following function is from good old K & R.

\*/

static void reverse(char s[])

{

int i, j;

char c;

for (i = 0, j = strlen(s)-1; i<j; i++, j--) {

c = s[i];

s[i] = s[j];

s[j] = c;

}

}

/\*

\* The following function is from good old K & R.

\*/

static void itoa(int n, char s[])

{

int i, sign;

if ((sign = n) < 0) /\* record sign \*/

n = -n; /\* make n positive \*/

i = 0;

do { /\* generate digits in reverse order \*/

s[i++] = n % 10 + '0'; /\* get next digit \*/

} while ((n /= 10) > 0); /\* delete it \*/

if (sign < 0)

s[i++] = '-';

s[i] = '\0';

reverse(s);

}

/\*

\* ======== temperature.c ========

\*/

#include <stdint.h>

#include <stddef.h>

#include <unistd.h>

#include <ti/display/Display.h>

/\* POSIX Header files \*/

#include <pthread.h>

#include <semaphore.h>

#include <signal.h>

#include <time.h>

/\* Driver Header files \*/

#include <ti/drivers/GPIO.h>

#include <ti/drivers/I2C.h>

/\* Example/Board Header files \*/

#include "Board.h"

/\* ======== Si7021 Registers ======== \*/

#define Si7021\_TMP\_REG 0xE3

#define Si7021\_HUM\_REG 0xE5

#define Si7021\_ADDR 0x40

/\*

\* ======== HIGH\_TEMP ========

\* Send alert when this temperature (in Celsius) is exceeded

\*/

#define HIGH\_TEMP 30

/\*

\* ======== TMP Registers ========

\*/

#define TMP006\_REG 0x0001 /\* Die Temp Result Register for TMP006 \*/

#define TMP116\_REG 0x0000 /\* Die Temp Result Register for TMP116 \*/

/\*

\* The CC32XX LaunchPads come with an on-board TMP006 or TMP116 temperature

\* sensor depending on the revision. Newer revisions come with the TMP116.

\* The Build Automation Sensors (BOOSTXL-BASSENSORS) BoosterPack

\* contains a TMP116.

\*

\* We are using the DIE temperature because it's cool!

\*

\* Additionally: no calibration is being done on the TMPxxx device to simplify

\* the example code.

\*/

#define TMP006\_ADDR 0x41;

#define TMP116\_BP\_ADDR 0x48;

#define TMP116\_LP\_ADDR 0x49;

/\* Temperature written by the temperature thread and read by console thread \*/

volatile float temperatureC;

volatile float temperatureF;

volatile float temperaturef;

volatile float temperature;

volatile float temp;

volatile float sample;

Display\_Handle display;

/\* Mutex to protect the reading/writing of the temperature variables \*/

extern pthread\_mutex\_t temperatureMutex;

/\*

\* ======== clearAlert ========

\* Clear the LED

\*/

//static void clearAlert(float temperature)

//{

// GPIO\_write(Board\_GPIO\_LED0, Board\_GPIO\_LED\_OFF);

//}

/\*

\* ======== sendAlert ========

\* Okay, just light a LED in this example, but with the SimpleLink SDK,

\* you could send it out over the radio to something cool!

\*/

//static void sendAlert(float temperature)

//{

// GPIO\_write(Board\_GPIO\_LED0, Board\_GPIO\_LED\_ON);

//}

/\*

\* ======== postSem ========

\* Function called when the timer (created in setupTimer) expires.

\*/

static void postSem(union sigval val)

{

sem\_t \*sem = (sem\_t\*)(val.sival\_ptr);

sem\_post(sem);

}

/\*

\* ======== setupTimer ========

\* Create a timer that will expire at the period specified by the

\* time arguments. When the timer expires, the passed in semaphore

\* will be posted by the postSem function.

\*

\* A non-zero return indicates a failure.

\*/

int setupTimer(sem\_t \*sem, timer\_t \*timerid, time\_t sec, long nsec)

{

struct sigevent sev;

struct itimerspec its;

int retc;

retc = sem\_init(sem, 0, 0);

if (retc != 0) {

return(retc);

}

/\* Create the timer that wakes up the thread that will pend on the sem. \*/

sev.sigev\_notify = SIGEV\_SIGNAL;

sev.sigev\_value.sival\_ptr = sem;

sev.sigev\_notify\_function = &postSem;

sev.sigev\_notify\_attributes = NULL;

retc = timer\_create(CLOCK\_MONOTONIC, &sev, timerid);

if (retc != 0) {

return(retc);

}

/\* Set the timer to go off at the specified period \*/

its.it\_interval.tv\_sec = sec;

its.it\_interval.tv\_nsec = nsec;

its.it\_value.tv\_sec = sec;

its.it\_value.tv\_nsec = nsec;

retc = timer\_settime(\*timerid, 0, &its, NULL);

if (retc != 0) {

timer\_delete(\*timerid);

return(retc);

}

return(0);

}

/\*

\* ======== temperatureThread ========

\* This thread reads the temperature every second via I2C and sends an

\* alert if it goes above HIGH\_TEMP.

\*/

void \*temperatureThread(void \*arg0)

{

uint8\_t txBuffer[1];

uint8\_t rxBuffer[2];

I2C\_Handle i2c;

I2C\_Params i2cParams;

I2C\_Transaction i2cTransaction;

sem\_t semTimer;

// timer\_t timerid;

// int retc;

/\* Configure the LED and if applicable, the TMP116\_EN pin \*/

GPIO\_setConfig(Board\_GPIO\_LED0, GPIO\_CFG\_OUT\_STD | GPIO\_CFG\_OUT\_LOW);

#ifdef Board\_GPIO\_TMP116\_EN

GPIO\_setConfig(Board\_GPIO\_TMP116\_EN, GPIO\_CFG\_OUT\_STD | GPIO\_CFG\_OUT\_HIGH);

/\* 1.5 ms reset time for the TMP116 \*/

sleep(1);

#endif

/\*

\* Create/Open the I2C that talks to the TMP sensor

\*/

I2C\_init();

Display\_init();

I2C\_Params\_init(&i2cParams);

i2cParams.bitRate = I2C\_400kHz;

i2c = I2C\_open(Board\_I2C\_TMP, &i2cParams);

if (i2c == NULL) {

while (1);

}

/\* Common I2C transaction setup \*/

i2cTransaction.writeBuf = txBuffer;

i2cTransaction.writeCount = 1;

i2cTransaction.readBuf = rxBuffer;

i2cTransaction.readCount = 2;

/\*

\* Determine which I2C sensor is present.

\* We will prefer sensors in this order: TMP116 (on BoosterPacks),

\* TMP116 (on-board CC32XX LaunchPads), and last TMP006

\* (on older CC32XX LaunchPads).

\*/

/\*

// Try TMP116 values

txBuffer[0] = TMP116\_REG;

i2cTransaction.slaveAddress = TMP116\_BP\_ADDR;

if (!I2C\_transfer(i2c, &i2cTransaction)) {

// Not BP TMP116, try LP TMP116

i2cTransaction.slaveAddress = TMP116\_LP\_ADDR;

if (!I2C\_transfer(i2c, &i2cTransaction)) {

// Not a TMP116 try TMP006

txBuffer[0] = TMP006\_REG;

i2cTransaction.slaveAddress = TMP006\_ADDR;

if (!I2C\_transfer(i2c, &i2cTransaction)) {

// Could not resolve a sensor, error

while(1);

}

}

}

\*/

// Try Si7021

txBuffer[0] = Si7021\_TMP\_REG;

i2cTransaction.slaveAddress = Si7021\_ADDR;

if (!I2C\_transfer(i2c, &i2cTransaction))

{

// Could not resolve a sensor, error

Display\_printf(display, 0, 0, "Error. No TMP sensor found!");

while(1);

}

else

{

Display\_printf(display, 0, 0, "Detected Si7021 sensor.");

}

// Take 20 samples and print them out onto the console

for (sample = 0; sample < 100; sample++)

{

if (I2C\_transfer(i2c, &i2cTransaction))

{

//

// Extract degrees C from the received data;

// see Si7021 datasheet

//

temp = (rxBuffer[0] << 8) | (rxBuffer[1]);

temperature = (((175.72 \* temp)/ 65536) - 46.85); // celsius

temperaturef = (temperature \* (1.8)) + 32; //farenheit

Display\_printf(display, 0, 0, "Sample %u: %d (C)", sample, temperaturef);

}

else

{

Display\_printf(display, 0, 0, "I2C Bus fault.");

}

}

/\*

\* The temperature thread blocks on the semTimer semaphore, which the

\* timerId timer will post every second. The timer is created in the

\* setupTimer function. It's returned so the thread could change the

\* period or delete it if desired.

\*/

// retc = setupTimer(&semTimer, &timerid, 1, 0);

// if (retc != 0) {

// while (1);

// }

// while (1)

// {

// if (I2C\_transfer(i2c, &i2cTransaction)) {

// /\*

// \* Extract degrees C from the received data; see sensor datasheet.

// \* Make sure we are updating the global temperature variables

// \* in a thread-safe manner.

// \*/

// pthread\_mutex\_lock(&temperatureMutex);

// temperatureC = (rxBuffer[0] << 6) | (rxBuffer[1] >> 2);

// temperatureC \*= 0.03125;

// temperatureF = temperatureC \* 9 / 5 + 32;

// pthread\_mutex\_unlock(&temperatureMutex);

//

// /\* Send an alert if the temperature is too high!! \*/

// if ((int)temperatureC >= HIGH\_TEMP) {

// sendAlert(temperatureC);

// }

// else {

// clearAlert(temperatureC);

// }

// }

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

/\*

// Common I2C transaction setup

i2cTransaction.writeBuf = txBuffer;

i2cTransaction.writeCount = 1;

i2cTransaction.readBuf = rxBuffer;

i2cTransaction.readCount = 2;

\*/

/\*

// Try Si7021

txBuffer[0] = Si7021\_TMP\_REG;

i2cTransaction.slaveAddress = Si7021\_ADDR;

if (!I2C\_transfer(i2c, &i2cTransaction))

{

// Could not resolve a sensor, error

Display\_printf(display, 0, 0, "Error. No TMP sensor found!");

while(1);

}

else

{

Display\_printf(display, 0, 0, "Detected Si7021 sensor.");

}

// Take 20 samples and print them out onto the console

for (sample = 0; sample < 20; sample++)

{

if (I2C\_transfer(i2c, &i2cTransaction))

{

//

// Extract degrees C from the received data;

// see Si7021 datasheet

//

temperature = (rxBuffer[0] << 8) | (rxBuffer[1]);

temperaturef = (((175.72 \* temperature)/ 65536) - 46.85);

Display\_printf(display, 0, 0, "Sample %u: %d (C)", sample, temperaturef);

}

else

{

Display\_printf(display, 0, 0, "I2C Bus fault.");

}

}

\*/

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

// /\* Block until the timer posts the semaphore. \*/

// retc = sem\_wait(&semTimer);

// if (retc == -1) {

// while (1);

// }

// }

}

/\*

\* ======== main\_tirtos.c ========

\*/

#include <stdint.h>

/\* POSIX Header files \*/

#include <pthread.h>

/\* RTOS header files \*/

#include <ti/sysbios/BIOS.h>

/\* Driver header files \*/

#include <ti/drivers/GPIO.h>

/\* Example/Board Header files \*/

#include <ti/drivers/Board.h>

/\* Mutex to protect the reading/writing of the temperature variables \*/

pthread\_mutex\_t temperatureMutex;

extern void \*temperatureThread(void \*arg0);

extern void \*consoleThread(void \*arg0);

/\* Stack size in bytes. Large enough in case debug kernel is used. \*/

#define THREADSTACKSIZE 1024

/\*

\* ======== main ========

\*/

int main\_app(void)

{

pthread\_t thread;

pthread\_attr\_t attrs;

struct sched\_param priParam;

int retc;

/\* Call driver init functions \*/

//Board\_init();

/\* Initialize the attributes structure with default values \*/

pthread\_attr\_init(&attrs);

/\* Set priority, detach state, and stack size attributes \*/

priParam.sched\_priority = 1;

retc = pthread\_attr\_setschedparam(&attrs, &priParam);

retc |= pthread\_attr\_setdetachstate(&attrs, PTHREAD\_CREATE\_DETACHED);

retc |= pthread\_attr\_setstacksize(&attrs, THREADSTACKSIZE);

if (retc != 0) {

/\* failed to set attributes \*/

while (1) {}

}

retc = pthread\_create(&thread, &attrs, consoleThread, NULL);

if (retc != 0) {

/\* pthread\_create() failed \*/

while (1) {}

}

/\*

\* Let's make the temperature thread a higher priority .

\* Higher number means higher priority in TI-RTOS.

\*/

priParam.sched\_priority = 2;

retc = pthread\_attr\_setschedparam(&attrs, &priParam);

if (retc != 0) {

/\* failed to set priority \*/

while (1) {}

}

retc = pthread\_create(&thread, &attrs, temperatureThread, NULL);

if (retc != 0) {

/\* pthread\_create() failed \*/

while (1) {}

}

/\* Create a mutex that will protect temperature variables \*/

retc = pthread\_mutex\_init(&temperatureMutex, NULL);

if (retc != 0) {

/\* pthread\_mutex\_init() failed \*/

while (1) {}

}

/\* Initialize the GPIO since multiple threads are using it \*/

//GPIO\_init();

/\* Start the TI-RTOS scheduler \*/

//BIOS\_start();

return (0);

}