kgarcia-snhu Update README.md History

1 contributor

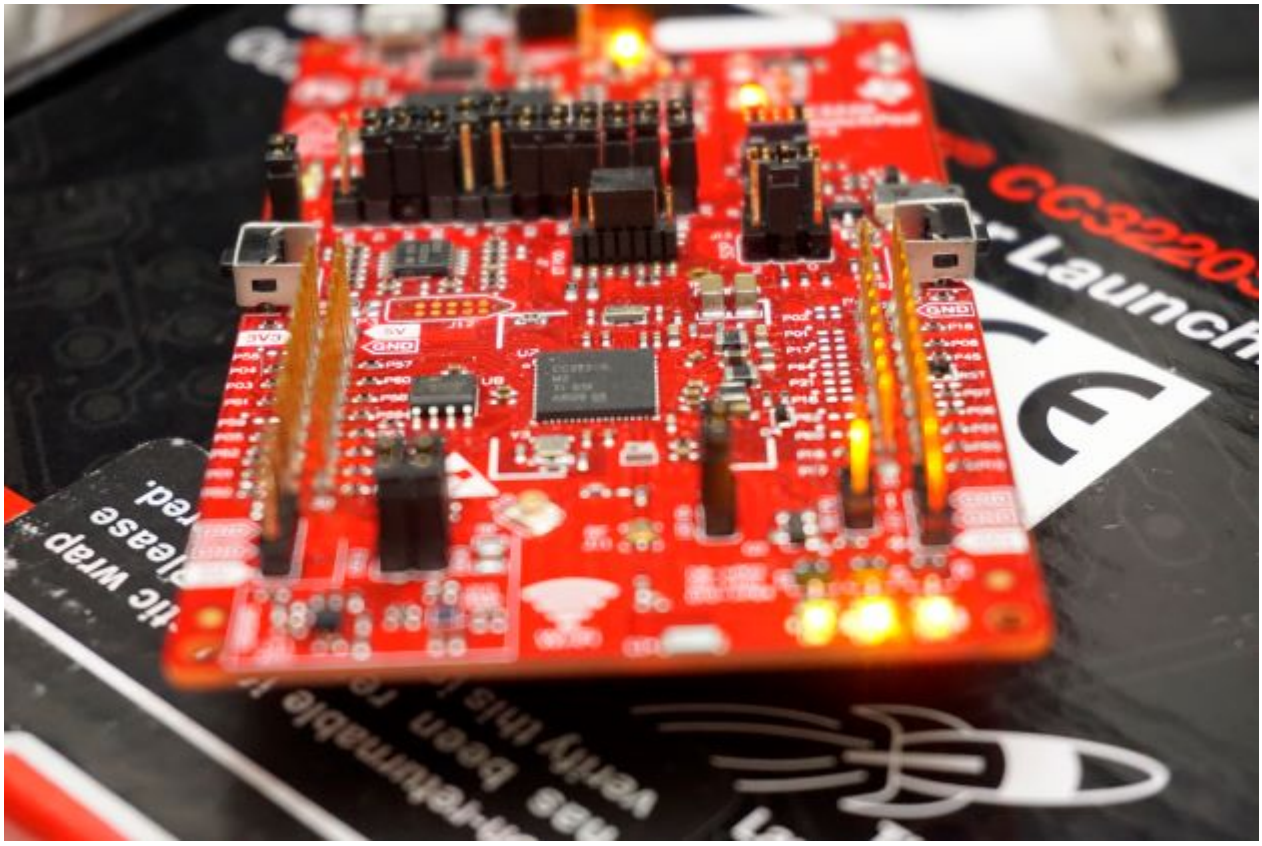
243 lines (172 sloc) | 12.1 KB

CS-350-Emerging-Sys-Arch-Tech

Emerging systems, architectures, and technologies

Summarize the project and what problem it was solving?

The thermostat supports the peripherals of I2C by reading the TMP006 temperature sensor, the peripherals of GPIO by activating a LED to indicate the output to the thermostat where LED on = heat on and LED off = heat off, and the UART peripherals by simulating data being sent to the server which can be viewed from the terminal Com3 echo characters written for <30,26,0,0339> <temperature(30), setpoint(26), heat(0), seconds>. When looking into the uses for hardware architecture of Texas Instruments, Microchip, and Freescale. I found that Texas Instruments Real-Time Microcontrollers are of lower cost and provide a diverse amount of implementation due to the 32bit configuration and amount of pins (38 pins). The Freescale architecture is intended to be used more for wireless applications like broadband routers, lowering power consumption yet increasing performance while using a single chip. The Microchip architecture contains a larger flash and ram capacity, uses a lower bit count, and includes (RTCC) for clock and calendar functions. The microchip performs well with integrated touch functionality for LCD displays. (Design News, 2017) To create a smart thermostat using the TI board, my recommendation is to use a Freescale architecture. The TI board CC3220S is a single-chip wireless MCU with 0MB of flash memory, 256kb of RAM, Arm Cortex M4 processor at 80Mhz. This prototype is developed to showcase how the CC3220S board can act as a thermostat accessing a temperature sensor (TMP006) to read the room temp (via I2C) degrees Celsius. Simulating data transfer to the server via terminal from the Com port (via UART) and indicate if the heating system is on or off by displaying a led light (via GPIO) when the temperature is different than the setpoint. The SysTec's thermostat architecture must have enough Flash and RAM to support the code and performance. Therefore, I believe that a Freescale architecture for a thermostat would resemble a Google Nest device. This architecture is geared towards wireless connectivity allowing access to the thermostat from mobile applications, containing limited functionality on the device as settings are controlled wirelessly. Regarding the other styles of architecture, a microchip architecture for a thermostat would resemble an older style of a temperature control unit that uses analog input like a dial and has no LCD display. A TI architecture would resemble a Honeywell digital thermostat that provides the user with input by digital buttons and contains an LCD display showcasing settings, option menus, and alerts.



/* ========== Summary ========== */

Application that toggles an LED(s) using a GPIO pin interrupt.

Peripherals & Pin Assignments When this project is built, the SysConfig tool will generate the TI-Driver configurations into the ti_drivers_config.c and ti_drivers_config.h files. Information on pins and resources used is present in both generated files. Additionally, the System Configuration file (*.syscfg) present in the project may be opened with SysConfig's graphical user interface to determine pins and resources used.

CONFIG_GPIO_LED_0 - Indicates that the board was initialized within mainThread() also toggled by CONFIG_GPIO_BUTTON_0 CONFIG_GPIO_LED_1 - Toggled by CONFIG_GPIO_BUTTON_1 CONFIG_GPIO_BUTTON_0 - Toggles CONFIG_GPIO_LED_0 CONFIG_GPIO_BUTTON_1 - Toggles CONFIG_GPIO_LED_1 BoosterPacks, Board Resources & Jumper Settings For board specific jumper settings, resources and BoosterPack modifications, refer to the Board.html file.

If you're using an IDE such as Code Composer Studio (CCS) or IAR, please refer to Board.html in your project directory for resources used and board-specific jumper settings.

The Board.html can also be found in your SDK installation:

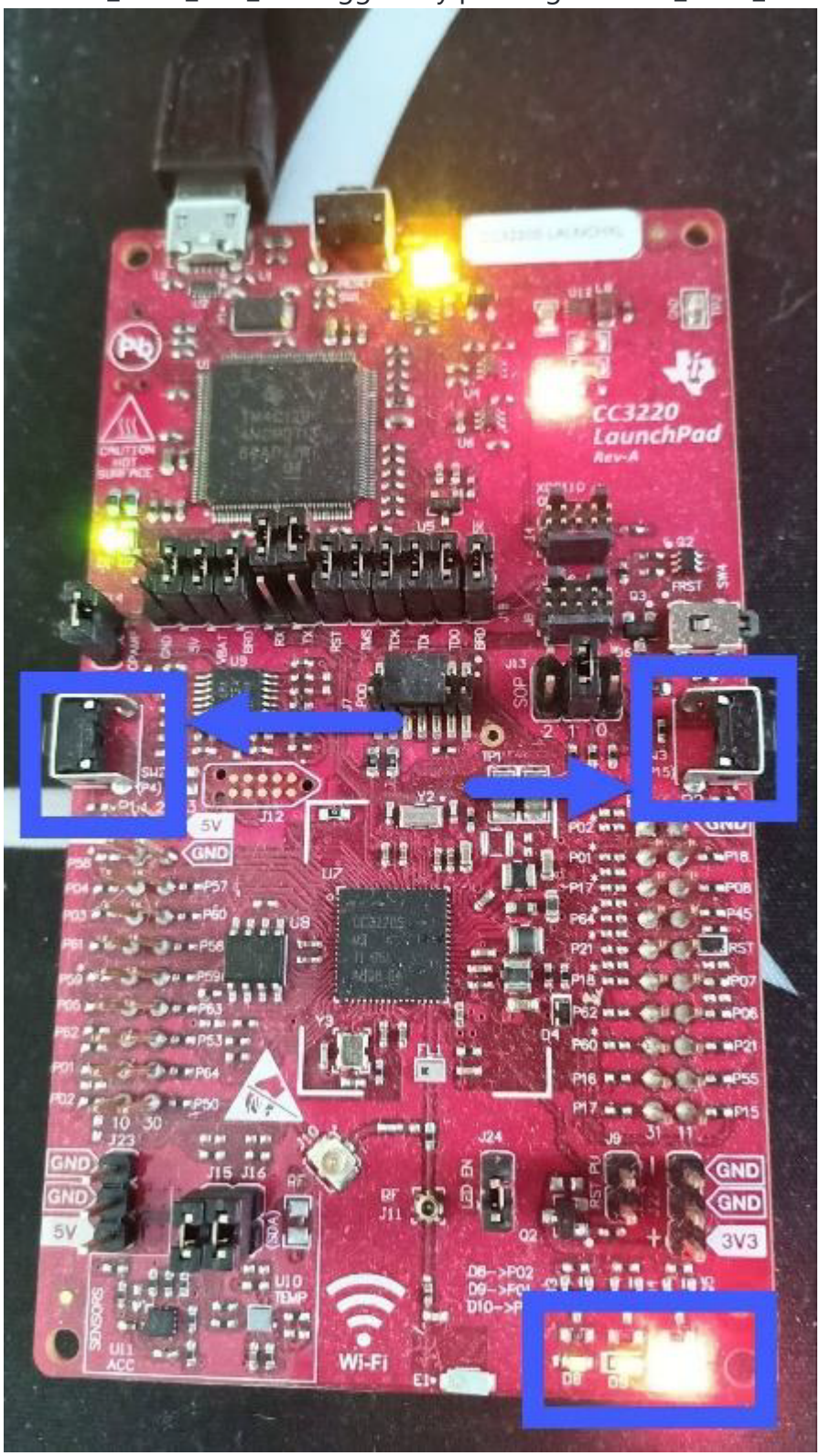
<SDK_INSTALL_DIR>/source/ti/boards/

/* ========== Example Usage ========== */

Run the example. CONFIG_GPIO_LED_0 turns ON to indicate driver initialization is complete.

CONFIG_GPIO_LED_0 is toggled by pushing CONFIG_GPIO_BUTTON_0.

CONFIG_GPIO_LED_1 is toggled by pushing CONFIG_GPIO_BUTTON_1.



Application Design Details The gpioButtonFxn0/1 functions are configured in the driver configuration file. These functions are called in the context of the GPIO interrupt.

Not all boards have more than one button, so CONFIG_GPIO_LED_1 may not be toggled.

There is no button de-bounce logic in the example.

TI-RTOS:

When building in Code Composer Studio, the configuration project will be imported along with the example. These projects can be found under <SDK_INSTALL_DIR>/kernel/tirtos/builds//(release|debug)/(ccs|gcc). The configuration project is referenced by the example, so it will be built first. The "release" configuration has many debug features disabled. These features include assert checking, logging and runtime stack checks. For a detailed difference between the "release" and "debug" configurations, please refer to the TI-RTOS Kernel User's Guide. FreeRTOS:

Please view the FreeRTOSConfig.h header file for example configuration information.

/* ========== Headers ========== */

/* Driver Header file */ `#include <ti/drivers/GPIO.h>`

/* Driver configuration */ `#include "ti_drivers_config.h"`

/* Timer header file */ `#include <ti/drivers/Timer.h>`

/* I2C header file */ `#include <ti/drivers/I2C.h>`

/* UART header file */ `#include <ti/drivers/UART.h>`

/* ========== UART Driver Params ========== */

// UART Global Variables char output[64]; int bytesToSend;

// Driver Handles - Global variables UART_Handle uart;

```
void initUART(void) {
    UART_Params uartParams;
    UART_init();

    // Configure the driver
    UART_Params_init(&uartParams);
    uartParams.writeDataMode = UART_DATA_BINARY;
    uartParams.readDataMode = UART_DATA_BINARY;
    uartParams.readReturnMode = UART_RETURN_FULL;
    uartParams.baudRate = 115200;``

    // Open the driver
    uart = UART_open(CONFIG_UART_0, &uartParams);
    if (uart == NULL) {
        /* UART_open() failed */
        while (1);
```

```
    }
}
```

/* ========== I2C Driver Params ========== */

```
I2C Global Variables static const struct { uint8_t address; uint8_t resultReg;
char *id; } sensors[3] = { { 0x48, 0x0000, "11X" }, { 0x49, 0x0000, "116" }, {
0x41,     0x0001, "006" } };

uint8_t txBuffer[1]; uint8_t rxBuffer[2]; I2C_Transaction i2cTransaction;
```

// Driver Handles - Global variables I2C_Handle i2c;

// Make sure you call initUART() before calling this function. void initI2C(void) { int8_t i, found; I2C_Params i2cParams;

```
DISPLAY(snprintf(output, 64, "Initializing I2C Driver - "))
```

// Init the driver I2C_init();

// Configure the driver I2C_Params_init(&i2cParams); i2cParams.bitRate = I2C_400kHz;

// Open the driver i2c = I2C_open(CONFIG_I2C_0, &i2cParams); if (i2c == NULL) { DISPLAY(snprintf(output, 64, "Failed\n\r")) while (1); }

```
DISPLAY(snprintf(output, 32, "Passed\n\r"))
```

// Boards were shipped with different sensors. // Welcome to the world of embedded systems. // Try to determine which sensor we have. // Scan through the possible sensor addresses

/* Common I2C transaction setup */ i2cTransaction.writeBuf = txBuffer; i2cTransaction.writeCount = 1; i2cTransaction.readBuf = rxBuffer; i2cTransaction.readCount = 0;``

```
found = false;
for (i = 0; i < 3; ++i)
{
    i2cTransaction.slaveAddress = sensors[i].address;
    txBuffer[0] = sensors[i].resultReg;

    DISPLAY(snprintf(output, 64, "Is this %s? ", sensors[i].id))
```

```
    if (I2C_transfer(i2c, &i2cTransaction))
    {
        DISPLAY(snprintf(output, 64, "Found\n\r"))
        found = true;
        break;
    }
    DISPLAY(snprintf(output, 64, "No\n\r"))
}

if(found)
{
    DISPLAY(snprintf(output, 64, "Detected TMP%s I2C address: %x\n\r",
sensors[i].id, i2cTransaction.slaveAddress))
} else {
    DISPLAY(snprintf(output, 64, "Temperature sensor not found, contact
professor\n\r"))
}
}

int16_t readTemp(void) { int j; int16_t temperature = 0;

i2cTransaction.readCount = 2;
if (I2C_transfer(i2c, &i2cTransaction))
{
/*
 * Extract degrees C from the received data;
 * see TMP sensor datasheet
 */
    temperature = (rxBuffer[0] << 8) | (rxBuffer[1]);
    temperature *= 0.0078125;
    /*
    * If the MSB is set '1', then we have a 2's complement
    * negative value which needs to be sign extended
    */
    if (rxBuffer[0] & 0x80)
    {
        temperature |= 0xF000;
    }
} else {
    DISPLAY(snprintf(output, 64, "Error reading temperature sensor (%d)\n\r",
i2cTransaction.status))
    DISPLAY(snprintf(output, 64, "Please power cycle your board by unplugging USB
and plugging back in.\n\r"))
}
return temperature;
}
```

/* ========== Timer Driver Params ========== */

// Driver Handles - Global variables Timer_Handle timer0;

```
void timerCallback(Timer_Handle myHandle, int_fast16_t status) { /* Raise timer
flag */ TimerFlag = 1; }
```

/* Initialize Timer and set Params */ void initTimer(void) { //Timer_Handle timer0; Timer_Params params;

/* Init the driver */ Timer_init();

/* Configure the driver */ Timer_Params_init(&params); params.period = 100000; // 100ms = 100000us used for temperature, setpoint, and heat 200ms, 500ms, 1000ms default 1000000 or 1sec params.periodUnits = Timer_PERIOD_US; params.timerMode = Timer_CONTINUOUS_CALLBACK; params.timerCallback = timerCallback;``

// Open the driver timer0 = Timer_open(CONFIG_TIMER_0, &params);

```
if (timer0 == NULL) {
    /* Failed to initialized timer */
    while (1) {}
}
if (Timer_start(timer0) == Timer_STATUS_ERROR) {
    /* Failed to start timer */
    while (1) {}
}
```

/* ======== gpioButtonFxn0 ======== */

Callback function for the GPIO interrupt on CONFIG_GPIO_BUTTON_0. Note: GPIO interrupts are cleared prior to invoking callbacks. / void gpioButtonFxn0(uint_least8_t index) { / Toggle a temperature setpoint for LED params */ Button0_Flag = 1; } / ======== gpioButtonFxn1 ======== */

Callback function for the GPIO interrupt on CONFIG_GPIO_BUTTON_1. Note: GPIO interrupts are cleared prior to invoking callbacks. / void gpioButtonFxn1(uint_least8_t index) { / Toggle a temperature setpoint for LED params */ Button1_Flag = 1; } // Temperature variables int16_t setpoint; // Desired Temperature set by buttons to increase or decrease value (UART simulation sent to server via wifi using terminal on Com3 port) int16_t temperature; // Sensor Temperature located on Launchpad near WIFI logo, small square chip TMP006 (via I2C) int8_t heat; // Indicates a 0 or 1, 0 = Heater and Temperature Light is off, 1 = Heater and Temperature Light are on (via GPIO and interrupt)

What tools and/or resources are you adding to your support network? I've been using digit, geeksforgeeks, tutorialspoint, SEI Cert, Barrgroup, w3schools, scientific research, and The Coding Lib

https://www.digit.in/technology-guides/fasttrack-to-embedded-systems/what-are-embedded-systems.html

https://www.geeksforgeeks.org/

https://www.tutorialspoint.com/cprogramming/c_overview.htm

https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

https://barrgroup.com/embedded-systems/how-to/coding-state-machines

https://www.w3schools.com/

https://www.scirp.org/index.aspx

The Coding Lib - https://www.youtube.com/watch?v=R2gIVQZU24U

What skills from this project will be particularly transferable to other projects and/or course work? The knowledge and skill of state machine implementation, diagrams, and the components along with the resources that explain why limitations and constraints contribute to improved functionality will help in other projects.