DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

A Wireless 3D Embedded RTOS Human Computer Interface

Ciarán O'Donnell

April 2019

BACHELOR OF ENGINEERING

IN

Electronic & Computer Engineering

MAJOR IN

Digital Interaction

Supervised by Dr. Derek Molloy

Acknowledgements

I would like to thank my supervisor Dr. Derek Molloy for his guidance, enthusiasm and commitment to this project, as well as his help in procuring parts for the project. Thanks, are also due to my family and friends who have been supportive of me and my work over the past few months as I have been less available to them and in more need of warmth and friendliness when I was. Finally, I would like to express my deep appreciation to my girlfriend Rachel who has been a massive help both in picking up on personal life requirements when I could not and emotional support through both this project and my entire university career to date.

Declaration

Abstract

This report investigates the design of a 3D Wireless RTOS Human Computer Interaction device. A device that would allow a user to interact with 3D-Elements on a computer in an intuitive and comprehensive way. As software becomes more capable of computing complex 3D information in real-time the way people interact with this data may need to develop alongside. This report investigates the design of a possible device to fill that purpose.

This report has a large amount of content on the development of a real-time operating system embedded system as part of this project. A lot of information here will be relevant to those with an interest in the development of such devices.

The device was designed to be built upon a Texas Instruments CC2650, with the use of an ADXL345 accelerometer. A lot of content revolves around the communication between these two devices and the wireless capabilities of the CC2650. The work completed with the Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C) interfaces make up a strong portion of the work completed within this project. Further development is needed to reach a working prototype than was completed in this report however the information and lessons learnt here could be further built upon to reach that point.

Table of Contents

Table of Figures

Chapter 1 - Introduction

Computer interaction has become daily life for millions of people around the world. Many computer devices have become a necessity in our day to day lives. Therefore, how people interact with computers has become very important. There are currently several standard ways to interact with computers including keyboard, mouse, monitor and speakers. However recently many more possible methods of interacting with computers have become more viable, including touchscreen interfaces, face and/or eye tracking, speech recognition.

This project will investigate the possibility of using a real-time operating system micro-controller as a base to design a device that allows a user to have 3D interactivity with a computer. This would allow for manipulation of 3D data such as architectural models, medical imagery, or an avatar within a video game environment.

Throughout the project a lot of research will be done on the design of an input device for a host computer, wireless communication of data, and the programming of a real time operating system device. This project has elected to use a device supporting TI-RTOS a specific real time operating system environment and therefore a large quantity of the project is dealing with the use and functionality of the software, so it may be of great interest to those working in the same environment but there is also a lot of information relevant to general RTOS's. Finally, another major element of this project is the use of SPI and I2C interfaces to communicate between to devices. Those interested in the interfacing of electronic devices such as this will find a lot of interesting information on this in the following report.

Chapter 2 - Technical Background

2.1 Human Computer Interaction

2.1.1 Current HCI Devices

Human computer interaction refers to the ability for a person to give input or take input from a computer device. There are already a vast number of devices that allow this. The most common human computer interface devices include, keyboard, mouse, display and speakers, but any device that a person may use to interact with a computer is a HCI. The device that will be investigated in this report is a HCI that allows a person to input 3D data to the computer.

2.1.2 3D HCI Devices

The 3D HCI device investigated within this report should track either 3D positional or rotational data and input to the computer. There have been similar devices designed before. The first major consumer 3D HCI device is the controller of the Nintendo game console called the “Wii”. It can be seen in the devices patent that was granted in the US on 09/09/2008, that the controller uses acceleration data collected before and after a predetermined amount of time to calculate the relative motion of the controller across two axis. [1] This allows the device to communicate rotational data to the console. The controller can also calculate positional data by means of an ultra-violet camera onboard the device. This works in tandem with ultraviolet LEDs placed across from the device as stated in this US patent assigned to Nintendo Co. [2] This gives the device information to determine its 3D positional data along a single axis parallel to the LEDs.

Another method of 3-dimensional human computer interfacing is displayed by Iason Oikonomidis, Nikolaos Kyriazis, and Antonis A. Argyros in their design for 3D Tracking of Hand Articulations. [3] This design uses the Xbox ‘Kinect’ camera and infrared sensor arrays and image processing techniques to quantify 3D information.

These are two examples of 3D HCI devices that show some of the mothods possible to communicate 3D information with a computer. However these devices have their own drawbacks, one such drawback being that they are proprietary in design.

2.1.3 Interfacing with a Computer

Communication with the host machine must be completed for the HCI device to work. Most modern HCI devices use universal serial bus in order to communicate with a computer. This report will investigate using USB as the communication bus between an MCU and the host computer for this functionality. Modern operating systems include support for general input devices. Windows allows for input devices to be designed using custom drivers or general human interface device communication over USB like described in Silicon Labs tutorial on the design of HIDs. [3] This tutorial is specifically for Silicon Labs devices, but the concepts apply to all USB HID devices.

Another potential resource for the design of communication between the device and host computer is with the use of Valve Corporation's 'Open VR' SDK. [4] This SDK include repositories to allow the design of VR controllers to work with steam VR. This would allow the device to communicate 3D information to the computer however it would also limit the device to only work within the 'Steam VR' ecosystem.

2.2 Real Time Operating System (RTOS) Embedded System

2.2.1 Real Time Operating Systems

A real time operating system refers to operating system software that has a predetermined amount of clock cycles for all instructions to take place. It is designed as such to allow the device to have fast, highly accurate, and efficient performance for time sensitive operations. The use of a RTOS for a HCI is beneficial as it allows for the fast performance which is important as when a user is interacting with a virtual environment any latencies in the users action and the action occurring on the computer are unwanted and users can likely detect changes in latency as low as 33msec according to a paper by Stephen R. Ellis, Mark J. Young, Bernard D. Adelstein, and Sheryl M. Ehrlich. [5]

2.2.2 Embedded System

An embedded system is what shall compute the 3D data and communicate it to the host computer. It is therefore important that the embedded system used is one capable of communication with a computer, computation of the 3D data and one that operates using a real-time operating system. The Texas Instrument Simplink cc2650 meets these requirements and was opted as the platform for this project. [6] This development board is capable of several different methods of wireless communication, as well as having a 48MHz processor which is more than capable of the 3D data computation needed. It has an array of GPIO pins to allow integration of many methods of 3D data collection. The cc2650 also has support for TI's real time operating system software TI-RTOS.

2.3 Three-Dimensional Data

2.3.1 3D Rotational Data

Tracking and recording of 3D data onboard a device is most often completed using one or more of the following three technologies, a gyroscope, accelerometer or a magnetometer. Each varies in operation, but all can be used as a method of computing 3D rotational data. A gyroscope is capable of calculating rotational velocity accurately, as it can sense motion relative to that of gravity. This allows for accurate rotational motion data but is susceptible to drift as the device has no way to calculate current rotational position. An accelerometer works via sensing acceleration along a single axis; however three linear accelerometers can be used orientated perpendicular to each other to calculate acceleration in an x, y and z direction and then calculate based off that the orientation of the device against the acceleration due to gravity on the accelerometers. A magnetometer simply senses the direction of a magnetic field and can take advantage of the planets natural magnetic field to calculate the devices rotation in reference to the magnetic north pole. This method is limited and susceptible to magnetic interference. Therefore, this method should most often be incorporated along with one or both of the previous two.

2.3.2 3D Positional Data

As well as rotational data of the device, the devices position in 3D space could also be collected and communicated to the host computer. There are various methods this data could be collected including the methods used by Nintendo's 'Wii' and Microsoft's 'Kinect' discussed earlier. Another possible method would be to use distance sensors onboard the device to calculate distances from know objects such as the ground. This method would allow the device to remain a self-contained device without the need of peripherals such as cameras but would be extremely susceptible to interference from other objects. Tracking the 3D position of the device is an interesting area to investigate but will remain a lower priority than the rotational data for the purpose of this report.

2.4 Wireless Communication

2.4.1 Wireless versus Wired Communication

To communicate with the host PC the device must have some form of connection. The device could connect to the host pc via a wired connection such as USB, or communicate wirelessly. The advantages of wired communication is that it can be simpler and more reliable however it also can become cumbersome. In this scenario where the device shall need to move freely within 3D space a cable would be very inconvenient and cumbersome and therefore this report will investigate the use of wireless communication for this device.

2.4.2 Current Standards

For the device to be functional it will need to be able to wirelessly transfer data to the computer. The method of transportation will have a major impact on the responsiveness of the device. As this device will be used with the computer it is communicating with at all times, the communication will therefore only ever be over a short distance. There are many new communication methods being standardised for short range communication wirelessly. These personal area networks (PANs) can be very fast and low powered ways of sending data point to point. Some of these methods of wireless communication are described within an article by Cheolhee Park and Theodore S. Rappaport. [7] The SimpleLink device described earlier are capable of many of these communication standards including Wi-Fi, Bluetooth, as well as others described on the ti website. [8] One interesting capability of this device is an extremely low powered and low overhead device to device communication between two of these SimpleLink devices. This could allow for one device to track and send 3D data and the other to receive the data and interface with the host computer over a wired connection such as USB. This could drastically simplify the communication with the computer.

Chapter 3 - Design of the Wireless 3D Embedded RTOS HCI

3.1 Parts Choice

3.1.1 MCU

The Texas Instruments cc2650 Launchpad was chosen as the micro controller for the device. For its wireless communication capabilities, adequate computational performance and user accessible to GPIO. Two of these are deployed in a transceiver, receiver pair, with the transceiver interfacing with the accelerometer and sending its data to the receiver. The receiver takes this data and then communicates it to the host computer as a HID instruction.

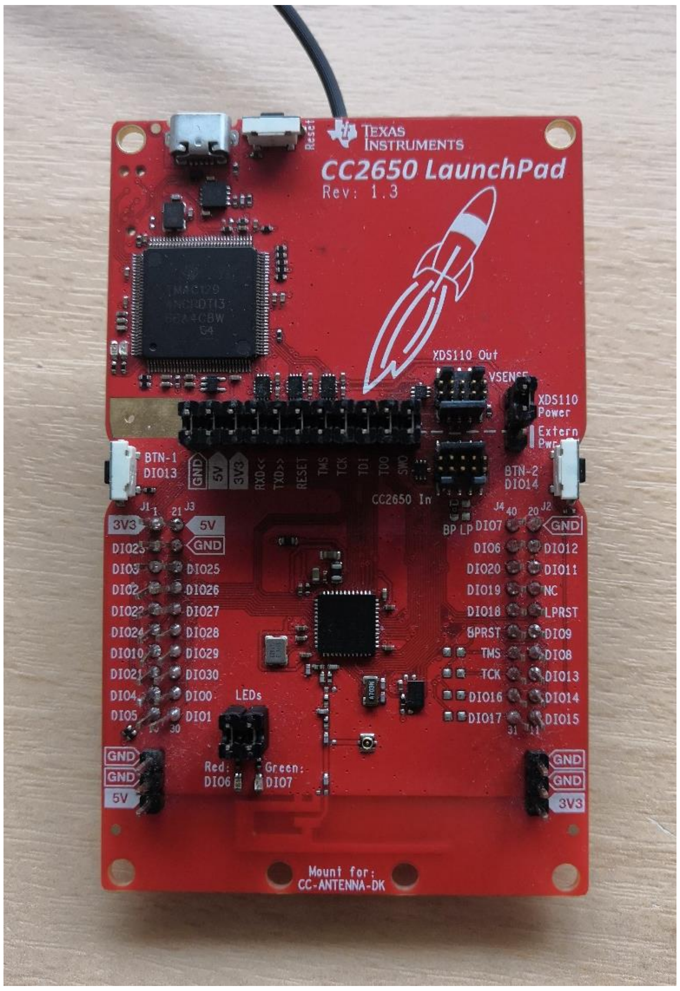


Figure 1 CC2650 LaunchPad MCU

3.1.2 Accelerometer/Gyroscope

For the gyroscope/accelerometer an ADXL345 was chosen. This device was chosen for its ability to communicate using either I2C or SPI interfaces, its ability to accurately measure across 3 axes up to a resolution of 13bits at ±16g, with lower resolution modes ±2g, ±4g, and ±8g, and its low typical power draw of 0.462mW. [9]

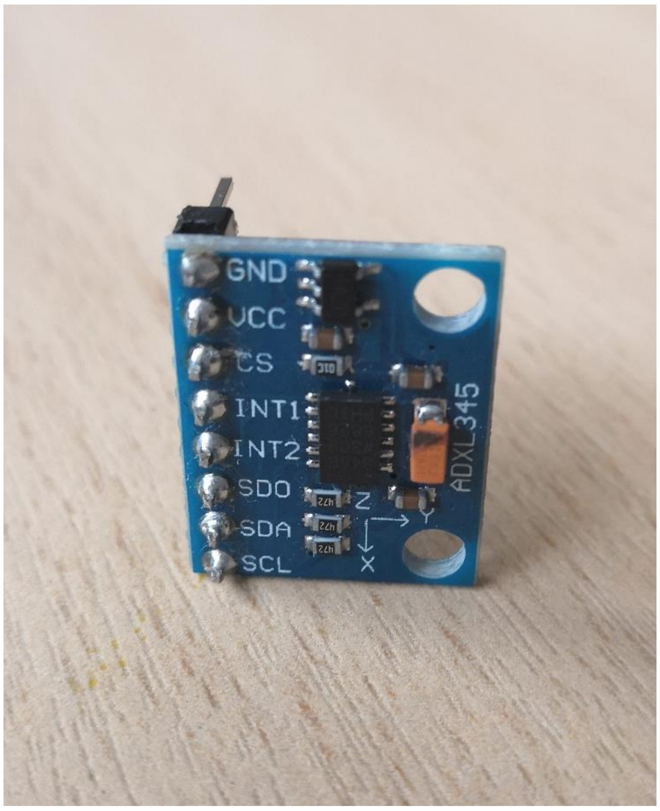


Figure 2 ADXL345 Accelerometer

3.2 TI-RTOS

A major element of this project is working within the real time operating system environment. It is therefore important to understand the concepts of how the specific environment the device will be designed in operates. TI-RTOS is the operating system that is to be used. Its syntax bares some similarities to C and C++ but differs substantially in operation.

3.2.1 Task Construction

TI-RTOS works on setting up threaded operations and then allowing the device bios to schedule and run each thread. The threads are constructed as tasks on initialisation of the device. The user must give the task parameters and functionality. Initialize the task and once all tasks and functionality of each task is defined the BIOS runs them all. Tasks must have a defined stack and stack size i.e. the area on the device the instructions will be stored, as well as a priority. This priority will determine the processing time that the bios grants the task. The tasks functionality must also be defined, in a similar way to a function definition in C or C++ and the task is constructed to point towards it.

```
Task_Params_init(&taskParams);
taskParams.stack = notifyTaskStack;
taskParams.stackSize = NOTIFY_TASK_STACK_SIZE;
taskParams.priority = NOTIFY_TASK_PRIORITY;

Task_construct(&notifyTask, NotificationTask_taskFxn, &taskParams, NULL);
```

[10]

Once a task is defined and created, once the command BIOS_start() is ran the threads will begin.

3.2.2 Pin Selection

Pin selection is another important element of TI-RTOS to understand in the design of this device. In order to control the pins of the cc2650, they are first declared on a pin config table this table. Then a pin handle is opened linked to the pin config table, and the pins are ‘opened’. This allows the user to now control the pins as they see fit. From that point pins can be controlled by referencing the pin table, the individual pin and then setting the pin value.

```
static PIN_Handle MyPinHandle;
static PIN_State MyPinState;
PIN_Config MyPinTable[] = {
    Board_LED0 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    Board_LED1 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    PIN_TERMINATE
};

/* Open LED pins */
MyPinHandle = PIN_open(&MyPinState, MyPinTable);
PIN_setOutputValue(MyPinHandle, Board_LED1, 1);
PIN_setOutputValue(MyPinHandle, Board_LED0, 0);
```

3.2.3 TI-RTOS Initial Programming Test

With these capabilities it is possible to test the TI-RTOS environment with a program similar to that of a hello world, flashing LEDs. This simple application is the first proof of concept of use of the TI-RTOS. A program was written to create a task to control the LEDs on the board which is available in the appendix at the end of this report.

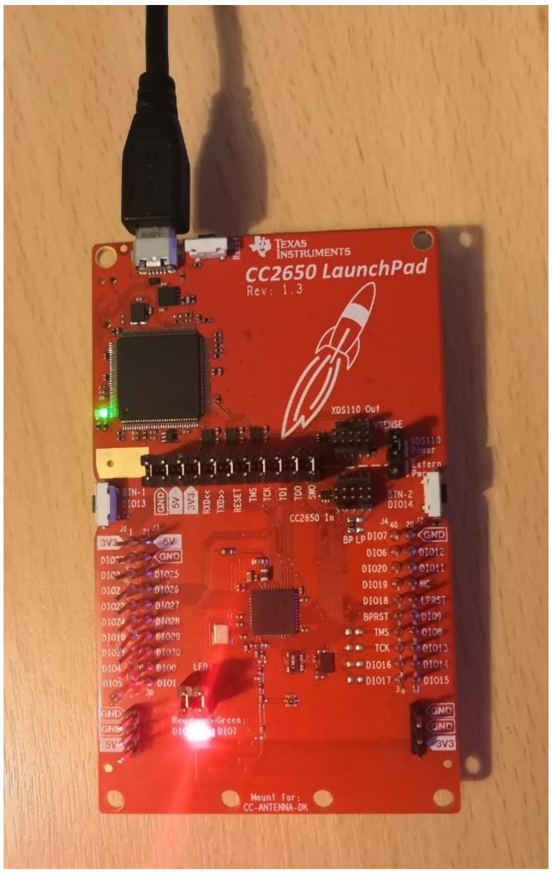


Figure 3 Initial TI-RTOS Test

3.2.4 TI-RTOS Pre-Supplied Libraries

Texas Instruments have several libraries wrote for TI-RTOS that are relevant to this project. There are libraries available for SPI and I2C communication built to simplify interfacing with other devices, SPI.h and I2C.h allow the user to create connections and make data transactions with the device. [11] [12] This should simplify communication with the ADXL345 possible and therefore the functionality of the libraries was studied.

3.3 Interfacing with the Accelerometer

The MCU must communicate with the ADXL345 to collect the necessary 3D data. There are two potential communication standards that both the CC2650 and the ADXL345 support, I2C, and SPI. Both possible designs have been detailed below.

3.3.1 Connecting SPI Device

The first design for the communication between the MCU and the ADXL345 is with the use of the SPI Interface. The SPI Interface uses a connection of a clock signal, two data lines, a master out slave in (MOSI) and master in slave out (MISO) as well as a chip select line to activate the slave for communication. The ADXL345 acts as a slave device with the CC2650 acting as the master and controlling communication between them both. The ADXL345 also has two active high interrupt pins as well as a standard supply voltage and ground pin.

The CC2650 can use any of the GPIO to serve for the communication but have pins pre-defined by the SPI library also so these will be the ones used in this design.

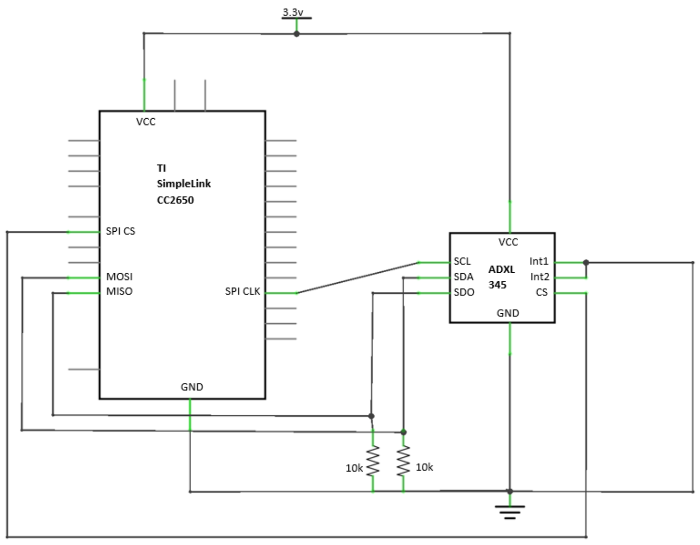


Figure 4 SPI Interface Schematic

The advantages of SPI over I2C is quicker communication as it requires less data transections due to the fact that I2C does not use a chip select line and instead transmits a slave devices ID to begin communication. Its disadvantages include the need for more GPIO as each device requires its own chip select line, and it has a more complex such as setting the phase and polarity of the clock, whereas in I2C these parameters are defined.

3.3.2 Connecting I2C Device

Interfacing with the ADXL345 using I2C uses a clock signal and a single data line. The master device initiates communication via sending the slave devices ID onto the communication line, then the slave acknowledges, and the master continues the communication. For the ADXL345 to have I2C mode activate the chip select line must be high and therefore in this design it is connected to Vcc, also the data lines are pulled high to ensure no floating connections, and the interrupt pins are again connected to ground. The SDO line acts as an alt address controller for I2C so that there are two possible address the device may use based on whether the alt address pin is high or low, so in this design it is also grounded.

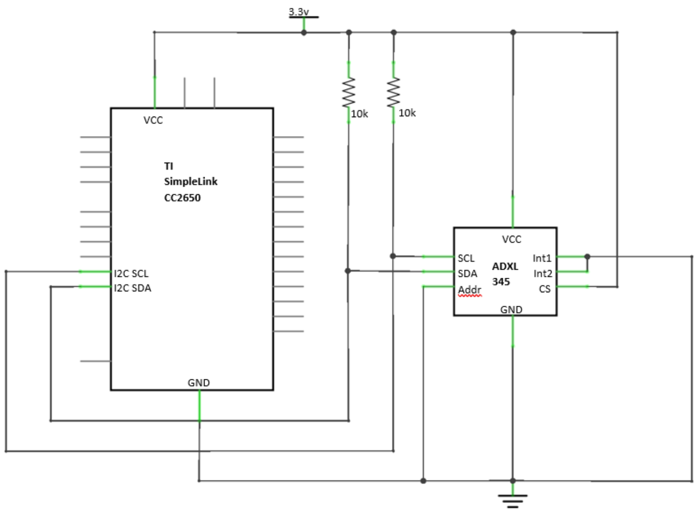


Figure 5 I2C Interface Schematic

Here we can see how the ADXL345 in I2C mode needs less of the GPIO of the MCU to function. Also, if there were more devices to be added they could all communicate over the same two GPIO pins as long as the slave device addresses do not conflict.

Another consideration with the ADXL345 is how the I2C mode is activated via the chip select pin being high. Therefore, when using the device in SPI mode there is the potential for a communication on the data line to be considered an I2C communication. The solution to this is described in the data sheet to simply add an OR gate with the chip select and data line before the SDA line of the ADXL345. [9]

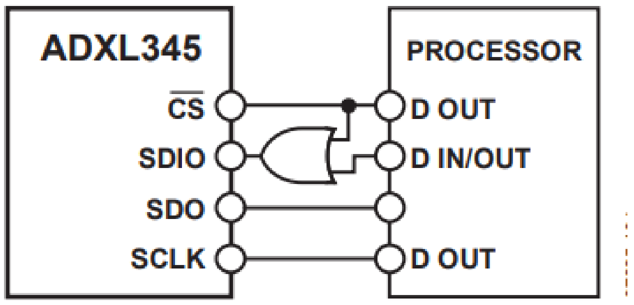


Figure 6 SPI CS Error Solution

3.3.3 Programming for SPI

SPI communication is done in TI-RTOS via first declaring an SPI Handle object and an SPI Parameters object. These objects handle the connection and store parameter settings respectively. The parameters include the transfer mode, frame format, SPI mode, bitrate, and data size. Once the parameters are set the connection can be opened and set as the handle. Then a transaction object is created with its own parameters for each transaction, including the transmit and receive buffers and the count of data chunks to send. So, a simple SPI transfer is wrote as below:

```
SPI_Params_init(&spiParams); //initialise spi params object
spiParams.transferMode = SPI_MODE_BLOCKING; //block thread until transaction complete
spiParams.transferTimeout = SPI_WAIT_FOREVER;//wait forever
spiParams.frameFormat = SPI_POL1_PHA1;//clock polarity 1 phase 1
spiParams.transferCallbackFxn = NULL; //no callback
spiParams.mode = SPI_MASTER; //master mode
spiParams.bitRate = 400000; //400MHz
spiParams.dataSize = 8; //data size
handle = SPI_open(0, &spiParams);//open spiHandle

transmitBuffer[0] = 0xAD;//write to power control
transmitBuffer[1] = 0x08;//turn device on

spiTransaction.count = 2;
spiTransaction.txBuf = transmitBuffer;
spiTransaction.rxBuf = recieveBuffer;

ret = SPI_transfer(handle, &spiTransaction);
if (!ret) {
    System_printf("Unsuccessful SPI transfer");
}
```

This full program code is available in the appendix at the end of this report.

For this device once, the connection is initialized a threads operation should be to constantly update x, y, and z data and a separate thread handles sending this data to the receiver. The X, Y, and Z data is accessible via their own SPI transactions to the registries 0x32 to 0x37. Each coordinate uses two bytes so multi-byte reads are needed for each.

3.3.4 Programming for I2C

The programming for I2C communication is very similar to that of the SPI interface. A connection handler object and parameters objects are made in the same way. The paramaters include slightly different data as clock phase and polarity does not need to be set. Then the transaction is given perameters in a similar way however this time must have a slave address specified.

```
    I2C_Params_init(&params);
    params.transferMode = I2C_MODE_BLOCKING; //thread waits for transaction
    params.transferCallbackFxn = NULL; //no callback needed
    params.bitRate = I2C_400kHz ; //400kHz

    txBuffer[0] = 0;//Get Device ID

    i2cTrans.writeCount   = 1; //one byte right (slave address is handled by I2C.H
    i2cTrans.writeBuf     = txBuffer;
    i2cTrans.readCount    = 1;
    i2cTrans.readBuf      = rxBuffer;
    i2cTrans.slaveAddress = 0x3A;//slave address when alt address pin is grounded

    handle = I2C_open(Board_I2C, &params);//open connection
    I2C_transfer(handle, &i2cTrans);//make transfer

    System_printf("testing %d \n", rxBuffer[0]);
    System_flush();
    I2C_close(handle);
```

Apart from these slight differences the data is then collected identically as stated in the SPI design. Once the thread begins collecting data the XYZ Data can be sent by the transmission thread to the receiver device.

3.3.5 Data Transmission

Once the data is received from the ADXL345 it must be sent wirelessly to the receiver. This runs in a separate thread that sends the XYZ data to the receiver continuously. It would be more energy efficient to send the data only when there is a change, but this would add additional latency and complexity and therefore instead the data will be sent at a constant rate and changes in the position will be calculated on the receiver's end.

TI have sample code for use with the CC2650 on sending packets and therefore this code will be tested and then deployed with the rest of the program and added as necessary. This code is available from the TI-RTOS resource files and also made available in the appendix of this report. [13]

3.4 Receiver Device

The receiver device is another CC2650, its operation is to receive the XYZ data from the transmitter, once received it will calculate any movement via a difference in the values to those previously received. Once calculated this difference will then be communicated to a host computer as movement of a human interface device (HID). One thread will receive the data and calculate the change in data as it is ingested, then a separate thread will communicate this to the computer. This will allow for one thread to be near constantly be waiting on updates from the transceiver as the other thread operates the interactions with the computer.

3.4.1 Receiving Data

Much like the data transmission, TI also have made available sample code for the recovery of data from a CC2650 to another, therefore this code will again be deployed tested and edited to suit this application. The code is available again from the TI Resource repositories for TI-RTOS and the CC2650 section specifically and again is displayed in the appendix of this report. [13]

3.4.2 Host PC Interfacing

Once the data is received and the change in data is calculated this can be communicated to the computer via a serial connection. This should use the HID standard and begin by communicating the parameters pf the device. For this device this would be the three axis or orientation. There range is given based on the resolution of values that will be calculated by the receiver device. Once the parameters are communicated and the device is initialised the device will update position via HID inputs every time the position of the transmission device changes.

Chapter 4- Implementation and Testing of Wireless 3D Embedded RTOS HCI

4.1 Transmission Device

4.1.1 Wiring of Accelerometer SPI

The initial task was the connect the ADXL345 to the CC2650 in order to accommodate communication. The SPI library on TI-RTOS can have all pins defined but also has default pins and these were elected to be used. The device for the time being will be powered via the USB connection and therefore the ADXL345 will be powered via the 3.3v pin on the CC2650. These default pins can be viewed on the CC2650 LuanchPad Quick Start Guide. [14]

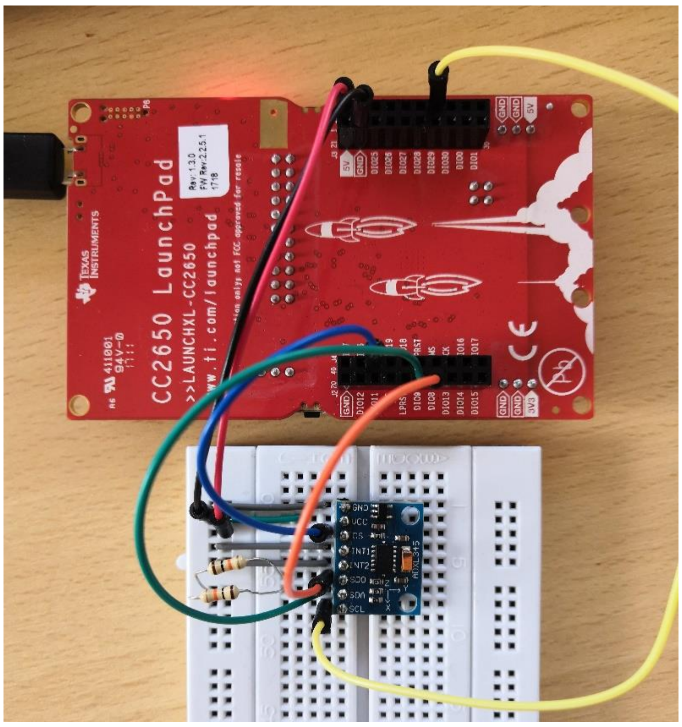


Figure 7 Wiring of SPI Interface

4.1.2 Programming of SPI Communication

Once connected the programming of the SPI communication could take place. Initially the SPI communication was run in the initialization section of the program to turn on and initialize the ADXL345. The initial task is to retrieve the device ID to ensure the communication is working. Below is an example of the initial SPI code written for the device:

```
Int main(){
  System_printf("Start Setup\n");
  Board_initGeneral();
  Board_initSPI();//initialize Board SPI
  ADXLInit();//turn on adxl345 get devID

  /* Configure task. */
  Task_Params taskParams;
  Task_Params_init(&taskParams);
  taskParams.stack = myTaskStack;
  taskParams.stackSize = sizeof(myTaskStack);
  Task_construct(&myTaskStruct, SPIFxn, &taskParams, NULL);

  BIOS_start();
  return (0);
}
ADXLInit(){
    System_printf("Start SPI task\n");
    SPI_Handle handle;
    SPI_Params spiParams;
    SPI_Transaction spiTransaction;
    uint8_t transmitBuffer[2];
    uint8_t recieveBuffer[2];
    uint8_t devid;
    bool ret;

    SPI_Params_init(&spiParams); //initialise spi params object
    spiParams.transferMode = SPI_MODE_BLOCKING; //block thread until transaction complete
    spiParams.transferTimeout = SPI_WAIT_FOREVER;//wait forever
    spiParams.frameFormat = SPI_POL1_PHA1;//clock polarity 1 phase 1
    spiParams.mode = SPI_MASTER; //master mode
    spiParams.bitRate = 400000; //400MHz
    spiParams.dataSize = 8; //data size
    handle = SPI_open(0, &spiParams);//open spiHandle

    transmitBuffer[0] = 0xAD;//write to power control
    transmitBuffer[1] = 0x08;//turn device on

    spiTransaction.count = 2;
    spiTransaction.txBuf = transmitBuffer;
    spiTransaction.rxBuf = recieveBuffer;
    ret = SPI_transfer(handle, &spiTransaction);
    if (!ret) {
        System_printf("Unsuccessful SPI transfer");
    }
    transmitBuffer[0] = 0x80;//read from address 0x00 with read bit 0x80 (devID)
    transmitBuffer[1] = 0x00;//fake request for interchange
    ret = SPI_transfer(handle, &spiTransaction);
    if (!ret) {
          System_printf("Unsuccessful SPI transfer");
      }
    System_printf("Devid = %D", recieveBuffer[1]);
    System_flush();
}
```

From that point it can continue to create a task that continuously updates the X, Y, and Z values from the ADXL345. The X, Y, and Z data will be global variables to allow for the transmission data to access it. Major issues with this code was met and is described further in the section 4.3 of this report.

4.1.3 Wiring of Accelerometer I2C

With the issues met using the SPI interface for communicating with the ADXL345 the decision was made to start experimenting with communicating over I2C. The wiring of the ADXL345 to communicate over I2C was not drastically different as TI also have default pins defined for use with the I2C.h library. Again, made available in the CC2650 LaunchPad QuickStart guide. [14]

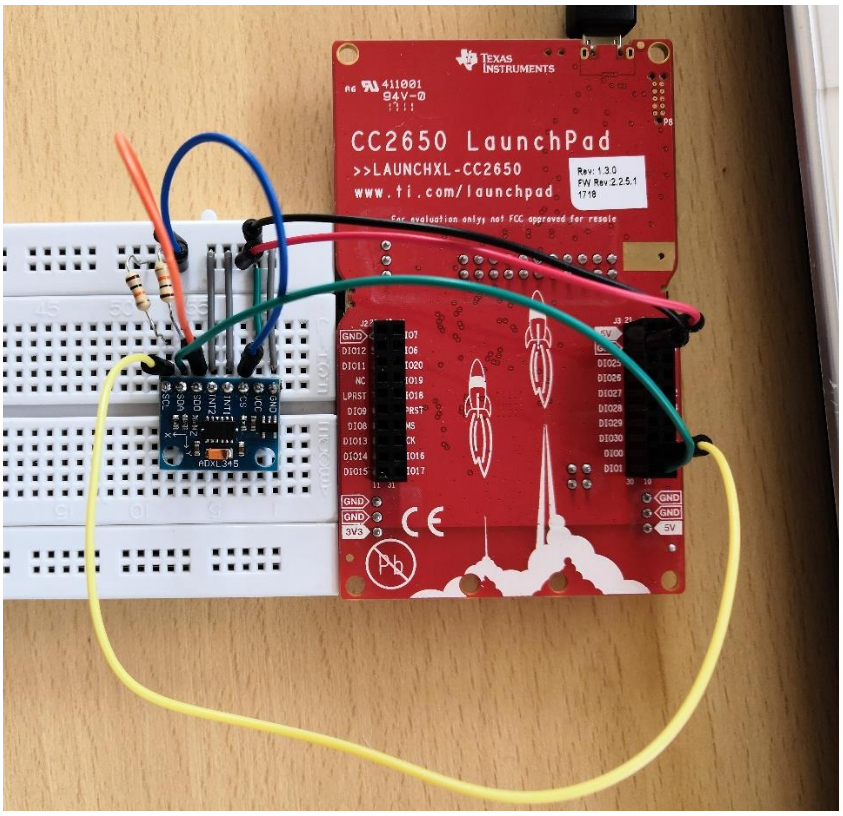


Figure 8 I2C Interface Wiring

4.1.4 Programming of I2C Communication

I2C code was developed in a similar way to the SPI code to work within the same workflow of first initializing the ADXL345's Power control registry to begin measurement and to then continue by retrieving the current X, Y, and Z data from the ADXL345. Again, this data would be stored in global variables to allow easy access to the transmission thread. Below is an extract showing th initialization code for the I2C communication with the ADXL345.

```
static void ADXLInit()
{
    I2C_Handle handle;
    I2C_Params params;
    I2C_Transaction i2cTrans;

    I2C_Params_init(&params);
    params.transferMode = I2C_MODE_BLOCKING; //thread waits for transaction
    params.bitRate = I2C_400kHz ; //400kHz

    txBuffer[0] = 0;//Get Device ID

    i2cTrans.writeCount   = 1; //one byte right (slave address is handled by I2C.H)
    i2cTrans.writeBuf     = txBuffer;
    i2cTrans.readCount    = 1;
    i2cTrans.readBuf      = rxBuffer;
    i2cTrans.slaveAddress = 0x3A;//slave address when alt address pin is grounded

    handle = I2C_open(Board_I2C, &params);//open connection
    I2C_transfer(handle, &i2cTrans);//make transfer

    System_printf("testing %d \n", rxBuffer[0]);
    System_flush();
    I2C_close(handle);
}


int main(void)
{
    Task_Params taskParams;
    /* Call board init functions */
    Board_initGeneral();
    I2C_init();
    ADXLInit();//initialize I2C comm

    Task_Handle Task1
    Task_Params_init(&taskParams);
    taskParams.arg0 = 1000000 / Clock_tickPeriod;
    taskParams.stackSize = 512;
    taskParams.stack = task0Stack;
    taskParams.priority = 3;
    Task1 = Task_create((Task_FuncPtr)I2CFxn, &taskParams, NULL);
    /* Start BIOS */
    BIOS_start();
    return (0);
}
```

4.1.5 Transmission of Data

Once the XYZ data is collected the transmission code, which is to run within its own thread can begin sending the data to the receiver device. This can be done via simply editing the available TI data transmission code to send each of the variables on at a time. There is a possibility for the data to in the global variable to be incorrect due to a race condition from both threads accessing the global variable, however as this project is just investigating the concept of these devices this is an issue that can be tackled at a later date, and may be better suited for the communication thread to have a function to retrieve the data from the ADXL345 communication thread. Below is the initial code for sending the X value of the accelerometer:

```
static void txTaskFunction(UArg arg0, UArg arg1)
{
    uint32_t time;
    RF_Params rfParams;
    RF_Params_init(&rfParams);

    RF_cmdPropTx.pktLen = PAYLOAD_LENGTH;
    RF_cmdPropTx.pPkt = packet;
    RF_cmdPropTx.startTrigger.triggerType = TRIG_ABSTIME;
    RF_cmdPropTx.startTrigger.pastTrig = 1;
    RF_cmdPropTx.startTime = 0;

    /* Request access to the radio */
    rfHandle = RF_open(&rfObject, &RF_prop, (RF_RadioSetup*)&RF_cmdPropRadioDivSetup, &rfParams);

    /* Set the frequency */
    RF_postCmd(rfHandle, (RF_Op*)&RF_cmdFs, RF_PriorityNormal, NULL, 0);

    /* Get current time */
    time = RF_getCurrentTime();
    while(1)
    {
        /* Create packet with incrementing sequence number and random payload */
        packet[0] = 1; //first com used to determine x y or z (1 for x)
        packet[1] = x0;
        packet[2] = x1;

        /* Set absolute TX time to utilize automatic power management */
        time += PACKET_INTERVAL;
        RF_cmdPropTx.startTime = time;

        /* Send packet */
        RF_EventMask result = RF_runCmd(rfHandle, (RF_Op*)&RF_cmdPropTx, RF_PriorityNormal, NULL, 0);
        PIN_setOutputValue(pinHandle, Board_LED0,!PIN_getOutputValue(Board_LED0));
    }
}
```

4.2 Receiver Device

4.2.1 Receiver Programming

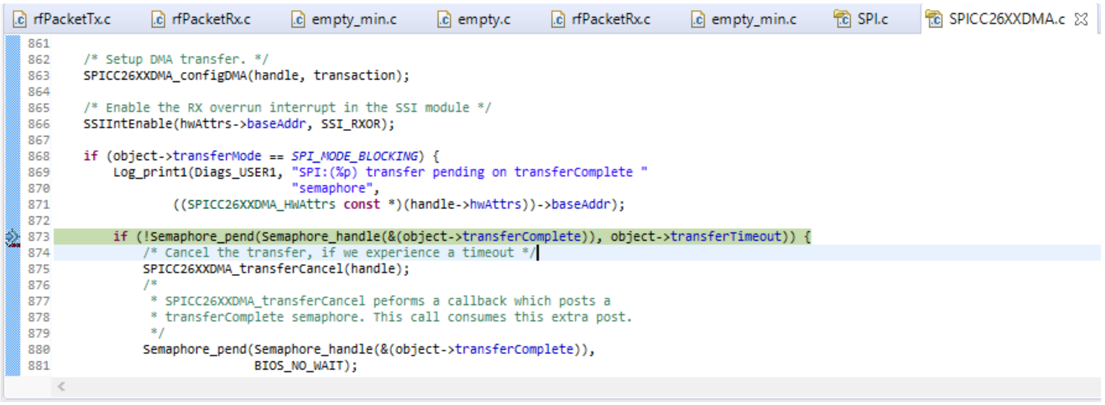
Once the transmitter device sends the data the receiver must receive and store this data. Again, the provided code from TI was used as a base to be edited to retrieve the data from the transmission device. Once receiver this data was compared to the data previously received to calculate a change in orientation of the transmitter. This change motion is then stored again as a global variable for the thread to communicate with the host PC to access. Below is the initial code for accepting the data. Again, built upon from the provided TI communication code:

```
static void rxTaskFunction(UArg arg0, UArg arg1)
{
    RF_Params rfParams;
    RF_Params_init(&rfParams);

    if( RFQueue_defineQueue(&dataQueue,
                            rxDataEntryBuffer,
                            sizeof(rxDataEntryBuffer),
                            NUM_DATA_ENTRIES,
                            MAX_LENGTH + NUM_APPENDED_BYTES))
    {
        /* Failed to allocate space for all data entries */
        while(1);
    }

    /* Modify CMD_PROP_RX command for application needs */
    RF_cmdPropRx.pQueue = &dataQueue;           /* Set the Data Entity queue for received data */
    RF_cmdPropRx.rxConf.bAutoFlushIgnored = 1;  /* Discard ignored packets from Rx queue */
    RF_cmdPropRx.rxConf.bAutoFlushCrcErr = 1;   /* Discard packets with CRC error from Rx queue */
    RF_cmdPropRx.maxPktLen = MAX_LENGTH;        /* Implement packet length filtering to avoid
PROP_ERROR_RXBUF */
    RF_cmdPropRx.pktConf.bRepeatOk = 1;
    RF_cmdPropRx.pktConf.bRepeatNok = 1;

    /* Request access to the radio */
    rfHandle = RF_open(&rfObject, &RF_prop, (RF_RadioSetup*)&RF_cmdPropRadioDivSetup, &rfParams);
    /* Set the frequency */
    RF_postCmd(rfHandle, (RF_Op*)&RF_cmdFs, RF_PriorityNormal, NULL, 0);
    /* Enter RX mode and stay forever in RX */
    RF_runCmd(rfHandle, (RF_Op*)&RF_cmdPropRx, RF_PriorityNormal, &callback, IRQ_RX_ENTRY_DONE);

    while(1);
}

void callback(RF_Handle h, RF_CmdHandle ch, RF_EventMask e)
{
    if (e & RF_EventRxEntryDone)
    {
        /* Toggle pin to indicate RX */
        PIN_setOutputValue(pinHandle, Board_LED1,!PIN_getOutputValue(Board_LED1));
        /* Get current unhandled data entry */
        currentDataEntry = RFQueue_getDataEntry();

        /* Handle the packet data, located at &currentDataEntry->data:
         * - Length is the first byte with the current configuration
         * - Data starts from the second byte */
        packetLength      = *(uint8_t*)(&currentDataEntry->data);
        packetDataPointer = (uint8_t*)(&currentDataEntry->data + 1);
        /* Copy the payload + the status byte to the packet variable */
        memcpy(packet, packetDataPointer, (packetLength + 1));
        RFQueue_nextEntry();
    }
}
```

4.3 Further Implementation and Testing

4.3.1 SPI Issues and Resolution

After the initial code to communicate between the CC2650 and ADXL345 was written it did not perform as expected. There was no response from the ADXL345. Upon further investigation it was found that the MCU was entering a state of waiting forever during the communication. The spi handle was opening correctly but on transfer the program stalled. The culprit was found to be Semaphore_pend() function shown here:

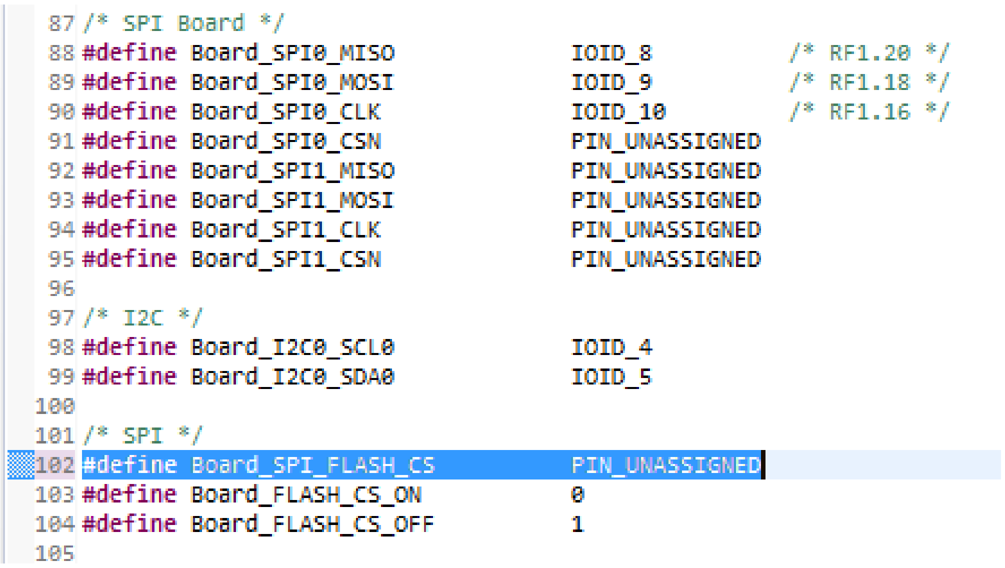


```

    /* Setup DMA transfer. */
    SPICC26XXDMA_configDMA(handle, transaction);

    /* Enable the RX overrun interrupt in the SSI module */
    SSIIntEnable(hwAttrs->baseAddr, SSI_RXOR);

    if (object->transferMode == SPI_MODE_BLOCKING) {
        Log_print1(Diags_USER1, "SPI:(%p) transfer pending on transferComplete "
                                "semaphore",
                ((SPICC26XXDMA_HWAttrs const *)(handle->hwAttrs))->baseAddr);

        if (!Semaphore_pend(Semaphore_handle(&(object->transferComplete)), object->transferTimeout)) {
            /* Cancel the transfer, if we experience a timeout */
            SPICC26XXDMA_transferCancel(handle);
            /*
             * SPICC26XXDMA_transferCancel peforms a callback which posts a
             * transferComplete semaphore. This call consumes this extra post.
             */
            Semaphore_pend(Semaphore_handle(&(object->transferComplete)),
                           BIOS_NO_WAIT);
```

Figure 9 SPI Semaphore_Pend() Breakpoint

This function had the bios wait on the transaction to complete, however due to the fact that this code was being ran during the initialization stage of the program before BIOS_begin() this function did not work and the program would stall here. So, a fix for this is to add is to a task to commence after the bios has begun.

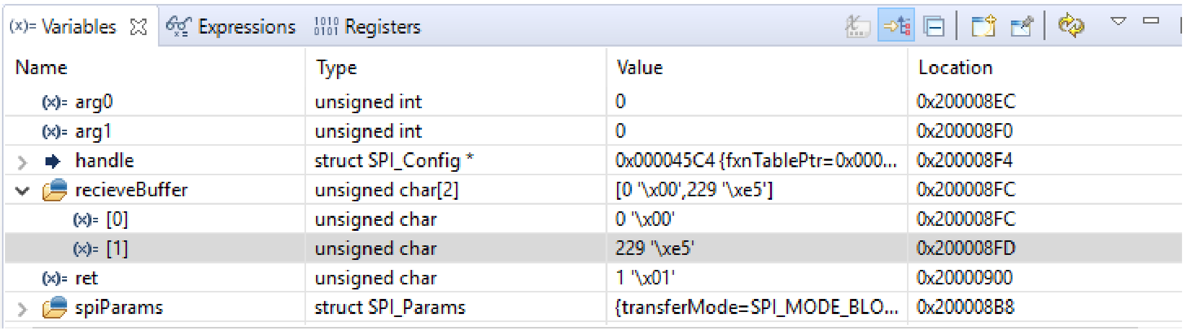
Once this was fixed there was CC2350 was reporting a return from the slave device, however on transfer the data received always remained 0x00. So while the transaction was being carried out on the CC2650s end the ADXL345 did not seem to be responding correctly. Further investigation showed that in the CC2650 boards header files the CS pin for SPI was not defined.



```
/* SPI Board */
#define Board_SPI0_MISO             IOID_8          /* RF1.20 */
#define Board_SPI0_MOSI             IOID_9          /* RF1.18 */
#define Board_SPI0_CLK              IOID_10         /* RF1.16 */
#define Board_SPI0_CSN              PIN_UNASSIGNED
#define Board_SPI1_MISO             PIN_UNASSIGNED
#define Board_SPI1_MOSI             PIN_UNASSIGNED
#define Board_SPI1_CLK              PIN_UNASSIGNED
#define Board_SPI1_CSN              PIN_UNASSIGNED

/* I2C */
#define Board_I2C0_SCL0             IOID_4
#define Board_I2C0_SDA0             IOID_5

/* SPI */
#define Board_SPI_FLASH_CS          PIN_UNASSIGNED
#define Board_FLASH_CS_ON           0
#define Board_FLASH_CS_OFF          1
```

Figure 10 CS Pin Assignment

With this pin assigned a value, the ADXL345 was still not returning data on transfer, to see if the chip select pin was still the issue the CS line of the ADXL345 was grounded as the program was ran. This time the correct value of the device ID (11100101 or 229 in decimal) was returned to the CC2650 determining that it was the chip select line causing the issue.



Name	Type	Value	Location
arg0	unsigned int	0	0x200008EC
arg1	unsigned int	0	0x200008F0
handle	struct SPI_Config *	0x000045C4 {fxnTablePtr=0x000...	0x200008F4
recieveBuffer	unsigned char[2]	[0 '\x00',229 '\xe5']	0x200008FC
[0]	unsigned char	0 '\x00'	0x200008FC
[1]	unsigned char	229 '\xe5'	0x200008FD
ret	unsigned char	1 '\x01'	0x20000900
spiParams	struct SPI_Params	{transferMode=SPI_MODE_BLO...	0x200008B8

Figure 11 SPI Correct Return Data

To ensure that the device still had chip select functionality a separate pin was connected to the ADXL345 device and controlled by the program to turn low before a transfer and return to high once a transfer was complete. This allowed for the chip select functionality to still be used and had the functionality tested via an LED connected to the same pin.

```
PIN_setOutputValue(ledPinHandle, Board_LED1, 0); //turn chip select low
delay_ms(1);
ret = SPI_transfer(handle, &spiTransaction); //transfer data (device ID)
PIN_setOutputValue(ledPinHandle, Board_LED1, 1); //set chip select high
```

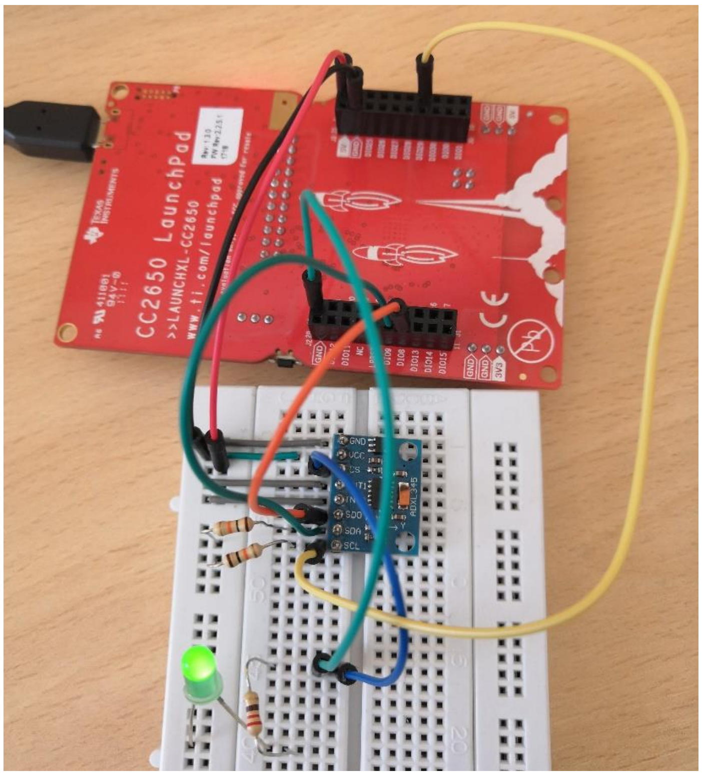
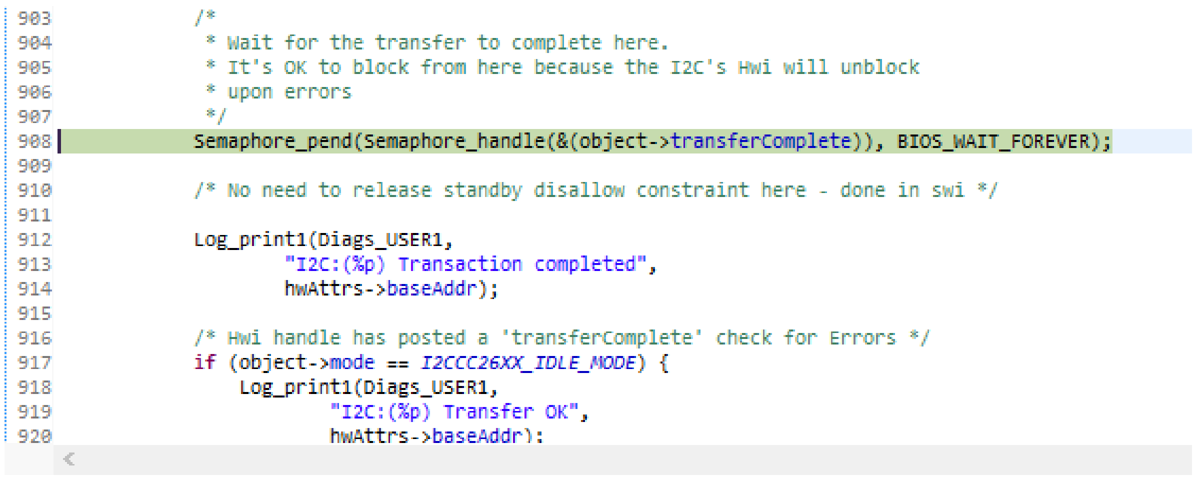


Figure 12 Custom Pin CS

4.3.2 I2C Issues

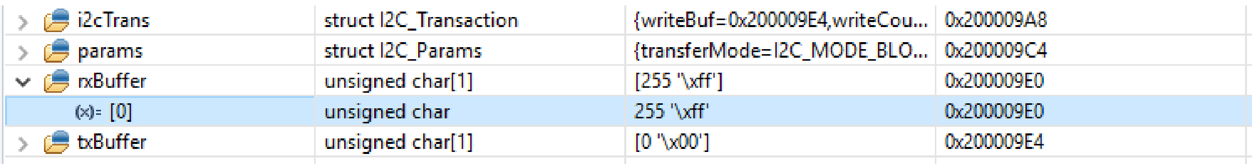
Similarly, to the SPI interface the I2C communication had some difficulties. Work was done on investigating these problems along side the work done to solve the SPI issues. To begin with the I2C communication was being halted on a similar Semaphore_pend() function. This meant that the functionality had to be moved to within a task.



```
        /*
         * Wait for the transfer to complete here.
         * It's OK to block from here because the I2C's Hwi will unblock
         * upon errors
         */
        Semaphore_pend(Semaphore_handle(&(object->transferComplete)), BIOS_WAIT_FOREVER);

        /* No need to release standby disallow constraint here - done in swi */

        Log_print1(Diags_USER1,
                "I2C:(%p) Transaction completed",
                hwAttrs->baseAddr);

        /* Hwi handle has posted a 'transferComplete' check for Errors */
        if (object->mode == I2CCC26XX_IDLE_MODE) {
            Log_print1(Diags_USER1,
                    "I2C:(%p) Transfer OK",
                    hwAttrs->baseAddr);
```

Figure 13 I2C Semaphor_pend() Breakpoint

With this correction made the program still ran into issues, as the value returned was 255 for each transaction, meaning the ADXL345 was not responding. Further investigation suggested that it may be due to how the device interprets its ID versus how the I2C.H file sends the address. The adxl345 expects its address followed by a read or write bit, Therefore the slave address appears as 0xA7 for a read rather than 0x53. Trying this technique, the device still did not appear to send any data back to the CC2650. As it was returning logic high for all transfers use of different pull-up resistor values and pull-down resistors was done to no avail.



> i2cTrans	struct I2C_Transaction	{writeBuf=0x200009E4,writeCou...	0x200009A8
> params	struct I2C_Params	{transferMode=I2C_MODE_BLO...	0x200009C4
∨ rxBuffer	unsigned char[1]	[255 '\xff']	0x200009E0
(x)= [0]	unsigned char	255 '\xff'	0x200009E0
> txBuffer	unsigned char[1]	[0 '\x00']	0x200009E4

Figure 14 I2C Return Data Error

At this point in time neither SPI or I2C communication were working so working with a separate chip was investigated to ensure the I2C.h functionality worked as expected. An MPU-6050 accelerometer was connected to test for I2C communication. Communication was successful with the MPU-6050, which determined that the I2C.h library worked as initially expected.

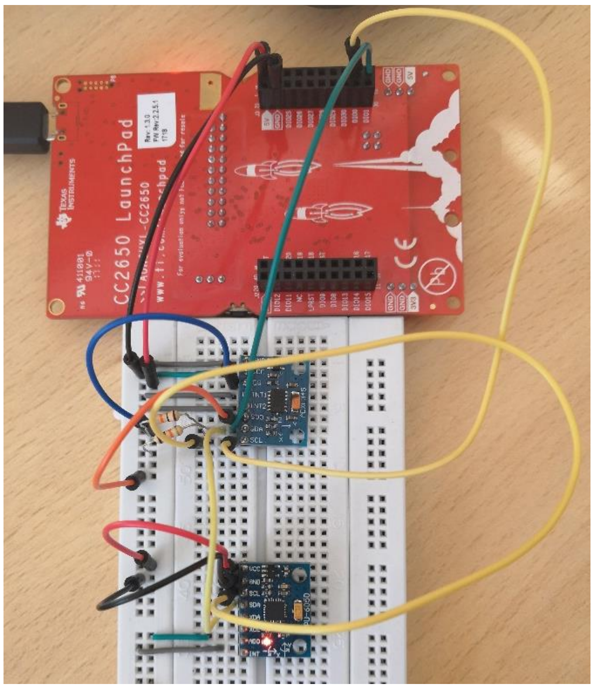
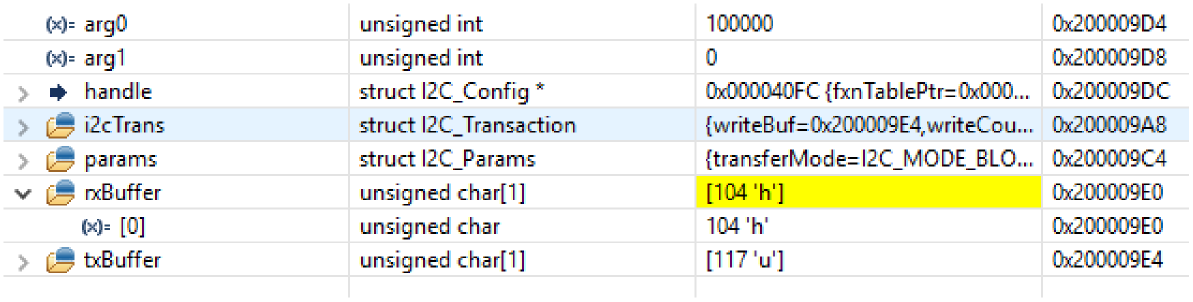


Figure 15 MPU I2C Wiring



(x)= arg0	unsigned int	100000	0x200009D4
(x)= arg1	unsigned int	0	0x200009D8
> handle	struct I2C_Config *	0x000040FC {fxnTablePtr=0x000...	0x200009DC
> i2cTrans	struct I2C_Transaction	{writeBuf=0x200009E4,writeCou...	0x200009A8
> params	struct I2C_Params	{transferMode=I2C_MODE_BLO...	0x200009C4
v rxBuffer	unsigned char[1]	[104 'h']	0x200009E0
(x)= [0]	unsigned char	104 'h'	0x200009E0
> txBuffer	unsigned char[1]	[117 'u']	0x200009E4

Figure 16 MPU I2C Correct Data

Returning to the ADXL345 for further investigation it was never discovered what was causing the I2C issues. It can only be assumed that the address value being sent by the code was in error in some way.

4.4 Final Testing

4.4.1 Wireless Transmission Testing

The first test undertaken was the range of the wireless communication. This was tested via having the transmitter device continuously transmit every 0.5 seconds and the receiver toggle an LED every time a packet is received, then the receiver was moved further and further from the transmitter in 5 metre increments. At each increment the amount of dropped packets over a 20 second period was noted. This was then repeated 4 times and results averaged.

Distance (m)	No. Dropped Packets	Percentage Data Dropped
5	0	0%
10	0	0%
15	3.5	8.75%
20	16	40%

Figure 17 Distance Testing Graph

As the device is most likely to be used in close proximity these results deem that the range is adequate however if the device is to be used in an application further from the receiver a more powerful antenna may need to be connected to the devices.

Then the effect of materials was investigated. First the transmitter was placed within a plastic container (High Density Polyethylene) to see if the connection was interrupted. The Polyethylene had no effect on the connection. Then the same experiment was done within a metal faraday and as expected the connection disconnected entirely. As expected there are some materials that effect the connection while others have negligible effect. As a 2.4GHz wireless connection it can be expected to have be affected by materials identically to that of WIFI signals.

4.4.2 Accelerometer Communication Testing

The next step was to confirm communication from the ADXL345 to the CC2650 This was done via the debug menu of the TI-RTOS IDE, Code Composer Studio. The device was orientated in 90-degree steps in one dimension and the output noted at each step, then then the same was done for each other dimension. Another issue was found that while the device ID could be read, no measurements were returning values. It was determined that the device power control registry was not being set correctly but this issue could not be rectified before the writing of this report.

4.4.4 Motion Calculation Testing

Unfortunately, without the correct communication with the ADXL345 the required measurements to conduct this test were not available. Therefore, this testing phase was not completed in time for this report and will need to be completed outside of the contents of this report.

Chapter 5 - Results and Discussion

With the project reaching the level of completion it did there are a few results that can be discussed about the findings.

5.1 MCU and Accelerometer Communication

The results from the communication between the CC2650 and the ADXL345 are not ideal but still can describe a lot about what was learnt throughout the implementation and testing of the device. The main takeaway could be considered fact that device interfaces are not often a plug and play set up, and even when devices have libraries designed to help with this kind of communication a large amount of work may still have to be done tailoring the functionality of these resources to the specific case being designed for. With more time it is highly likely this project could progress to the point where the accelerometer and MCU communicate effectively and therefore it is not strong enough evidence to suggest that the devices investigated in this report could not be used as a 3D Wireless HCI device.

5.2 Wireless Communication

The wireless communication worked as intended using the cc2650. It was fully capable of sending and receiving data in the required environment for the device being investigated within this report. The results suggest that the cc2650's wireless communication abilities could be confidently deployed to fulfil the requirements of a 3D Wireless HCI device, and further development could continue to implement these techniques of wireless communication.

Chapter 6 – Ethics

With the world moving towards everyday life further integrating the use of electronics, the importance of the ethical issues around how people interact with these computers become more important to discuss. In terms of the method of human computer interaction discussed in this project two ethical discussions that are relevant to discuss include the possible further alienation of some users, and the possible benefits for other users and the relationship between these two outcomes.

With the evolution of technology happening at such a rapid rate over the past several decades an unfortunate side effect is that as more of daily life's task must be completed using computers people who are less capable with computers, be it due to learning difficulties, not having the advantage of growing up with these technologies, or early adoption of them, their life becomes more difficult with every complexity introduced to these interactions with technology. Therefore, there is a possibility that a wide adoption of a new input method for base interaction with a computer could have the effect of further alienating these people. This begs the question on what an acceptable level of complexity is to expect of the general populous to understand when interacting with technology.

There is also the possibility that a new form of input could be more intuitive than current methods. Perhaps rather than further alienation of the less tech literate the opposite could occur with a 3D HCI. It may be appropriate for the tech industry to have a stronger attention to research what ways to help those who struggle with modern technology interactions and perhaps the form of input from this report is one worth investigating.

So, on one hand you have the possibility of helping improve some people's quality of life, or the possibility of hurting it. If there was to be wide adoption of this form of input for daily life tasks it will be extremely important for studies to be complete to ensure that there is not those left behind and left struggling with daily life.

Another ethical discussion revolves around the use of wireless devices and the saturation of RF bandwidths in the future. As technology evolves more and more devices communicate wirelessly using RF signals. There is potential for this growth of wireless devices to grow to such an extent that signal interference will become a drastic issue. Minor effects of this are already observable when many WiFi networks overlap and not enough separate 2.4GHz WiFi channels for each network exist. The question in this case is who should take responsibility for this possibility. It could be the case that further research in wireless communications could make RF bandwidth use efficiency so great that interference is a negligible issue. So, should further steps be taken to ensure RF bandwidth does not become a scarce resource. Further limitations on the frequencies available for consumer products could be put in place, or should confidence be placed in the industry to research solutions less restricting. Or perhaps the point of extreme RF saturation will not be reached, and further restrictions now will only hurt development for no long-term benefit. Again, the likely best course of action is for further research into the potential issue and to build a general awareness in the industry of the potential issue.

Chapter 7 - Conclusions and Further Research

The investigation done throughout this report has given an interesting insight into the design and implementation of a 3D Wireless RTOS HMI device. Although the device was not completed within the time frame of this report a lot of value can still be acquired from the progress made. A lot of the value of this work stems from the lessons learned in working within an unfamiliar RTOS software environment, that can be applied both within this project and software and electronic projects in general. It is a lesson to be keen and detailed in the study of a new software environments operation. In regard to TI-RTOS in particular a lot of value can be garnered from what was learned here about the operation and available resources with TI-RTOS. Much of the time with this project was spent interfacing a component using I2C and SPI interfaces. This content is very interesting and can have a wide appeal in electronics due to the massive inventory of devices that use these standards. Finally, there is also interesting information gathered throughout the research and design sections of the project on the design and interfacing of a input device with a host computer. Future work with this project should continue to work towards getting the MCU and Accelerometer to communicate more successfully. To experiment further with the wireless data transfer communication of the cc2650 devices and possibly compare different wireless communication standards the board is capable of, and also to build upon the research done on designing an input device for a computer to interface the device fully with a host computer.

References

[1] K. Sato, “Motion determining apparatus and storage medium having motion determining program stored thereon”. US Patent US 7424388 B2, 9 September 2008.

[2] K. Ohta and K. shi, “Image processing apparatus and storage medium storing image processing program”. US Patent US 2007/0211027 A1, 13 September 2007.

[3] Silicon Labs, “AN249 Human Interface Device Tutorial,” [Online]. Available: https://www.silabs.com/documents/public/application-notes/AN249.pdf. [Accessed 10 February 2019].

[4] Valve Corporation, “Github.com/ValveSoftware,” 2 February 2019. [Online]. Available: https://github.com/ValveSoftware/openvr. [Accessed 5 March 2019].

[5] S. R. Ellis, M. J. Young, B. D. Adelstein and S. M. Ehrlich, “DISCRIMINATION OF CHANGES OF LATENCY,” in *Human Factors and Ergonomics Society,* Houston, Texas, 1999.

[6] Texas Instruments, “SimpleLink™ CC2650 wireless MCU LaunchPad™ Development Kit,” 2019. [Online]. Available: http://www.ti.com/tool/launchxl-cc2650#technicaldocuments. [Accessed 10 February 2019].

[7] P. Cheolhee and S. R. Theodore, “Short-Range Wireless Communications for Next-Generation Networks: UWB, 60 GHz Millimeter-Wave WPAN, and ZigBee,” *IEEE Wireless Communications,* vol. 14, no. 4, pp. 70-78, 2007.

[8] Texas Instruments, “TI SimpleLink MCU Platform,” Texas Instruments, 2018. [Online]. Available: http://www.ti.com/wireless-connectivity/simplelink-solutions/overview/overview.html. [Accessed 2 December 2018].

[9] Analog Devices, “Data Sheet ADXL345 Rev.E,” 2015. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf. [Accessed 12 February 2019].

[10] Texas Instruments, “Adding Custom RTOS Task,” 1 October 2015. [Online]. Available: http://processors.wiki.ti.com/index.php/Adding_Custom_RTOS_Task#Anatomy_of_an_RTOS_Task. [Accessed 12 March 2019].

[11] Texas Instruments Incorporated, “SPI.h File Reference,” 2017. [Online]. Available: http://dev.ti.com/tirex/content/simplelink_cc26x2_sdk_1_60_00_43/docs/tidrivers/doxygen/html/_s_p_i_8h.html. [Accessed 12 March 2019].

[12] Texas Instruments Incorporated, “I2C.h File Reference,” 2016. [Online]. Available: http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/tirtos/2_20_00_06/exports/tirtos_full_2_20_00_06/products/tidrivers_full_2_20_00_08/docs/doxygen/html/_i2_c_8h.html. [Accessed 12 March 2019].

[13] Texas Instruments, “TI Resouce Explorer - CC2650 LaunchPad,” [Online]. Available: http://dev.ti.com/tirex/#/DevTool/CC2650%20LaunchPad/?link=Development%20Tools%2FKits%20and%20Boards%2FCC2650%20LaunchPad. [Accessed 20 March 2019].

[14] Texas Instruments, “CC2650 LuanchPad QuickStart Guide,” 2015. [Online]. Available: http://www.ti.com/lit/ml/swru451/swru451.pdf. [Accessed 14 March 2019].

[15] O. Iason, K. Nikolaos and A. A. Antonis, “Efficient Model-based 3D Tracking of,” in *British Machine Vision Confrenece*, Dundee, 2011.

Appendix

1. TI-RTOS Flashing LED Code:

```c
/* XDCtools Header files */
#include <xdc/std.h>
#include <xdc/runtime/System.h>
/* BIOS Header files */
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/knl/Task.h>
/* TI-RTOS Header files */
#include <ti/drivers/PIN.h>
/* Board Header files */
#include "Board.h"

#define TASKSTACKSIZE   512

Task_Struct task0Struct;
Char task0Stack[TASKSTACKSIZE];

/* Pin driver handle */
static PIN_Handle MyPinHandle;
static PIN_State MyPinState;

PIN_Config MyPinTable[] = {
    Board_LED0 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    Board_LED1 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    PIN_TERMINATE
};

Void Flashing(UArg arg0, UArg arg1)
{
    while (true) {
        Task_sleep((UInt)arg0);
        PIN_setOutputValue(MyPinHandle, Board_LED0, !PIN_getOutputValue(Board_LED0));
        PIN_setOutputValue(MyPinHandle, Board_LED1, !PIN_getOutputValue(Board_LED1));
    }
}

int main(void)
{
    Task_Params taskParams;
    Board_initGeneral();

    /* Construct Task  thread */
    Task_Params_init(&taskParams);
    taskParams.arg0 = 1000000 / Clock_tickPeriod;
    taskParams.stackSize = TASKSTACKSIZE;
    taskParams.stack = &task0Stack;
    Task_construct(&task0Struct, (Task_FuncPtr)Flashing, &taskParams, NULL);

    /* Open LED pins */
    MyPinHandle = PIN_open(&MyPinState, MyPinTable);
    PIN_setOutputValue(MyPinHandle, Board_LED1, 1);
    PIN_setOutputValue(MyPinHandle, Board_LED0, 0);

    System_printf("Starting the LED Flashing");
    System_flush();
    BIOS_start();

    return (0);
}
```

2. SPI Device ID Code:

```
#include <xdc/std.h>
#include <xdc/runtime/System.h>

/* BIOS Header files */
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/drivers/PIN.h>
#include <ti/drivers/SPI.h>
#include <ti/drivers/GPIO.h>
#include <ti/drivers/Watchdog.h>

/* Board Header files */
#include "Board.h"

#define TASKSTACKSIZE   512

Task_Struct task0Struct;
Char task0Stack[TASKSTACKSIZE];

void delay_s(int dly) {

    while (dly > 0) {
        __delay_cycles(48000000);
        dly--;

    }
}
void delay_ms(int dly) {

    while (dly > 0) {
        __delay_cycles(48000);
        dly--;

    }
}


Char myTaskStack[1024];
Task_Struct myTaskStruct;


Void SPIFxn(UArg arg0, UArg arg1) {
  System_printf("Start SPI task\n");

  SPI_Handle handle;
  SPI_Params spiParams;
  SPI_Transaction spiTransaction;
  uint8_t transmitBuffer[2];
  uint8_t recieveBuffer[2];
  bool ret;

  SPI_Params_init(&spiParams); //initialise spi params object
  spiParams.transferMode = SPI_MODE_BLOCKING; //block thread until transaction complete
  spiParams.transferTimeout = SPI_WAIT_FOREVER;//wait forever
  spiParams.frameFormat = SPI_POL1_PHA1;//clock polarity 1 phase 1
  spiParams.transferCallbackFxn = NULL; //no callback
  spiParams.mode = SPI_MASTER; //master mode
  spiParams.bitRate = 400000; //400MHz
  spiParams.dataSize = 8; //data size
  handle = SPI_open(0, &spiParams);//open spiHandle

  transmitBuffer[0] = 0xAD;//write to power control
  transmitBuffer[1] = 0x08;//turn device on

  spiTransaction.count = 2;
  spiTransaction.txBuf = transmitBuffer;
  spiTransaction.rxBuf = recieveBuffer;

  ret = SPI_transfer(handle, &spiTransaction);
  if (!ret) {
      System_printf("Unsuccessful SPI transfer");
  }
  delay_s(3);

  while(true){

      transmitBuffer[0] = 0x80;//read from address 0x00 with read bit 0x80 (devID)
      transmitBuffer[1] = 0x00;//fake request for interchange

      ret = SPI_transfer(handle, &spiTransaction);
      if (!ret) {
            System_printf("Unsuccessful SPI transfer");
        }
      delay_s(3);
      System_printf("Devid = %D", recieveBuffer[1]);
      System_flush();
      delay_s(3);
  }

}

// ======== SETUP ========
Int main(){

  System_printf("Start Setup\n");

  Board_initGeneral();
  Board_initSPI();

  /* Configure task. */
  Task_Params taskParams;
  Task_Params_init(&taskParams);
  taskParams.stack = myTaskStack;
  taskParams.stackSize = sizeof(myTaskStack);
  Task_construct(&myTaskStruct, SPIFxn, &taskParams, NULL);

  System_printf("End Setup\n");

  BIOS_start();

  return (0);
}
```

3. TI Packet Transmission Code

```c
/*
 * Copyright (c) 2015-2016, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/***** Includes *****/
#include <stdlib.h>
#include <xdc/std.h>
#include <xdc/runtime/System.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>

/* Drivers */
#include <ti/drivers/rf/RF.h>
#include <ti/drivers/PIN.h>

/* Board Header files */
#include "Board.h"

#include "smartrf_settings/smartrf_settings.h"

/* Pin driver handle */
static PIN_Handle ledPinHandle;
static PIN_State ledPinState;

/*
 * Application LED pin configuration table:
 *   - All LEDs board LEDs are off.
 */
PIN_Config pinTable[] =
{
    Board_LED0 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    Board_LED1 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    PIN_TERMINATE
};


/***** Defines *****/
#define TX_TASK_STACK_SIZE 1024
#define TX_TASK_PRIORITY   2

/* Packet TX Configuration */
#define PAYLOAD_LENGTH      30
#define PACKET_INTERVAL     (uint32_t)(4000000*0.5f) /* Set packet interval to 500ms */
```

```c
/***** Prototypes *****/
static void txTaskFunction(UArg arg0, UArg arg1);



/***** Variable declarations *****/
static Task_Params txTaskParams;
Task_Struct txTask;    /* not static so you can see in ROV */
static uint8_t txTaskStack[TX_TASK_STACK_SIZE];

static RF_Object rfObject;
static RF_Handle rfHandle;

uint32_t time;
static uint8_t packet[PAYLOAD_LENGTH];
static uint16_t seqNumber;
static PIN_Handle pinHandle;


/***** Function definitions *****/
void TxTask_init(PIN_Handle inPinHandle)
{
    pinHandle = inPinHandle;

    Task_Params_init(&txTaskParams);
    txTaskParams.stackSize = TX_TASK_STACK_SIZE;
    txTaskParams.priority = TX_TASK_PRIORITY;
    txTaskParams.stack = &txTaskStack;
    txTaskParams.arg0 = (UInt)1000000;

    Task_construct(&txTask, txTaskFunction, &txTaskParams, NULL);
}

static void txTaskFunction(UArg arg0, UArg arg1)
{
    uint32_t time;
    uint8_t test = 12;
    RF_Params rfParams;
    RF_Params_init(&rfParams);

    RF_cmdPropTx.pktLen = PAYLOAD_LENGTH;
    RF_cmdPropTx.pPkt = packet;
    RF_cmdPropTx.startTrigger.triggerType = TRIG_ABSTIME;
    RF_cmdPropTx.startTrigger.pastTrig = 1;
    RF_cmdPropTx.startTime = 0;

    /* Request access to the radio */
    rfHandle = RF_open(&rfObject, &RF_prop, (RF_RadioSetup*)&RF_cmdPropRadioDivSetup, &rfParams);

    /* Set the frequency */
    RF_postCmd(rfHandle, (RF_Op*)&RF_cmdFs, RF_PriorityNormal, NULL, 0);

    /* Get current time */
    time = RF_getCurrentTime();
    while(1)
    {
        /* Create packet with incrementing sequence number and random payload */
        packet[0] = (uint8_t)(8);
        packet[1] = (uint8_t)(seqNumber++);
        uint8_t i;
        for (i = 2; i < PAYLOAD_LENGTH; i++)
        {
            packet[i] =  test;
        }

        /* Set absolute TX time to utilize automatic power management */
        time += PACKET_INTERVAL;
        RF_cmdPropTx.startTime = time;

        /* Send packet */
        RF_EventMask result = RF_runCmd(rfHandle, (RF_Op*)&RF_cmdPropTx, RF_PriorityNormal, NULL, 0);
        if (!(result & RF_EventLastCmdDone))
        {
            /* Error */
            while(1);
        }

        PIN_setOutputValue(pinHandle, Board_LED0,!PIN_getOutputValue(Board_LED0));
    }
}

/*
 *  ======== main ========
 */
int main(void)
{
    /* Call board init functions. */
    Board_initGeneral();

    /* Open LED pins */
    ledPinHandle = PIN_open(&ledPinState, pinTable);
    if(!ledPinHandle)
    {
        System_abort("Error initializing board LED pins\n");
    }

    /* Initialize task */
    TxTask_init(ledPinHandle);

    /* Start BIOS */
    BIOS_start();

    return (0);
}
```

4. TI Packet Reciever Code

```c
/*
 * Copyright (c) 2015-2016, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/***** Includes *****/
#include <stdlib.h>
#include <xdc/std.h>
#include <xdc/cfg/global.h>
#include <xdc/runtime/System.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>

/* Drivers */
#include <ti/drivers/rf/RF.h>
#include <ti/drivers/PIN.h>
#include <driverlib/rf_prop_mailbox.h>

/* Board Header files */
#include "Board.h"

#include "RFQueue.h"
#include "smartrf_settings/smartrf_settings.h"

#include <stdlib.h>

/* Pin driver handle */
static PIN_Handle ledPinHandle;
static PIN_State ledPinState;

/*
 * Application LED pin configuration table:
 *   - All LEDs board LEDs are off.
 */
PIN_Config pinTable[] =
{
    Board_LED0 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    Board_LED1 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    PIN_TERMINATE
};


/***** Defines *****/
#define RX_TASK_STACK_SIZE 1024
#define RX_TASK_PRIORITY   2
```

```
/* Packet RX Configuration */
#define DATA_ENTRY_HEADER_SIZE 8  /* Constant header size of a Generic Data Entry */
#define MAX_LENGTH             30 /* Max length byte the radio will accept */
#define NUM_DATA_ENTRIES       2  /* NOTE: Only two data entries supported at the moment */
#define NUM_APPENDED_BYTES     2  /* The Data Entries data field will contain:
                                   * 1 Header byte (RF_cmdPropRx.rxConf.bIncludeHdr = 0x1)
                                   * Max 30 payload bytes
                                   * 1 status byte (RF_cmdPropRx.rxConf.bAppendStatus = 0x1) */



/***** Prototypes *****/
static void rxTaskFunction(UArg arg0, UArg arg1);
static void callback(RF_Handle h, RF_CmdHandle ch, RF_EventMask e);

/***** Variable declarations *****/
static Task_Params rxTaskParams;
Task_Struct rxTask;    /* not static so you can see in ROV */
static uint8_t rxTaskStack[RX_TASK_STACK_SIZE];

static RF_Object rfObject;
static RF_Handle rfHandle;

/* Buffer which contains all Data Entries for receiving data.
 * Pragmas are needed to make sure this buffer is 4 byte aligned (requirement from the RF Core) */
#if defined(__TI_COMPILER_VERSION__)
    #pragma DATA_ALIGN (rxDataEntryBuffer, 4);
        static uint8_t rxDataEntryBuffer[RF_QUEUE_DATA_ENTRY_BUFFER_SIZE(NUM_DATA_ENTRIES,
                                                                 MAX_LENGTH,
                                                                 NUM_APPENDED_BYTES)];
#elif defined(__IAR_SYSTEMS_ICC__)
    #pragma data_alignment = 4
        static uint8_t rxDataEntryBuffer[RF_QUEUE_DATA_ENTRY_BUFFER_SIZE(NUM_DATA_ENTRIES,
                                                                 MAX_LENGTH,
                                                                 NUM_APPENDED_BYTES)];
#elif defined(__GNUC__)
        static uint8_t rxDataEntryBuffer [RF_QUEUE_DATA_ENTRY_BUFFER_SIZE(NUM_DATA_ENTRIES,
            MAX_LENGTH, NUM_APPENDED_BYTES)] __attribute__ ((aligned (4)));
#else
    #error This compiler is not supported.
#endif

/* Receive dataQueue for RF Core to fill in data */
static dataQueue_t dataQueue;
static rfc_dataEntryGeneral_t* currentDataEntry;
static uint8_t packetLength;
static uint8_t* packetDataPointer;

static PIN_Handle pinHandle;

static uint8_t packet[MAX_LENGTH + NUM_APPENDED_BYTES - 1]; /* The length byte is stored in a
separate variable */


/***** Function definitions *****/
void RxTask_init(PIN_Handle ledPinHandle) {
    pinHandle = ledPinHandle;

    Task_Params_init(&rxTaskParams);
    rxTaskParams.stackSize = RX_TASK_STACK_SIZE;
    rxTaskParams.priority = RX_TASK_PRIORITY;
    rxTaskParams.stack = &rxTaskStack;
    rxTaskParams.arg0 = (UInt)1000000;

    Task_construct(&rxTask, rxTaskFunction, &rxTaskParams, NULL);
}

static void rxTaskFunction(UArg arg0, UArg arg1)
{
    RF_Params rfParams;
    RF_Params_init(&rfParams);

    if( RFQueue_defineQueue(&dataQueue,
                            rxDataEntryBuffer,
                            sizeof(rxDataEntryBuffer),
                            NUM_DATA_ENTRIES,
                            MAX_LENGTH + NUM_APPENDED_BYTES))
    {
        /* Failed to allocate space for all data entries */
        while(1);
    }

    /* Modify CMD_PROP_RX command for application needs */
    RF_cmdPropRx.pQueue = &dataQueue;           /* Set the Data Entity queue for received data */
    RF_cmdPropRx.rxConf.bAutoFlushIgnored = 1;  /* Discard ignored packets from Rx queue */
    RF_cmdPropRx.rxConf.bAutoFlushCrcErr = 1;   /* Discard packets with CRC error from Rx queue */
    RF_cmdPropRx.maxPktLen = MAX_LENGTH;        /* Implement packet length filtering to avoid
PROP_ERROR_RXBUF */
    RF_cmdPropRx.pktConf.bRepeatOk = 1;
    RF_cmdPropRx.pktConf.bRepeatNok = 1;

    /* Request access to the radio */
    rfHandle = RF_open(&rfObject, &RF_prop, (RF_RadioSetup*)&RF_cmdPropRadioDivSetup, &rfParams);

    /* Set the frequency */
    RF_postCmd(rfHandle, (RF_Op*)&RF_cmdFs, RF_PriorityNormal, NULL, 0);

    /* Enter RX mode and stay forever in RX */
    RF_runCmd(rfHandle, (RF_Op*)&RF_cmdPropRx, RF_PriorityNormal, &callback, IRQ_RX_ENTRY_DONE);

    while(1);
}

void callback(RF_Handle h, RF_CmdHandle ch, RF_EventMask e)
{
    if (e & RF_EventRxEntryDone)
    {
        /* Toggle pin to indicate RX */
        PIN_setOutputValue(pinHandle, Board_LED1,!PIN_getOutputValue(Board_LED1));

        /* Get current unhandled data entry */
        currentDataEntry = RFQueue_getDataEntry();

        /* Handle the packet data, located at &currentDataEntry->data:
         * - Length is the first byte with the current configuration
         * - Data starts from the second byte */
        packetLength      = *(uint8_t*)(&currentDataEntry->data);
        packetDataPointer = (uint8_t*)(&currentDataEntry->data + 1);

        System_printf("Testing %d", *packetDataPointer);
        System_flush();

        /* Copy the payload + the status byte to the packet variable */
        memcpy(packet, packetDataPointer, (packetLength + 1));

        RFQueue_nextEntry();
    }
}

/*
 *  ======== main ========
 */
int main(void)
{
    /* Call board init functions. */
    Board_initGeneral();
    System_printf("Test\n");
    System_flush();
    /* Open LED pins */
    ledPinHandle = PIN_open(&ledPinState, pinTable);
    if(!ledPinHandle)
    {
        System_abort("Error initializing board LED pins\n");
    }

    /* Initialize task */
    RxTask_init(ledPinHandle);

    /* Start BIOS */
    BIOS_start();

    return (0);
}
```