



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Control of RF Components within a Passive RF Tracking System

Jack Heeks – 2111738

School of Applied Computing

Supervisor: Jennifer Prince Devaraj

Project report submitted as part of the requirements for the
award of

BEng (Hons) Embedded Electronic Systems

April 2025

Abstract

This project investigates the development of a system designed to control the programmable RF components within a passive RF tracking system utilised by Maritime Countermeasures team during trials. The objective was to optimize the accuracy, reliability, and efficiency of RF device control within the tracking system to allow for greater control of the system during operation.

A prototype control system was designed using PIC18 microcontrollers and utilising the various communication protocols available (UART, I2C, and SPI) to interface with the RF devices, such as tuneable bandpass filters and a frequency synthesizer.

Extensive simulation, PCB design, and prototype testing were undertaken to ensure robust data transmission, precise device control, and adaptability for the variety of potential test scenarios. Custom firmware incorporating interrupt-driven data handling and safety mechanisms was developed to guarantee operational integrity. The project also addressed ethical sourcing, risk management, and environmental considerations. The final system achieved reliable RF device programming with the flexibility required for dynamic field trials, providing a scalable foundation for future device control enhancements.

Contents

Table of Figures	iv
List of Tables	iv
Introduction	1
Literature Review	2
Microcontroller	2
Data Protocols	4
One Wire	5
Universal Asynchronous Receiver-Transmitter	5
Inter-Integrated Circuit	6
Serial Peripheral Interface	7
Control Area Network	7
Universal Serial Bus	7
PCB Design	8
Power	8
Signals	9
In-Circuit Serial Programming	10
Development Methodology	12
Code Development	12
RF Devices	12
Intended Functionality	12
Flowcharts	13
Integrated Development Environment	16
Functions	16
Interrupt Routines	20
Simulation	24
Development Kit	24
Prototype Development	25
Wiring diagram	25
Schematic	25
Stripboard Translation	26
LCD Display	26
Power Sensing Circuitry	26
Printed Circuit Board	28
Control Board	28

Prototype Testing	33
Project Management	36
Ethical Concerns	36
Project Risks	37
Risk Assessment	39
Future Improvement	42
Conclusion	44
Reflection	44
References	46
Bibliography	50
Appendix I	53
Host PIC	53
main.c	53
main.h	58
interrupts.c	58
interrupts.h	60
Appendix II	61
Transfer PIC	61
main.c	61
main.h	66
interrupts.c	66
interrupts.h	68
Appendix III	69
Display PIC	69
main.c	69
main.h	72
Interrupts.c	74
interrupts.h	75
Libraries	75
LCD.c	75
LCD.h	83
Appendix IV	86
Redundant Functions	86
spi_transmit()	86
UARTRxISR()	87
Appendix V	88

Control of RF Components within a Passive RF Tracking System

Data Sheets	88
RF Devices	88
Microcontroller	88
Programmable Oscillator	88
Voltage Regulators	88
Comparators	88
USB-Serial IC	88
Appendix VI	89
Proposed Gantt Chart	89
Actual Project Timeline	90
Appendix VII	91
Project Risk Priority Number Tables	91
Appendix VIII	92
Risk Matrix Tables	92

Table of Figures

Figure 1 – RISC Architecture ^[1]	2
Figure 2 – CPU Harvard Architecture ^[5]	2
Figure 3 – Example PIC18Fxxx Architecture ^[10]	3
Figure 4 – Serial and Parallel Data Flow	4
Figure 5 – Asynchronous Serial Signal	4
Figure 6 – Synchronous Serial Signal	4
Figure 7 – Data Transmission Methodologies	4
Figure 8 – Communication Protocol Categories	5
Figure 9 – 1-Wire Data Protocol	5
Figure 10 – UART Data Protocol	6
Figure 11 – I2C Block Diagram	6
Figure 12 – SPI Block Diagram	7
Figure 13 – CAN Block Protocol	7
Figure 14 – USB Connections	8
Figure 15 – Planes Throughout a PCB	9
Figure 16 – SPI Standard and Daisy Chain Methods	10
Figure 17 – Potential In-Circuit Serial Programming Layout	11
Figure 18 – YIG Filter Datasheet Screenshot ^[38]	12
Figure 19 – Frequency Synthesiser Datasheet Screenshot ^[39]	12
Figure 20 – Host PIC Flowchart	13
Figure 21 – Transfer PIC Flowchart	14
Figure 22 – Transfer PIC Flowchart	15
Figure 23 – Block Diagram	24
Figure 24 – Curiosity Nano Evaluation Kit	24
Figure 25 – Proteus Simulation	24
Figure 26 – Wiring Diagram	25
Figure 27 – Schematic Overview	25
Figure 28 – LCD Testing Stripboard	26
Figure 29 – LM393 based under and over voltage sensing circuit	27
Figure 30 – Control Board Circuit Schematic	29
Figure 31 – Control Board PCB Top Copper Layer	29
Figure 32 – Control Board PCB Ground Plane Layer	30
Figure 33 – Control Board PCB 5V Power Plane Layer	30
Figure 34 – Control Board PCB Bottom Copper Layer	31
Figure 35 – 3D View of Designed PCB	31
Figure 36 – Design Rules Checker Output	32
Figure 37 – Bench Testing	33
Figure 38 – Proteus I2C Debugger	33
Figure 39 – UART Final Character and I2C Transmission	34
Figure 40 – I2C Frame Data	35
Figure 41 – SPI Data Frame	35
Figure 42 – Proposed Gantt Chart	36

List of Tables

Table 1 – I2C Transmission Speeds Table	9
Table 2 – Table of Identified Project Risks	38
Table 3 – Work-based Risk Assessment	41

Introduction

Threat Detection and Countermeasures (TDaC) Group's Maritime Countermeasures Team has been developing a passive Radio Frequency (RF) tracking system for use in a variety of trials. The aim of which will be to obtain the RF transmissions for use in wider ongoing research through observation and tracking of RF sources; allowing tests to be performed and studies carried out during further trials.

This project aims to explore and develop the control mechanisms, through the development of a prototype control system, required to program RF devices. These devices are used within the passive tracking system – a method of tracking RF sources without the need for active transmission. A passive system was chosen as it is simpler to implement and more cost effective than an active system, which also requires more power to operate. There are also licensing requirements and EMF hazard considerations which apply when transmitting RF in open air, also all the targets of interest are transmitting RF signals we are able to receive.

The goal of this project is to develop the accuracy, reliability, and efficiency of the passive RF tracking system by optimizing the control of the RF components within the system. This will involve designing, simulating, and testing the system to develop the prototype. These key stages will then be used to implement the control algorithms and techniques researched to develop the device control and system performance. The system requirements include the capability to reprogramme the devices, via inputting key parameters, for versatility across multiple frequency bands for a multitude of test scenarios. It needs to be compact to allow for integration to an enclosure mounted on the Pan-Tilt head hosting and aiming the RF tracking capability.

Literature Review

Microcontroller

Peripheral Interface Controllers (PIC) are a type of microcontroller developed by Microchip Technology in 1993, these microcontrollers were produced to support Programmed Data Processor (PDP) computers interfacing with periphery devices. PIC microcontrollers are popular for their ease of programming, low cost, wide range and availability, and serial programming capability.^[1] Due to their popularity PIC microcontrollers are used for a wide range of applications, from simple sensor nodes to complex real-time control systems. Although the assembly language is specific to the PIC microcontroller architecture, they are also able to support several programming languages including C language.^[2]

PIC microcontrollers are built around Reduced Instruction Set Computer (RISC) architecture (shown in Figure 1), which utilises a small and highly optimised set of instructions as opposed to highly specialized set of instructions typically found in other architectures.^[3] The memory architecture follows the Harvard design (Figure 2) of separated program and data memories and buses.^[4]

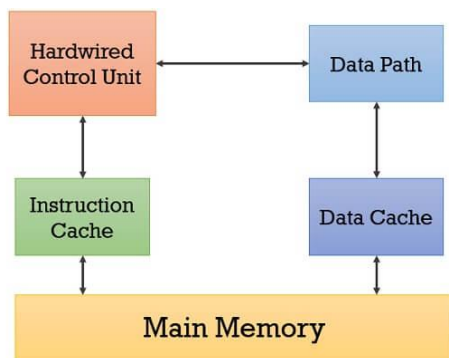


Figure 1 – RISC Architecture^[1]

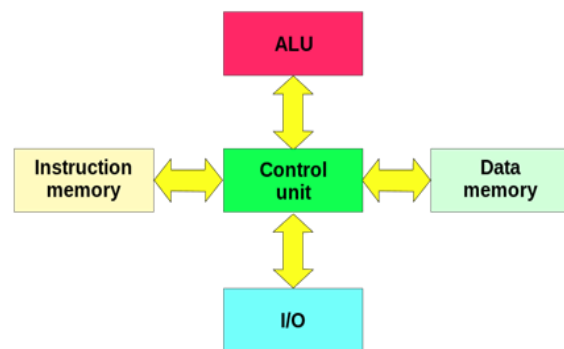


Figure 2 – CPU Harvard Architecture^[5]

PIC Microcontroller's CPU consists of:

- Arithmetic logic unit (ALU)
- Memory unit (MU)
- Control unit (CU)
- Accumulator

PIC Microcontrollers are divided into 4 large families:

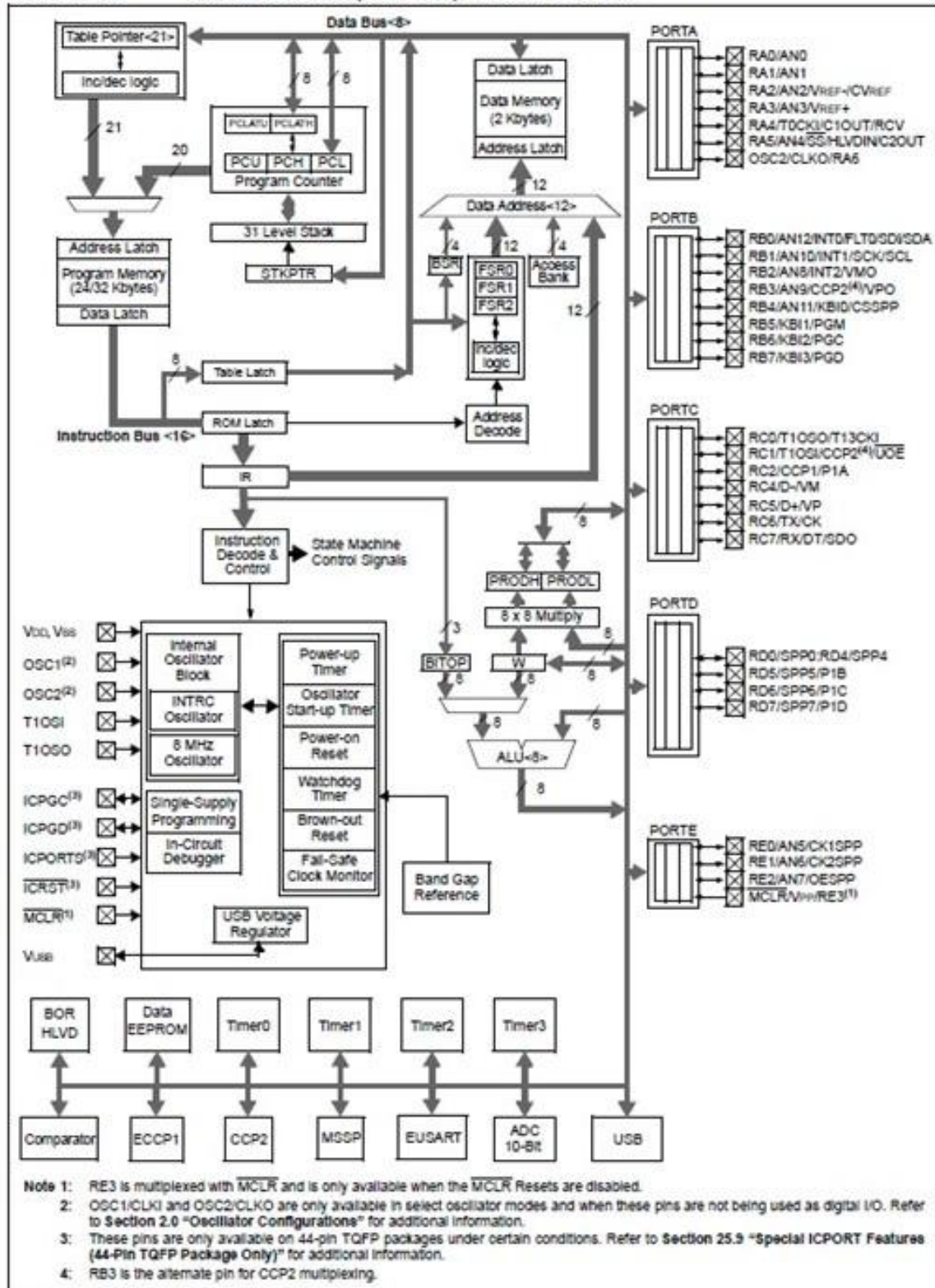
- First family PIC10 (10FXXX) called Low End
- Second family PIC12 (PIC12FXXX) called Mid-Range
- Third family PIC16 (16FXXX)
- Fourth family PIC 17/18 (18FXXX)^[6]

The preferred microcontroller is the PIC18 family as it has a 16bit program word architecture and advanced RISC architecture. It is also capable of providing 16 MIPS and has linear memory as opposed to segmented memory.^[7] This series of microcontrollers have more features and an increased instruction set, with 16bit instruction word, making them more sophisticated and better suited to C compilers.^[8]

The 18 series feature enhanced status registers, a phase-locked loop (PLL) clock generator, ability to hardware multiply two 8bit numbers, and more external interrupts with two prioritised interrupt vectors. These PICs also have very different memory structures with increased storage, alongside

2111738

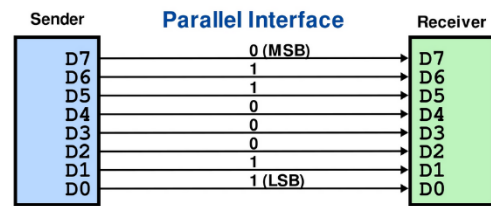
Figure 3 – Example PIC18Fxxxx Architecture^[10]



Data Protocols

Data communication can be separated into Parallel and Serial.^[11]

During parallel communication, each bit of data is transmitted simultaneously on separate communication lines. This method is used for shorter distance communication. Number of wires required for transmission depends on the number of bits being transmitted, making it less cost efficient and increasing the circuit complexity. But it is faster than serial transmission.



Serial communication transmits data bit by bit on single wire. Serial is used for long distance transmissions. A transmission requires only one communication line to transmit data, meaning this is more cost efficient, with a less complex circuit than parallel data transmission.^{[12], [13]}

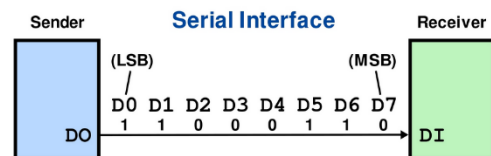


Figure 4 – Serial and Parallel Data Flow

Serial communication utilises two methods:

- **Asynchronous**^[14]
Transfers single byte at a time
Does not a clock signal
Examples include UART, RS232

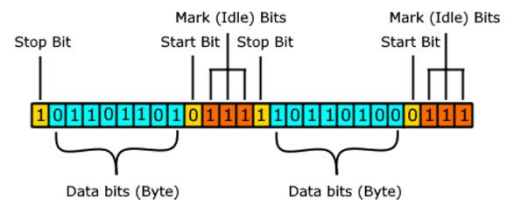


Figure 5 – Asynchronous Serial Signal

- **Synchronous**^[15]
Transfers a block of data (characters) at a time
Requires a clock signal
Examples include SPI, I2C

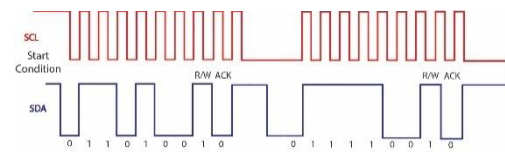


Figure 6 – Synchronous Serial Signal

When data is transmitted and received, it is a duplex transmission. Duplex is divided into three categories: ^{[16], [17]}

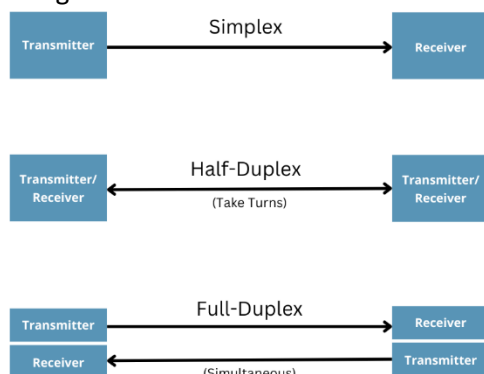


Figure 7 – Data Transmission Methodologies

Simplex – Data is transmitted in only one direction.

Half duplex – Data is transmitted in two directions but only one way at a time.

Full duplex – Data is transmitted both ways at the same time.

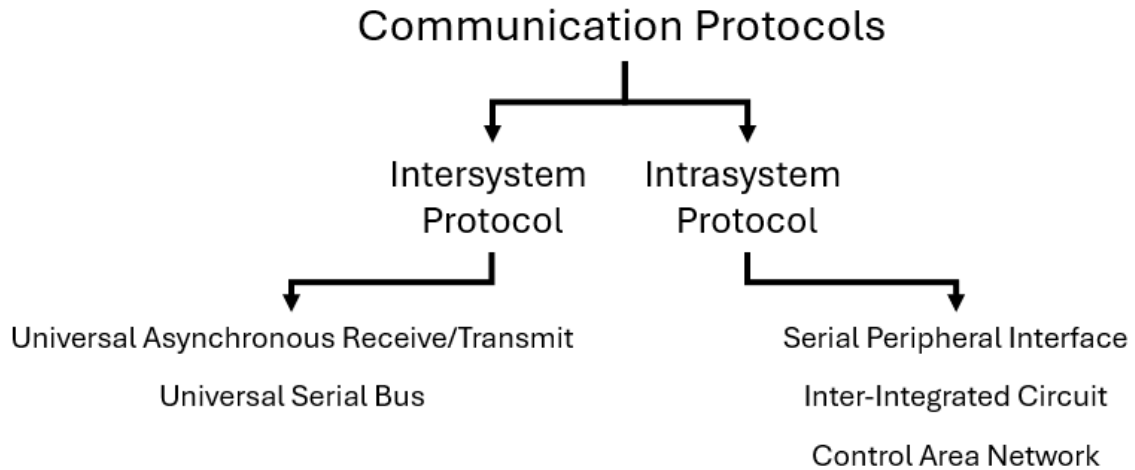


Figure 8 – Communication Protocol Categories

One Wire

Using the 1-Wire data protocol a one-to-many connection can be established with a single data transmission or power wire; resulting in a half-duplex, bi-directional connection with data rates of 16.3kHz.^[18] This is a serial transmission protocol utilising a single data line and ground reference, “Each 1-Wire slave device has a unique, unalterable, factory-programmed, 64-bit identification number (ID), which serves as device address on the 1-Wire bus”.^[19] Typically 1-Wire systems operate over one of four voltage ranges between 1.71 – 5.25V and most devices along the bus have no power supply as the bus acts as a parasitic power supply. Unlike other serial communication protocols 1-Wire devices are designed for momentary contact environments; disconnecting or loss of contact from the bus restores slave devices to a reset state.

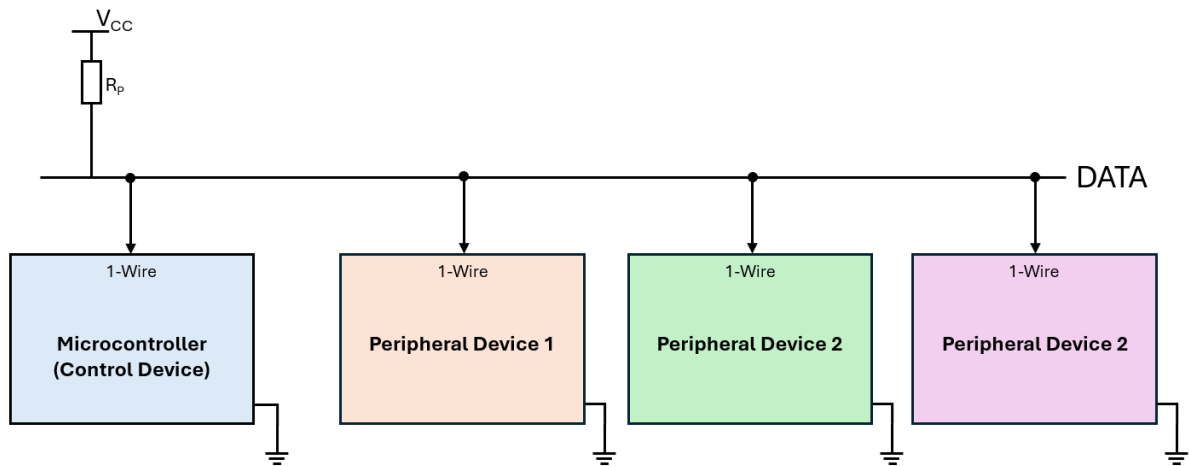


Figure 9 – 1-Wire Data Protocol

Universal Asynchronous Receiver-Transmitter

This Protocol was used to receive commands sent by the user from the control laptop and send instructions to the Frequency Synthesiser.

UART was designed to communicate asynchronously with devices like computers or GPS without using too many CPU cycles, making it an efficient data transmission protocol. This is most commonly used for serial communication between communications and sensors.^[20]

UART transmits serial data as parallel data – typically one byte – received from a data bus sent by another component, it is a two-wired connection between UART nodes. , “this protocol also specifies the number of bits of data and the parity”^[21] The data rate must be set at Tx and Rx prior to transmission else the timing will be off and the data rendered unusable.

UART identifies data packets with a low start bit and high stop bit due to the absence of a clock pulse. UART supports bi-directional, asynchronous and serial data transmission. Can operate as a Simplex, Half-Duplex, or Full-Duplex method.^[22]

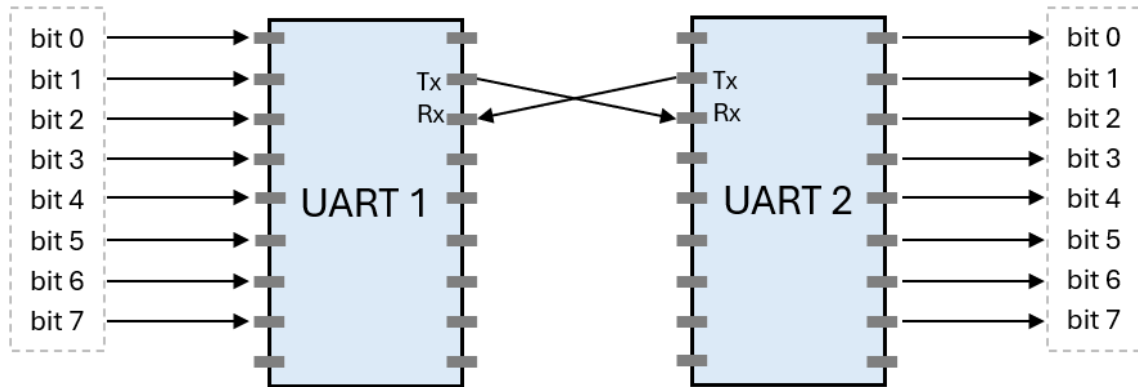


Figure 10 – UART Data Protocol

Inter-Integrated Circuit

This protocol was used for communication between the PIC devices on the PCB.

I²C is a two-wire serial communication protocol enabling many to many data connections. This protocol utilises a data transmission line and a clock signal and these lines are kept in a high voltage state using pull-up resistors, however while transmitting the voltages along both lines vary to send the data.^[18] This protocol typically is used to transmit data to and from multiple master and slave devices, such as microcontrollers and sensors, using 7bit addresses to transmit data to one or more of the devices along the bus.^[20] I²C utilises an open-drain output driver to interface the data and clock signals.

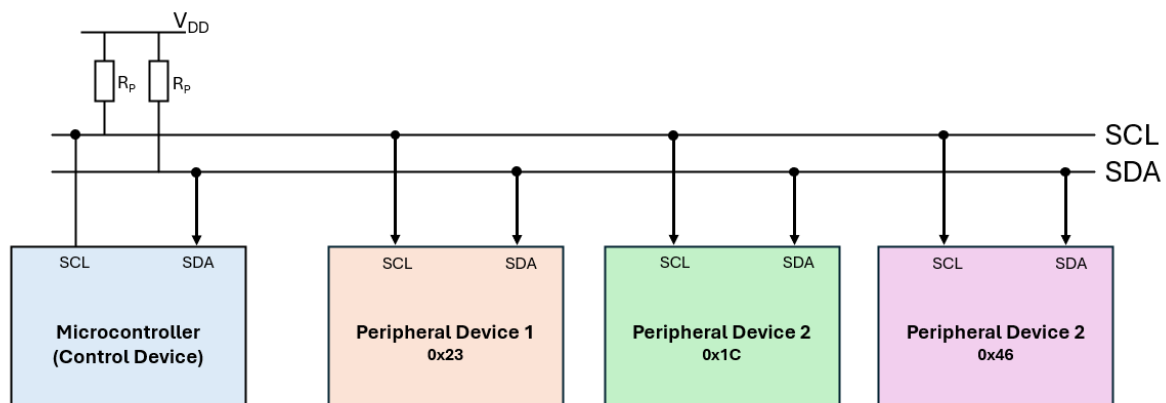


Figure 11 – I2C Block Diagram

Serial Peripheral Interface

SPI was used to send instructions to the Bandpass filters.

SPI is a full-duplex serial data transmission protocol allowing for continuous and uninterrupted data transmission as there are two data lines, therefore no need for start and stop bits as with UART and I²C data transmissions.^[16] Device addressing is done individually via the Chip Select pins of the Control microcontroller pulling a CS line low so that the chip can receive data bytes from the bus or write data bytes to the bus for the Control chip.^[22] SPI is typically used to transfer data between microcontrollers and microprocessors, at high speeds as well as consuming less power, over a short distance.^[20] However, SPI requires four lines to send data to one peripheral device and does not receive an acknowledgement of data sent.^[21]

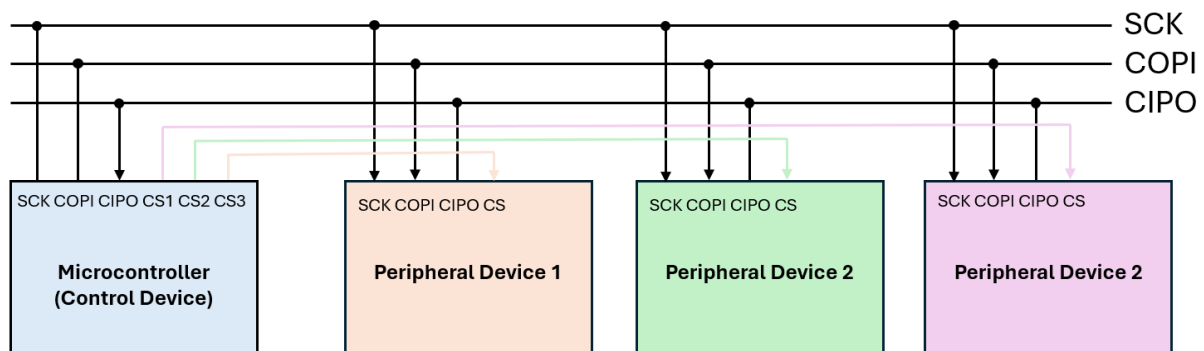


Figure 12 – SPI Block Diagram

Control Area Network

The CAN protocol transmits data throughout a network without the need for a centralized communications controller. CAN is primarily utilised in the automotive industry; used to send and receive data among devices and sensors on a vehicle. It is also applied in industrial applications for inter-communication between devices and a control system.

CAN is able to self-diagnose and fix data errors, it also has high electrical interference immunity. CAN is a broadcast bus meaning it is not possible to send a specific message to a particular node the messages are relatively short – 94 bits maximum utility road. The hardware provides local filtering allowing each node can focus primarily on the messages relevant to it.^[20]

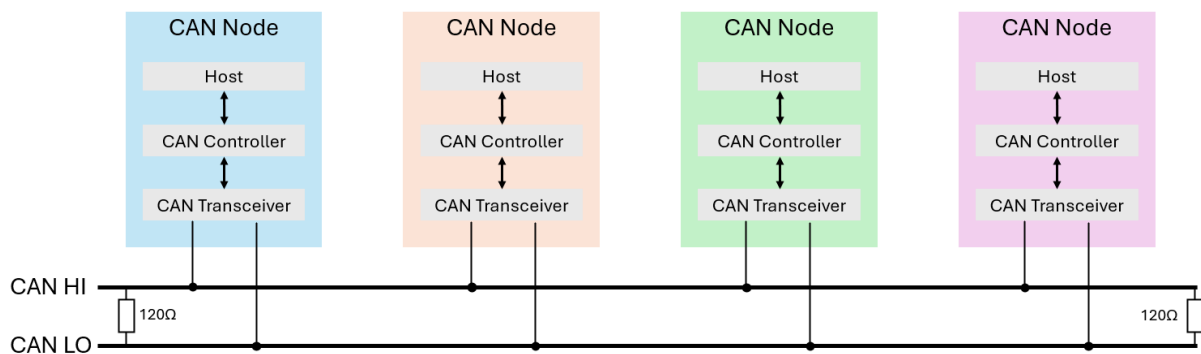


Figure 13 – CAN Block Protocol

Universal Serial Bus

The intension of the UBS protocol is to allow device hot swapping and enhance plug-N-play for systems. Plug-and-play allows the Operating System to configure and discover a new peripheral

without starting the computer, whereas hot swapping removes and replaces a new peripheral device without rebooting the computer.

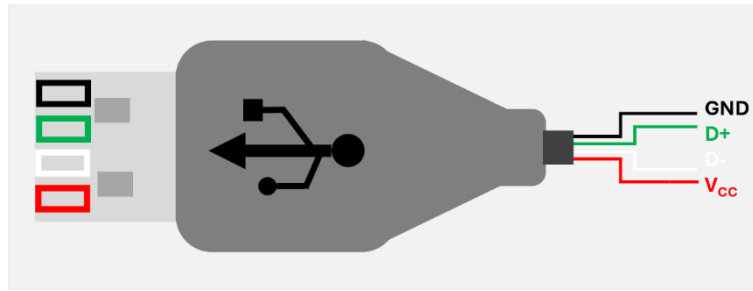


Figure 14 – USB Connections

All USB data transfers are initiated by the host as it uses a polling mechanism, the processor continually checks whether the input/output device is prepared for transmitting data or not. This means only the processor has to check the I/O devices making USB low-cost and simple.^[23]

The host controls the communication timing intervals (frames) determined by the USB speed and issues a Start of Frame (SOF) sequences on the data lines at the beginning of each frame. The USB host performs the data transmissions during the frames.^[24]

Data is broadcast to the bus and assigned to devices by unique numbers, these numbers are assigned by the host when devices are attached to the bus. Each device also contains endpoints, which are the sources and destinations of communications to and from the bus.^[25] These endpoints used for a variety of activities including auto-detection and configuration of the device on the bus and the two zero endpoints are the only ones accessible until the device is properly connected on the bus.

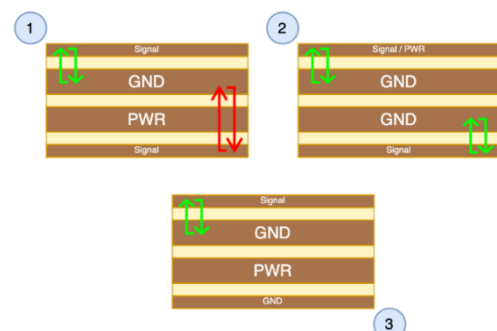
USB data flow is based around data pipes, logical channels within the bus forming connections between host and device logical entity. The data pipe uses the address, endpoint and also the direction to define it and ensure data is correctly directed to and from the device. There are two types of USB pipe; bi-directional message pipes for short, simple commands and status responses like control data; and uni-directional stream pipes for transferring isochronous, interrupt or bulk transfers.^{[25], [26]}

PCB Design

The Prototype development included a PCB to host the PICs and provide a clean environment for the data transmissions taking place.

Power

Routing the power distribution for a PCB is crucial to ensure voltage stability and minimise noise within a PCB. Figure 15 shows typical four-layer PCB layouts, the use of power and ground planes in multi-layer PCBs reduces voltage drop and noise boosting overall performance. Keeping planes close together minimises loop inductance to improve power delivery. If planes are not possible, thick traces should be used to aid in producing the same results.^[27]



When distributing power to multiple components there are two main topologies to consider – Star or Tree. Each is a method of minimising noise coupling throughout the system and ensure uniform power distribution. While distributing power it is important to consider adding decoupling capacitors or ferrite beads and filters to the board.

Figure 15 - Planes Throughout a PCB

Decoupling capacitors should be placed near the power pins of ICs and other sensitive components to create local charge reserves and high-frequency noise filters by stabilizing voltage levels and enhancing power supply noise resistance. And ferrite beads and filters should be implemented for designs with multiple power domains or sensitive components to isolate noise and reduce EMI effects.^[28]

Signals

When routing signals on a PCB it is important to understand the requirements for each signal type, impedance control, trace length matching, crosstalk, and layer stack up are the main considerations. Signal return paths and termination should be used to ensure signal integrity throughout the PCB design.^[29]

I2C

Based on the data transmission speed the pull up resistors for the data and clock buses can be calculated using the values from the Table 1 and Equation 1 shown below. The pull-up resistors and bus capacitance are the two main considerations when routing I2C buses.

Mode	Data rate / clock speed	Max. rise / fall time	Min. rise / fall time	Directionality
Standard	100 kHz	1000 ns	-	Bidirectional
Fast	400 kHz	300 ns	20 ns*	Bidirectional
Fast-plus	1 MHz	300 ns	20 ns*	Bidirectional
High speed	3.4 MHz (100 pF bus)	120 ns**	15 ns**	Bidirectional
	1.7 MHz (400 pF bus)	240 ns**	30 ns**	
Ultra-fast	5 MHz	50 ns	25 ns	Unidirectional

*Assumes VDD/VCC = 5.5 V. Scales down linearly if VDD/VCC is lower

**Divide these values by 2 for the clock line

Table 1 – I2C Transmission Speeds Table

Selecting the correct pull-up and series resistors, an RC charging/discharging circuit is formed from the pull-up resistors and the capacitance of the V_{DD}/V_{CC}. The resistor (R_P) values must obey the inequality below:

$$\frac{V_{CC} - 0.4V}{0.003A} < R_P < \frac{t_{rise}}{0.8473C_b}$$

Equation 1 - I2C pull-up resistor value

0.8473 is defined for a 30%-70% transition time – standard in I2C specification. For a 10%-90% transition time, replace 0.8473 with 2.2. This will further limit the value of the pull-up resistors.

The capacitance of the bus lines is determined by calculating the V_{CC} bus impedance using the transmission line impedance equation. The bus capacitance is then calculated using the impedance and the propagation delay for the line. Series resistors are not required under the I2C standard, however can be included to protect devices from voltage spikes and slow down the rise/fall times.^[30]

Ensure a ground plane is used in the PCB to reduce noise and interference, keeping signal traces as short as possible will minimize signal loss and interference, decoupling capacitors should be placed near the power pins of each IC to reduce noise and improve stability of output signals.^[31]

SPI

This protocol is like I2C, it requires 4 lines but allows for two modes of component layout. A single controller transmitting to a single device, is simply point-to-point, whereas transmitting to multiple devices depends on the number of chip-select outputs provided by the driver. Typically, the driver will have equal peripherals and chip-select outputs – Standard mode. The second mode – daisy-chaining – relies on the single chip-select successively triggering each device in the daisy chain.

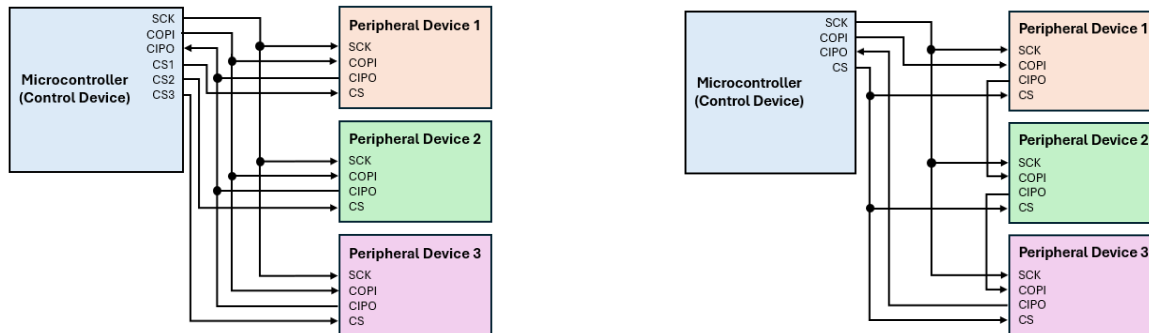


Figure 16 – SPI Standard and Daisy Chain Methods

Unlike routing I2C signals, SPI is highly configurable. Similar RC discharging methods can be implemented to control output current and rise/fall times from interfaces; however, SPI is a push-pull interface and does not draw power through a pull-up resistor. The rise time of a transmission signal is determined by the external capacitance and driver's output impedance.

When routing SPI signal lines it is important to place components close together to reduce the length of traces, ensure that data transfer lines are the same length, copper weight and impedance; the impedance must be consistent for the bus. It is also important to consider the board material, FR-4 is suitable for most speeds, high-speed designs may require more specialised PCB materials.^[32]

While terminating resistors are not necessary for a SPI bus series signal resistors may be included at the outputs of the devices to slow down the transmitted signal and damping any ringing.^[33]

UART

A typical UART termination method is to include series termination to reduce overshoot. UART transmission lines may idle at high or low levels, requiring pull-up resistors to set the desired idle level. Component specifications will detail any need for pull-up resistors.^[30]

Any UART trace layouts should adhere to the following. A series resistor placed on the U0TXD trace should be placed close to the chip and away from the crystal. Both U0TXD and U0RXD traces should be as short as possible. And the UART trace should be surrounded by ground copper or ground via stitching^[34] to reduce noise along the trace.

In-Circuit Serial Programming

This method of programming allows for the code to be uploaded to the PIC while it is in place on the PCB, allowing for the PIC to be soldered in place before having the program uploaded, or allowing for program updates and changes during the PCB lifecycle. There are five pins required for this:

V_{DD} – Supply Voltage

V_{SS} – Ground

PGC/ICSPCLK – Programming Clock

PGD/ICSPDAT – Programming Data

MCLR/ V_{PP} – Programming Voltage^[35]

When writing code to the PIC it must be powered through the power supply pins and a programming voltage of ~13V must be applied to the programming power pin to allow for the PIC to receive the data via the data and clock programming pins.^[36]

The main consideration when designing a circuit with ICSP capabilities:

- Wherever possible avoid using the programming pins of the targeted device in the design or ensure that the application circuit is suitably isolated by high-impedance loads.
- The MCLR/V_{PP} pin must not be directly connected to V_{DD} and the rest of the circuit must be protected from this voltage, use a diode and the reset pull-up resistor in series to separate the 13V from the V_{DD} line.
- If possible, configure the MCLR pin as a reset as a programmer will attempt to reset the PIC to beginning the programming process. "If the MCLR pin is configured as an input, the programmer may have problems putting the chip into program-mode. This is especially problematic if the target has been previously programmed and is using the programming clock or data lines as outputs."^[37]
- Ensure the low-voltage programming pin is pulled to ground, if it is left floating, or high, it may interfere with the programming process.
- Always connect every power pin to the appropriate source because if even one of the pins is not connected it may result in programming errors. This also applies to analog supply pins even there is no analog function.
- Bypass capacitors must be used between V_{DD} and V_{SS}. A single capacitor of 0.1uF is the recommended minimum, typically a larger value is used.

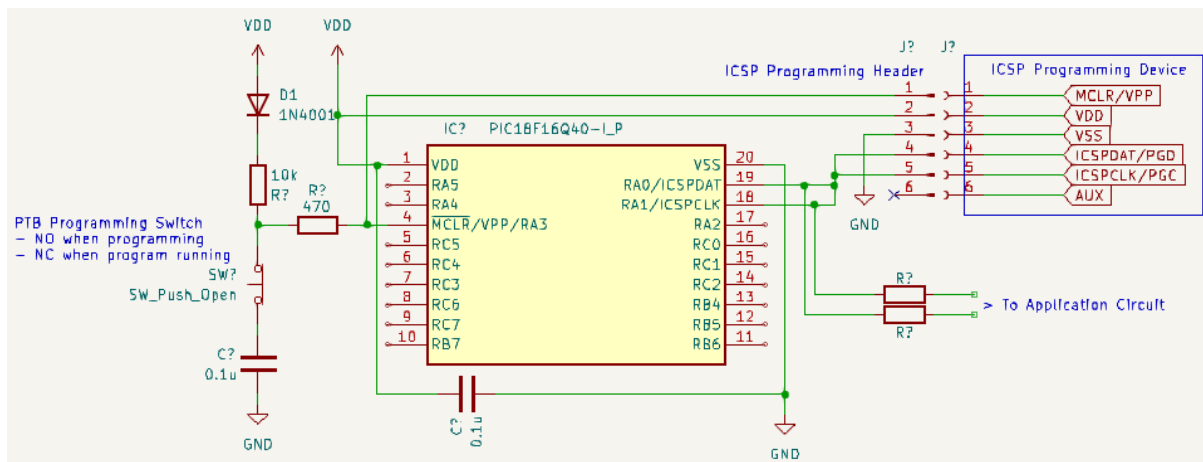


Figure 17 – Potential In-Circuit Serial Programming Layout

Development Methodology

I have chosen to use the MPLAB X IDE and XC8 C language compiler to write the programs as the PICs and provided Real In-Circuit Emulator device are Microchip products, Proteus will be used to simulate the code during development as this software like the ICE device and MPLAB is freely available for use during this project. From this the PCB designs will be carried out on KiCAD for the same reasons and once the fabrication is complete the prototype will be tested in stages building up to a full system test.

Code Development

This project required code developing to program each of the PICs in the control system, the code written is developed from the flow charts shown in figures 20, 21, and 22. The host PIC flowchart shows the data flow from user input on the control laptop to the I2C bus, where the data will be used in the transfer and display PICs. The transfer PIC flowchart shows the how the data is categorised and formatted before it is then retransmitted to the SPI bus or as a UART message, and the display PIC flowchart shows how the data is sorted prior to being displayed for the user.

RF Devices

The figures below show the data protocols required or available for use with each of the devices.

Digital Interface (P1-1, 2, 3, 4)

The MLWI digital driver interface is a standard 3-wire connection compatible with SPI/QSPI/MICROWIRE interfaces. The 3-wire serial interface will operate in a 5V or 3.3V logic system. The chip-select input (SELECTn) frames the serial data loading at the data input pin (DATA). Immediately following SELECTn's high-to-low transition, the data is shifted synchronously and latched into the input register on the rising edge of the serial-clock input (CLOCK). After 16 data bits have been loaded into the serial input register, it transfers its contents to the DAC latch on SELECTn's low-to-high transition (Figure 2). Note that if SELECTn does not remain low during the entire 16 CLOCK cycles, data will be corrupted. In this case, reload the DAC latch with a new 16-bit word.

Figure 18 – YIG Filter Datasheet Screenshot^[38]

6.1 Introduction to Commands

The synthesizer can be programmed using a set of simple commands accessed by the micro-USB serial interface or TTL USER Port and either a terminal emulator program such as Putty, TeraTerm or the recommended 5000term.exe. The 5000term.exe is a download available on the valonrf.com website. A direct digital TTL interface is also available at the 14-pin User Port.

Once connection to the serial port has been established, type "help" in the terminal program for a list of the latest available commands.

Figure 19 – Frequency Synthesiser Datasheet Screenshot^[39]

The RF Switch inputs need to be alternated at a rate between 1ms – 5s due to the Pulse Repetition Frequency (PRF) of the targets. This removes the need for a programmable oscillator as the switch control can come from the Host PIC GPIO directly.

Intended Functionality

Based on the communication protocols available it was decided to structure the data flow around simple, robust and reliable protocols I was familiar with, such as: UART, I²C, and SPI. UART (RS232) provides the communication between the laptop and Host PIC and I²C for data flow between the PICs within the system; then SPI and RS232 send the setup instructions to the Yttrium Iron Garnett (YIG) tuneable bandpass filters and Frequency Synthesiser device from the transfer PIC.

The Host PIC would receive all the instructions from the laptop sent by the user and use known values to assign an identifier to the data before it is written to the I²C bus. The YIG and Synth data is

sent via the bus to the transfer PIC, while the duty cycle value is handled internally on the Host PIC and a pin is switched at a specific rate for the RF switch control. All instructions received by the Host will be sent to the Display PIC over the bus.

Upon receiving the data from the bus, the Transfer PIC uses the identifier (element 0) of the receive array to assign the data to the correct instruction array. Data from the instruction array is then used in the functions which send the instructions to the YIG filters and the Synth device. The identifier is removed at the start of the function, before data is sent; the YIG frequency value requires some processing to turn the float value into a 16bit instruction that can be sent as two bytes to the SPI bus.

Flowcharts

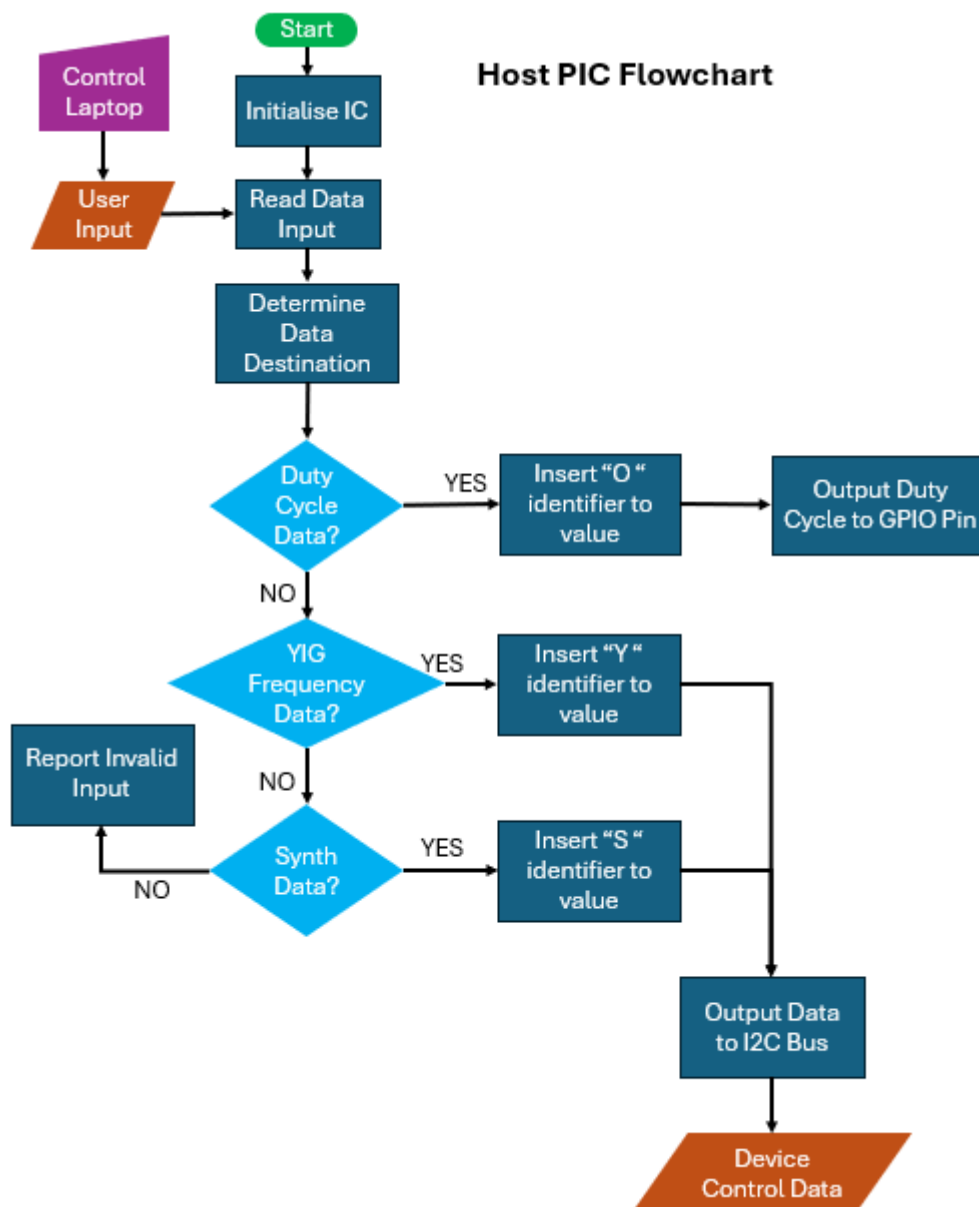


Figure 20 – Host PIC Flowchart

The host PIC receives the data input to the control laptop, determines the intended device and inserts an identifying character before it is then written to the I2C bus.

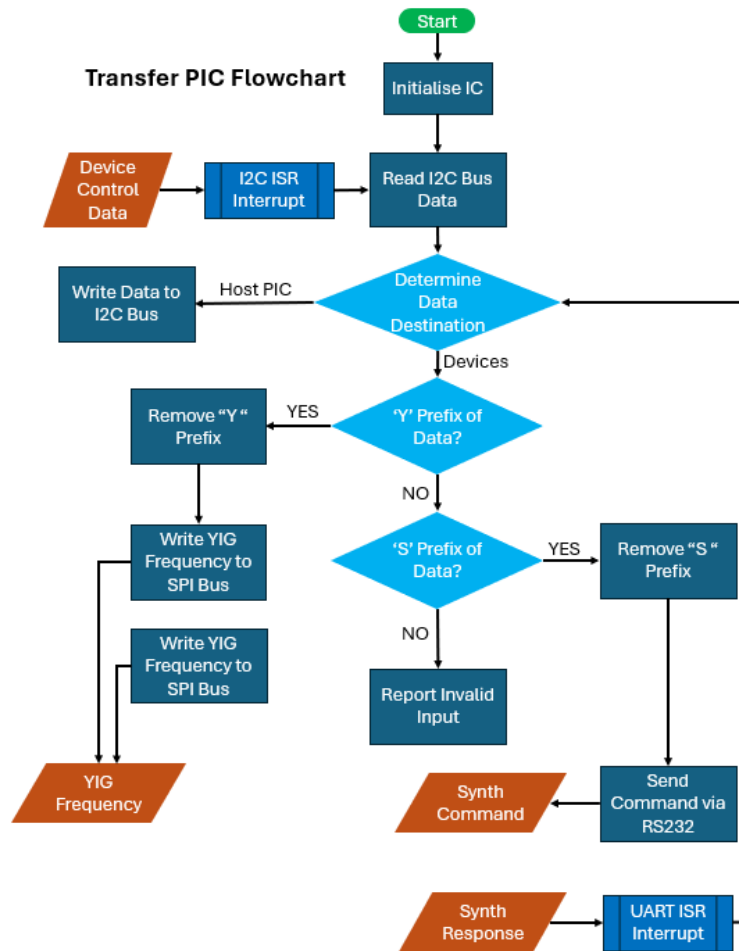


Figure 21 - Transfer PIC Flowchart

The transfer PIC initialises, and an Interrupt Service Routine (ISR) is used to read I2C bus data, then the data identifying token is used to determine the target device. The data is modified to remove the token before it is written to the SPI bus (YIG filters) or sent over UART (Frequency Synthesiser). Any responses from the Frequency Synthesiser are then forwarded back to the Host PIC.

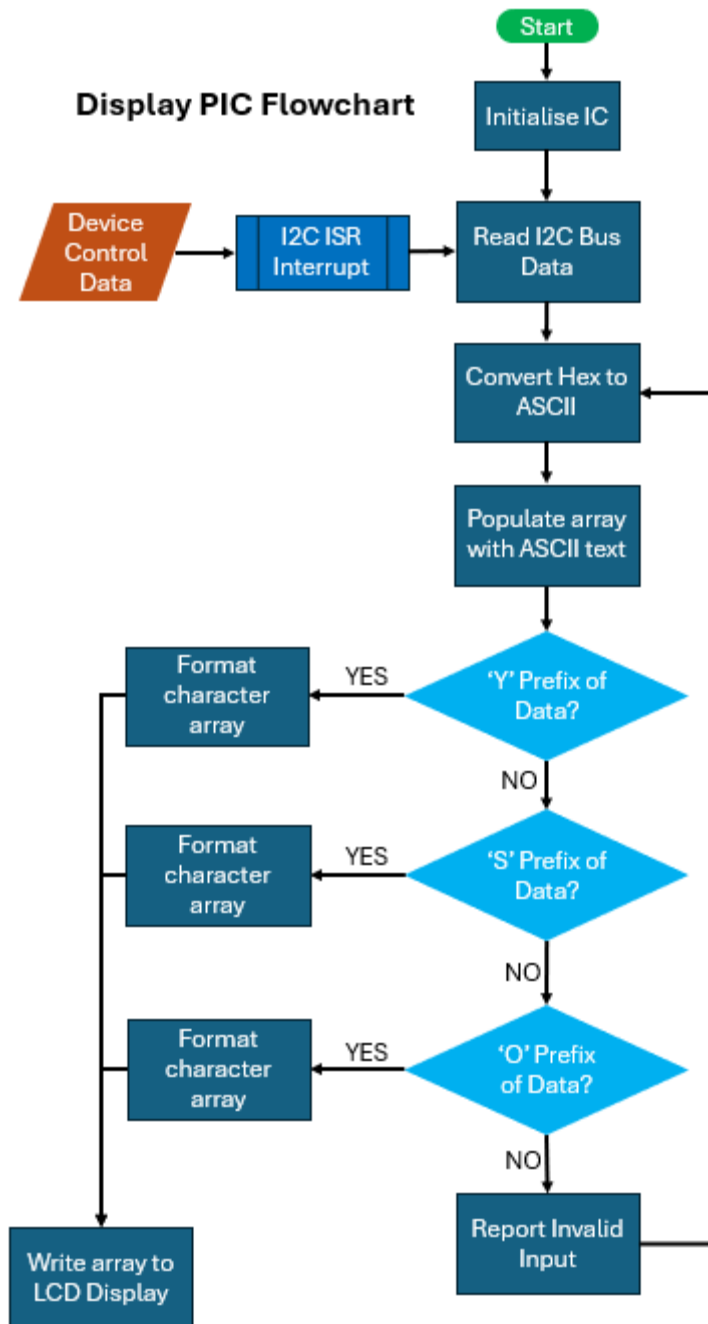


Figure 22 - Transfer PIC Flowchart

The display PIC reads the data from the I2C bus using an ISR interrupt, then the data is converted from Hexadecimal to ASCII characters to populate the character array. The character array is then identified using the data token which is the first element of the character array. The strings are formatted for readability then written to the LCD display.

Integrated Development Environment

Throughout the project, during code development, various core data processing functions were developed along with Interrupt Service Routines to carry out the data transfers and byte handling. The functions include data identification and initiation of data transfers, all the interrupts developed are carrying out byte transmission and reception to handle data transfers between the control laptop, PICs, and RF devices.

Functions

```
I2C_transmit()
void i2c_transmit(char address) {
    PIE1bits.SSP1IE = 1;           // enable interrupt on SSP1 buffer empty
    SSP1CON2bits.SEN = 1;          // set start bit in the register
    i2cMsg[0] = address;           // load the client address into element 0 of the array
}
```

I2C_transmit() function starts the process of transmitting data to the I²C bus. The function starts by enabling the SSP1 interrupt, then the Start Condition Enable bit of the Con2 register is set to initiate the communication on the bus, and finally the address is loaded into element 0 of the *i2cMsg* array. The function then hands over the [Interrupt Service Routine](#).

```
array_length()
int array_length(void) {
    int length = 0;
    for(int i = 0; i < 12; i++) {
        if(rcvBuff[i] == '\0') {
            length = i;
            break;
        }
        else {
            continue;
        }
    }
    return length;
}
```

Array_length() function returns an integer length of the UART receive buffer array. This is done by iterating through the array elementwise until a NULL character is found, upon reaching the NULL the code breaks from the FOR loop and returns the length value to the main loop. The ELSE case ensures the FOR loop continues iterating until a NULL is found.

```
data_identifier()
void data_identifier(int token) {
    token = arrLen - 1;
    if(rcvBuff[token] == 0x25) {
        for(int j = 0; j < 4; j++) {
            dutyCycle[j] = rcvBuff[j];
        }
    }
    else {
        char c = rcvBuff[0];
        if(0x40 < c && c < 0x5B) {
            synthCmd[0] = 0x73;
            for(int k = 0; k <= arrLen; k++) {
                synthCmd[k+1] = rcvBuff[k];
            }
        }
        else if (0x2F < c && c < 0x3A) {
            yigFreq[0] = 0x79;
        }
    }
}
```

```

        for(int l = 0; l <= arrLen; l++) {           // .. then fill array offset by 1 element
            yigFreq[l+1] = rcvBuff[l];
        }
    }
    else {                                           // if not a Capital Letter or number not relevant data
        //cout << "Bad Data" << endl;
        return;
    }
}
}
}

```

Data_identifier() is a critical function in system operation provides identification of received data. Using the array length the token position is identified by subtracting one from the value to find the percentage character preceding the NULL character; this identifies the data as a duty cycle value. Upon identifying the token in the array, the contents of the rcvBuff array are written to the dutyCycle array elementwise. Should the token not be found in the array the ELSE case uses the first element of the rcvBuff array – in the form of the variable c – to sanity check if the character is alphanumeric, if it is not the code exits the routine.

Upon entering the IF statement, the variable c is tested to check if it falls within the range of capitalised letters in the ASCII table, if true (greater than 0x40 (@) and less than 0x5B ([)) the code proceeds into the statement as this is likely to be a command for the Frequency Synthesiser. Inside the statement the synthCmd array is then initialised with an identity token (s) as the first element for ease of identification during transmission over the I²C bus. Next the FOR loop populates the synthCmd array with the data received starting at element 1 so as not to overwrite the identity token. The array is filled until the indexer (k) reaches the arrLen to ensure the NULL character is included as this is needed for the data transmission to the I²C bus.

If the previous statement is found to be false, this process is repeated in the ELSE IF statement to check if the variable c is a number. If c is greater than 0x2F (/) and less than 0x3A (:) it falls within the range of the integer numbers within the ASCII table, this means it must be a YIG filter frequency value. Similarly to the Frequency Synthesiser command, an array called yigFreq is prepared (to temporarily store the value before transmission to the I²C bus) with the identity token (y) written to the first element and the data written to the remaining elements using a for loop in the same fashion as the synthCmd array.

```

        load_i2c_buff()
void load_i2c_buff(void) {
    if(dutyCycle[0] == 0x6F) {                     // if identity token is matched enter statement
        for(dcPtr = 0; dcPtr <= arrLen; dcPtr++) { // using array pointer carry out loop for arrLen
            i2cMsg[dcPtr + 1] = dutyCycle[dcPtr]; // load array from buffer leaving element free for address
        }
        i2cRdy = 1;                               // flag I2C message as ready
    }
    else if(synthCmd[0] == 0x73) {
        for(synthPtr = 0; synthPtr <= arrLen; synthPtr++) {
            i2cMsg[synthPtr + 1] = synthCmd[synthPtr];
        }
        i2cRdy = 1;
    }
    else if(yigFreq[0] == 0x79) {
        for(yigPtr = 0; yigPtr <= arrLen; yigPtr++) {
            i2cMsg[yigPtr + 1] = yigFreq[yigPtr];
        }
        i2cRdy = 1;
    }
    else {

```

```

    return;
}
}

```

The load I²C buffer function performs the same operation for each temporary storage array. It carries out a final check on the character arrays prior to transmission to the bus, it is checking the first element of all the arrays for the respective identity tokens. If these tokens are identified the data is then loaded into the i2cMsg buffer, which is the final temporary data storage as this buffer empties into the SSP1 buffer.

```

    data_sorting()
void data_sorting(char id) {
    switch(id) {
        case 's':
            // use identity token in i2c message
            // if ID is 's'
            for(synthPtr = 0; synthPtr <= arrLen; synthPtr++) { // use array pointer to loop for arrLen
                synthCmd[synthPtr] = i2cMSG[synthPtr+2]; // fill char array from i2cMsg buffer exclude addr + ID
                i2cMSG[synthPtr] = 0x00; // reset pointer
            }
            i2cMSG[0] = 0x00;
            i2cMSG[1] = 0x00;
            synthPtr = 0;
            break;
        case 'y':
            for(yigPtr = 0; yigPtr <= arrLen; yigPtr++) {
                yigFreq[yigPtr] = i2cMSG[yigPtr+2];
                i2cMSG[yigPtr] = 0x00;
            }
            i2cMSG[0] = 0x00;
            i2cMSG[1] = 0x00;
            yigPtr = 0;
            break;
        default:
            return;
            break;
    }
}

```

Data_sorting() function parses the second element of the i2cMSG array to identify and then sort the incoming data correctly. During [i2cRxISR\(\)](#) all received bytes are read from the SSP1 buffer into the i2cMSG array for stop-gap storage while the data is identified, it then proceeds into one of the cases where each byte is written into the appropriate data storage array. Once the identifying token of 's' or 'y' is matched the FOR loop writes the data out of the i2cMSG buffer from element two as this removes the client address and token character and reads this into the first element of the chosen array.

```

    convert_2_float()
double convert_2_float() {
    double f = strtod(yigFreq, NULL);
    if(f < 6 || f > 18) { // sense check the calculated value
        f = 6; // if f < 6 or f > 18 set f as 6 (boot freq)
    }
    else {
        f = f; // if okay allow f value
    }
    return f;
}

```

Convert_2_float() function converts the character array into a float value number using the strtod() function. it reads from the start of the yigFreq array until it hits a NULL character and returns the

value as a double. The IF statement acts as a validation check of the value to determine if it is outside the range of the filter; when the value is outside the 6-18 range *f* is set to 6 as this is the boot frequency of the YIG filters.

```

    yig_freq_instr()
void yig_freq_instr(double freq1) {                // optional second freq
    double yig1_16bit = 65535 * ((freq1 - 6) / (18 - 6)); // GHz freq to 16bit value equation / 6 and 18GHz are the max
    and min freq
    //double YIG2_16bit = 65535 * ((freq2 - 6) / (18 - 6)); // optional second freq equation, duplicate process with YIG2
    identifier for multiple YIGs @ diff Freqs
    short yig1_16bit_round = round(yig1_16bit);
    char yig1_upper = (yig1_16bit_round >> 8);      // separate rounded value (8 MSBs)
    char yig1_lower = (yig1_16bit_round >> 0);      // separate rounded value (8 LSBs)
    yigInstr[0] = yig1_upper;                       // Load 8 MSBs into first position of array
    yigInstr[1] = yig1_lower;                       // Then load 8 LSBs into next position
}

```

Yig_freq_instr() function reads in the frequency value as a double and converts it to a 16bit integer using the equation – $65535 \cdot \left(\frac{freq-6}{18-6}\right)$ – and then rounding the result to the nearest integer value. Next the 16bit short is split into two bytes by separating the short into the 8 Most Significant Bits and Least Significant Bits. These bytes are then stored in the *yig* instruction array.

```

    spi_transmit()
void spi_transmit(int cs) {
    PIE3bits.SSP2IE = 1;                // enable the interrupt
    yigPtr = 0;                          // ensure pointer is reset
    yigInstr[2] = '\0';                  // append NULL char to yig instruction array
    if(!cs) {                            // while the Chip Select line is held low
        SSP2BUF = yigInstr[yigPtr];      // ... load first byte into shift register
        yigInstr[yigPtr] = 0x00;
        yigPtr++;
    }
}

```

The *spi_transmit()* function is controlled by the chip select lines; it reads the value of the pin as a local variable. When the *yig1Sel* value is set in the main loop it is read into the function – *spi_transmit(yig1Sel)* – to control the loops. While the *cs* variable is set low the code enters the loop to setup the SPI interrupt, this involves enabling the interrupt on Synchronous Serial Port 2, resetting the pointer to 0, and appending a NULL character to the end of the array for the interrupt to break on. Finally, because the chip select line is being held low the first byte is loaded into the shift register; the [interrupt](#) will then take over once the flag is raised as the SSP2BUF is full.

Interrupt Routines

To produce the interrupts required the code had to be written based on the data sheet for the PIC as each protocol requires slightly different initiation and operation, each of the protocols react to an interrupt flag which triggers the Interrupt Service Routine but they then each require different checks and controls to ensure the bytes are handled correctly.

TOISR()

```
void TOISR (void) {                                // interrupt gets called every 100us
    TOCONbits.TMR0ON = 0;                          // stop the timer
    INTCONbits.TMR0IF = 0;                         // clear timer0 interrupt flag
    TMR0H = 0xF6;                                  // rest the timer registers, high first due to 16 bit write
    TMR0L = 0x0C;

    LATCbits.LATC2 = ! LATCbits.LATC2;             // Test Pin

    // SCHEDULE THE HEARBEAT TASK (likely to become master flash in the future)
    hbTaskCtr++;                                   // increment the heartbeat task counter
    if(hbTaskCtr == 0) {                           // check if the counter 0
        schedFlags.heartBeat = 1;                 // perform blink routine
        hbTaskCtr = (unsigned int)0xFE0B;          // restore counter value
    }
    TOCONbits.TMR0ON = 1;                          // set the timer going again
}
```

This interrupt routine runs a blink as a base routine for each PIC to indicate the status, providing an indication of if the PIC is programmed and running correctly. It initially turns off the timer and clears the interrupt flag, then resets the timer registers values. The routine also toggles pin C2 to act as a test pin to indicate interrupt operation. The heartbeat task is carried out when the counter reaches 0, causing the heartbeat bit of the schedule flags union to be set. Finally, the heartbeat counter is reset returning the code from the IF case and setting the timer again.

UARTRxISR()

```
char rcvPtr = 0;
char crRcvd = 0;
char UARTRxISR(void) {
    if(RCSTAbits.FERR == 1) {                      // if there is a frame error
        rcvBuff[rcvPtr] = RCREG1;                  // read the data from the register
        rcvBuff[rcvPtr] = 0x00;                    // overwrite the data
    }
    else {
        if (RCREG1 == '\r') {                      // check for a carriage return if found...
            rcvPtr = 0;                             // reset the pointer
            crRcvd = 1;                             // set value to 1 when carriage return found
        }
        else {
            rcvBuff[rcvPtr] = RCREG1;               // put the next character in the transmit buffer
            rcvPtr++;                                // increment the pointer
        }
    }
    return crRcvd;                                 // return the value to the main loop
}
```

This interrupt reads the incoming data from the RC1REG into a temporary storage buffer called rcvBuff before it is passed to the relevant character array prior to transmission. It starts by establishing if a frame error has been detected based on whether the Frame Error bit in the RCSTA register has been set, if the frame error bit is set the data is read into the rcvBuff and overwritten to remove it.

If no frame error is detected the interrupt must determine if a carriage return has been sent, if not all the data in the shift register is read sequentially into the next free element of the rcvBuff array. This continues until a carriage return is received in the shift register, once this happens the IF statement is true and the interrupt resets the pointer to the start of the array and sets the crRcvd value to 1. The value is returned to the main loop, where it is used to signal the data is ready to be read.

```

    I2CTxISR()
char TxStageCntr = 0;                                // setup a switch expression
void I2CTxISR(void) {
    PIR1bits.SSP1IF = 0;                             // clear the interrupt flag
    switch (TxStageCntr) {
        case 0:
            SSP1BUF = i2cMsg[i2cMsgPtr];              // put the next character in the transmit buffer
            i2cMsgPtr++;                               // increment the pointer
            TxStageCntr++;
            break;
        case 1:
            if(SSP1CON2bits.ACKSTAT == 0) {           // if the ACK is received from client
                if(i2cMsg[i2cMsgPtr] != '\0') {       // check for last bit of data
                    SSP1BUF = i2cMsg[i2cMsgPtr];      // put the next character in the transmit buffer
                    i2cMsgPtr++;                      // increment array element pointer
                }
                else {
                    SSP1CON2bits.PEN = 1;             // send stop bit
                    TxStageCntr++;                    // increment switch expression
                    i2cMsgPtr = 0;                     // reset array element pointer
                }
            }
            else {
                SSP1CON2bits.PEN = 1;                 // send stop bit
                TxStageCntr++;                         // increment switch expression
                i2cMsgPtr = 0;                         // reset array element pointer
            }
            break;
        case 2:
            TxStageCntr = 0;                           // reset switch expression
            PIE1bits.SSP1IE = 0;                       // disable the interrupt
            break;
        default:
            SSP1CON2bits.PEN = 1;                       // send stop
            TxStageCntr = 2;                            // in fail carry out stage 2
    }
}

```

This interrupt works based on a switch statement to change the type of operation that is required of the MSSP for the I2C operation, the case is determined by the TxStageCntr which is incremented or set to specific values at certain points in the routine.

Case 0 sends the first byte of data – the client address – and increments both the counter and pointer. Case 1 is used to determine if an ACK was received. Checking the ACKSTAT bit provides the control for the IF statement, if it is 0 the character is written to the buffer and the pointer is incremented; else the stop bit is sent, the stage counter is incremented, and the pointer is reset.

Case 2 is the exit case, resetting the counter and disabling the interrupt. The default case sends the stop bit and then sets the stage counter to 2.

```

    I2CRxISR()
void I2CRxISR(void) {
    PIR1bits.SSPIF = 0;
    if(inByteCtr == 0) {
        address
        i2cMSG[inByteCtr] = SSP1BUF;
        if (i2cMSG[inByteCtr] == transfAddr) {
            inByteCtr++;
        }
        else {
            processI2C = 2;
        }
        SSPCON1bits.CKP = 1;
    }
    else {
        i2cMSG[inByteCtr] = SSP1BUF;
        inByteCtr++;
        if (inByteCtr == 4) {
            processI2C = 1;
            inByteCtr = 0;
        }
        SSPCON1bits.CKP = 1;
    }
}

```

This interrupt clears the interrupt flag and uses the input Byte counter to establish if the byte received is the address or data. The buffer is then emptied into the i2cMSG array, it is then compared with the device address to ensure that the address in the buffer matches that client address; the counter is then incremented, else the process variable is set to 2 to indicate the unmatched address. The final step is to release the clock line to return control to the host.

If the input Byte counter is greater than 0 each byte is read from the buffer into the i2cMSG array and the pointer incremented until the counter reaches a specific integer where the process is set to 1 and the byte counter is reset, and the clock line control is returned to the host.

```

    UARTRxISR()
void UARTRxISR(void) {
    if (synthCmd[synthPtr] == '\0') {
        PIE1bits.TX1IE = 0;
        synthPtr = 0;
    }
    else {
        TXREG1 = synthCmd[synthPtr];
        synthCmd[synthPtr] = 0x00;
        synthPtr++;
    }
}

```

This interrupt is generated when the EUSART interrupt is enabled, and the first character is written:

```

    PIE1bits.TX1IE = 1;
    TXREG1 = synthCmd[synthPtr];
    synthCmd[synthPtr] = 0x00;
    synthPtr++;

```

The data transmission is started by writing the first character from the array to the Tx Register; this causes the interrupt flag to be set, which allows the Interrupt routine to take over. The synthPtr must be incremented to prevent the first character being retransmitted.

Once inside the routine an IF statement is used to determine if the next value received is a NULL. While the data within the synthCmd array is valid it is continuously written into the TXREG buffer, to be transmitted to the Frequency Synthesiser, the data transmission ceases once the synthPtr variable is pointing to a NULL. When the NULL triggers the IF statement the interrupt is disabled to prevent it running again and the synthPtr is reset.

SPITxISR()

```
char spiEnd = 0;
void SPITxISR(void) {
    PIR3bits.SSP2IF = 0;           // reset interrupt flag
    if (yigPtr == 2) {             // check to make sure the last character in the message hasn't been sent, if not...
        yigPtr = 0;                // ...and the pointer
        spiEnd = 1;                // flag to rest of code interrupt finished
        PIE3bits.SSP2IE = 0;       // disable the interrupt
    }
    else {
        SSP2BUF = yigInstr[yigPtr]; // put the next character in the transmit buffer
        yigInstr[yigPtr] = 0x00;    // clear the element
        yigPtr++;                   // increment the pointer
    }
}
```

This interrupt (like the UART Tx interrupt above) is initiated by the transmission of the first byte which takes place in the [function](#). As the YIG filters use a standard instruction I can use a more deterministic method to transmit the instruction to the SPI bus, the 16-bit instruction prepared when it is split into two bytes stored in the yigInstr array. I am then able to use the yigPtr value in the IF statement to identify which of the bytes has been transmitted to the bus and by adding the third byte to the array I am then able to terminate the interrupt when that element is reached, ensuring only the two bytes are transmitted to recreate the 16-bit instruction generated previously in the code.

Simulation

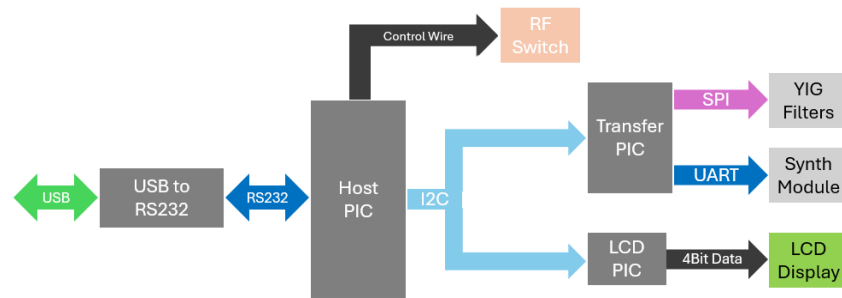


Figure 23 – Block Diagram

Development Kit

PIC18F16Q40

In lieu of being able to simulate my selected microcontroller on Proteus I have elected to procure PIC18F16Q40 Curiosity Nano Evaluation Kit[40], to allow for rapid code development and assessment. These development boards were utilised to aid the simulation of the system being designed as the boards have inbuilt PICKit™ programming capability, and on-board voltage regulation allowing for a simple breadboard layout while developing the code required to run the system.

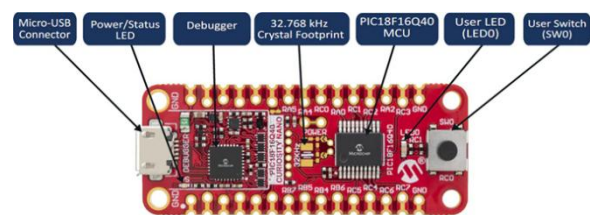


Figure 24 – Curiosity Nano Evaluation Kit

PIC18F46K22

Having spent a large amount of time trying to setup the 16Q40 for I2C communication it was established that the architecture of this pic is different, using dedicated I2C pins rather than a MSSP port like many older PICs. The 16Q40 I2C port has a control register (for the associated pins) where the slew rate, pull-up selection, and the input threshold are selected. I believe that I was unable to setup this port correctly likely due to the pull-up resistor selection and as no output was received at the pins.

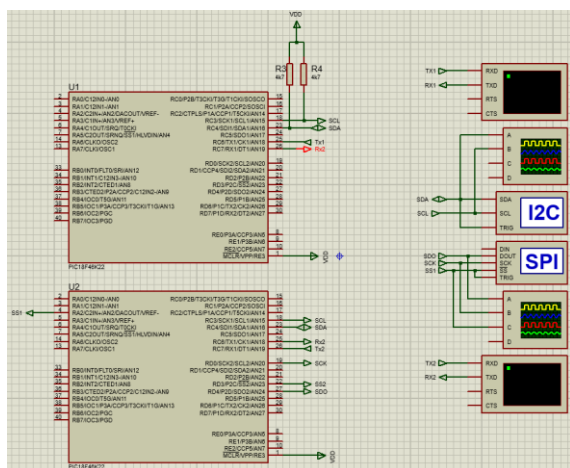


Figure 25 – Proteus Simulation

over I2C with the first PIC receiving instructions from UART and the second PIC outputting instructions over SPI and UART. This is to represent the laptop input to the Host PIC, then the internal communication within the system, and finally the output of the system from the transfer PIC. Due to limitations in the XC8 compiler in Proteus only an I²C test transmission was able to be simulated, no further code could be developed using this simulation technique.

This discovery and some advice from a more experienced PIC programmer helped inform the decision to change the PIC to an older model – the 18F46K22. Swapping to this PIC allowed me to simulate on the Proteus simulation software, and having procured some of the PICs test of the code iterations could be resumed using a [MPLAB Real ICE](#) device and a [Universal Programming Module](#).

The Proteus simulations were setup to emulate two PIC devices communicating with each other

Stripboard Translation

LCD Display

Figure 29, below, shows the LCD display board made to ensure the test setup was more robust than when it was connected with flying wires into a breadboard, as this caused issues with power and data to and from the display. This was comprised of a header socket at the top to plug in the LCD and a row of pins and sockets to connect any type jumper cables to, the board could be powered externally with some flying power cables and the potentiometer is included for contrast control. It is designed as a 4bit display (the solder bridge grounds data pins 0-3) as this is how the library operates. This Veroboard test setup enabled me to reliably debug and test the LCD each time I used it.

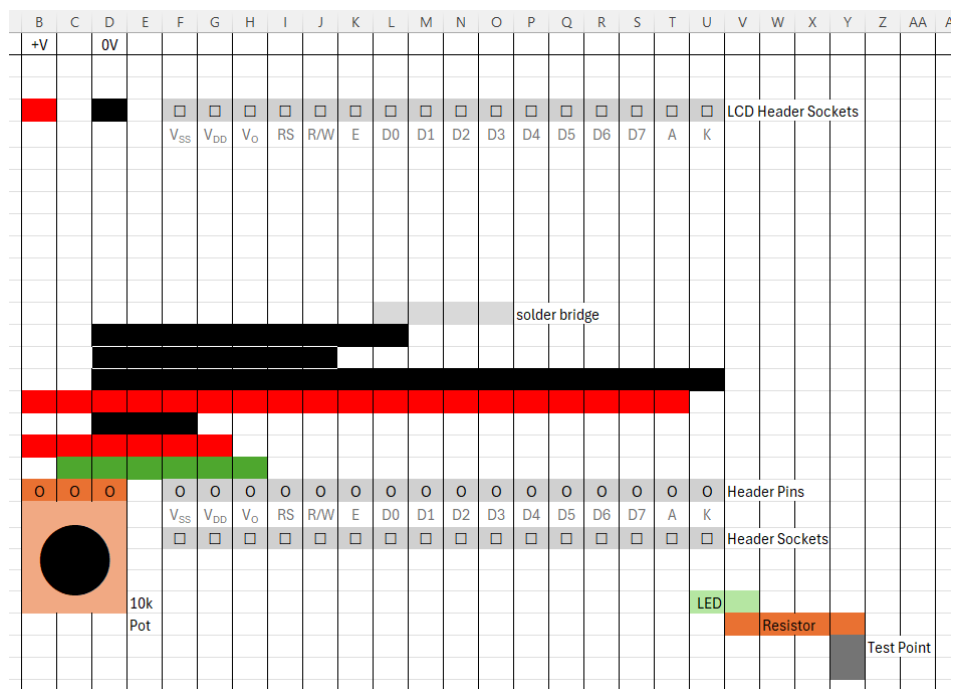


Figure 28 - LCD Testing Stripboard

Power Sensing Circuitry

The data sheet for the YIG filters indicates there is a tight tolerance on the voltage for the driver input – “+15 V ± .5 V @ Filter Tuning Current +50 mA, Max. and -15 V ± .5 V @ 50 mA”[38] – this is within the output tolerance of the STMicroelectronics linear voltage regulators that I am using to power the YIG filters, both the L7815 and L7915 have output ranges from ±4.8 – ±5.2V, meaning that potentially over or undervoltage could occur during the operation of the filter damaging the driver circuitry.

Having established that the power tolerance of the linear voltage regulators was outside the tolerance required by the filters, I decided to develop a voltage protection circuit for the driver circuitry. This would use an LM393 comparator Operational Amplifier and trimmable potentiometers to sense an under or overvoltage event and disconnect the power to the filter to protect it as they are expensive and take a long time to procure. Figure 30 shows the Veroboard translation of the voltage protection circuit, this circuit served to test the positive and negative supply lines and establish if this circuit could act to sense the voltage and disconnect the power if it dropped below or rose above the filter tolerance.

This circuit uses the output of the trim pot to compare the sensed voltage, supplied by the voltage regulators, to the reference voltage. The Zener diode provides a 3V reference at the Op Amp for the trim pot output to be compared with, the high set op amp output is set to 0V when the sensed voltage at the inverting input exceeds the reference at the non-inverting input, allowing the relay to be switched open. The low set op amp output is set to 0V when the sensed voltage at the non-inverting input falls below the reference voltage at the inverting input; this again switches the relay open circuit.

[illegible]

Figure 29 - LM393 based under and over voltage sensing circuit

This test circuit worked in the positive half of the supply circuit but was incorrectly reference for the negative half of the supply. As such this circuit would require a split voltage operational amplifier capable of operating across $\pm 18V$ – LM139 – this would allow the circuit to be tripped should either rail of the filter supply be outside the tolerance set out in the datasheet.

Printed Circuit Board

Control Board

Based on the initial research carried out to start this project the control PCB was designed as a four-layer circuit board (as per [Figure 15](#)) comprised of a top copper signal layer, a ground plane underneath on top of the power plane, and with a bottom copper ground plane. In line with best practices where possible as many of the signal traces as possible have been kept on the signal layer, however in some cases some data signal and other traces had to be routed to the bottom copper ground layer through vias. This was done as they were unable to be routed through the top layer, these traces have been routed to the bottom copper because it is intended as a power ground plane whereas the inner layer ground plane is intended for signal integrity and ideally should remain unbroken to improve signal quality, the bottom copper was selected to route the extra signals to minimise the effect of cross talk that would occur if the traces were routed through the inner ground plane.

The PCB was designed using the application notes on the datasheet as such powerline capacitors have been included for supply stabilisation and noise mitigation as the microcontrollers are performing several data transmissions meaning noise needs to be reduced as much as possible at the source as well as across the PCB. In a similar vein oscillator crystals have been included in the design to ensure a stable and accurate clock signal is provided for baud rate generation, this is especially important for asynchronous communication protocols as this ensures the microcontroller closely matches the desired baud rate of the devices it is communicating with.

As this is a prototype PCB all the functionality of the Frequency Synthesiser has been included by placing a header connection for the User Port to allow for total control during testing, aiding in development, as this will enable the user to understand what functionality is useful and what is not needed in the future final product. There are also LEDs on the PCB with varied intended purposes; a power LED indicates the board is powered, at least one LED per PIC to be operated by the 'Blinky' operation indicating PIC functionality, and indicator LEDs for the outputs of the synthesiser User Port. The remaining LEDs allow for future development and can be programmed as desired.

Control of RF Components within a Passive RF Tracking System

Circuit Schematic

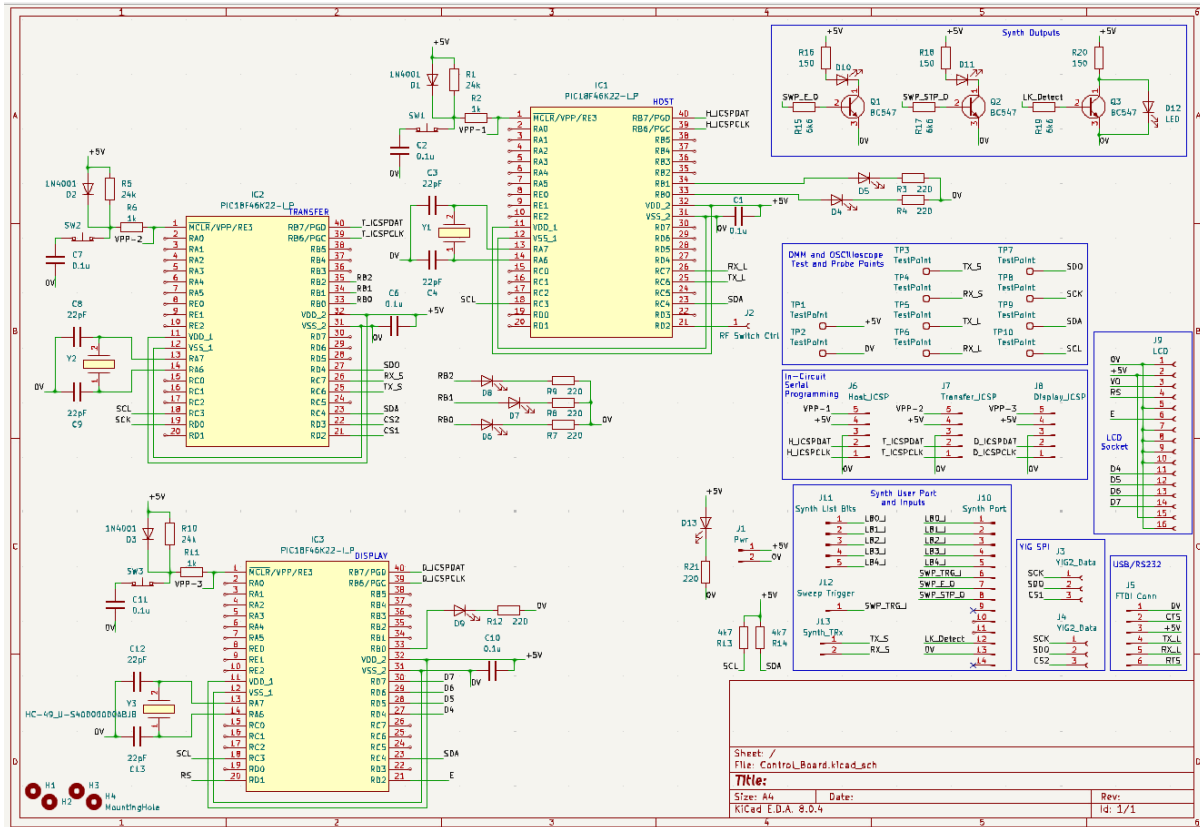


Figure 30 – Control Board Circuit Schematic

Top Copper

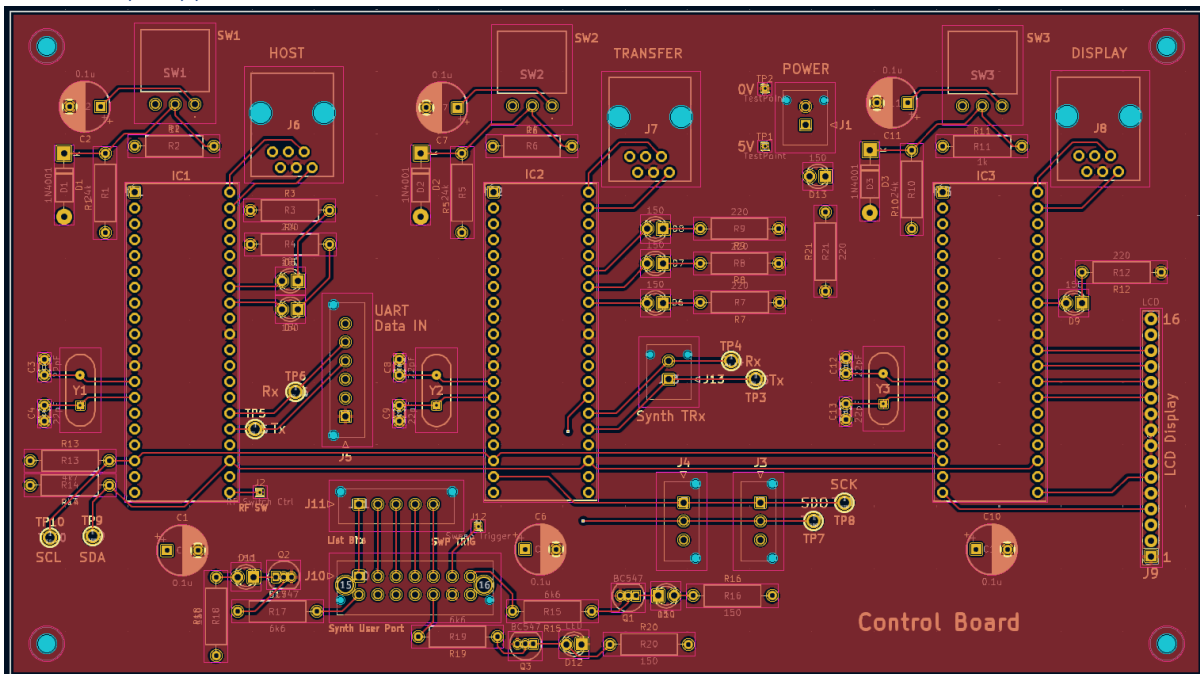


Figure 31 – Control Board PCB Top Copper Layer

Control of RF Components within a Passive RF Tracking System

Ground Plane

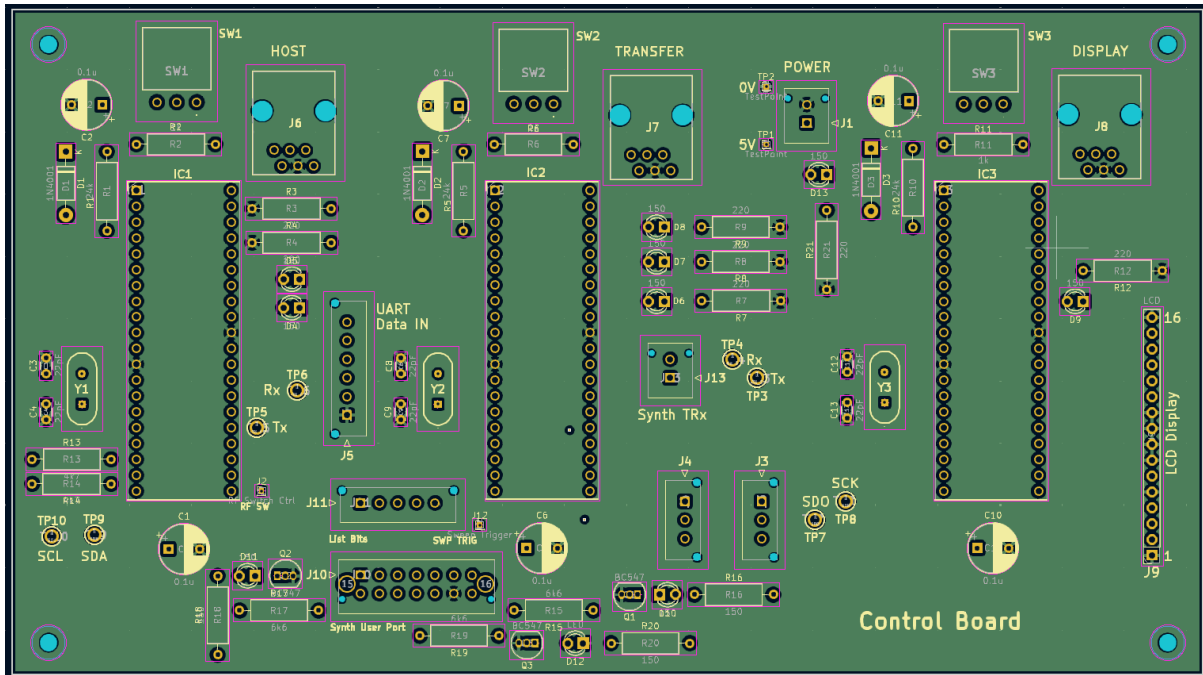


Figure 32 – Control Board PCB Ground Plane Layer

Power Plane

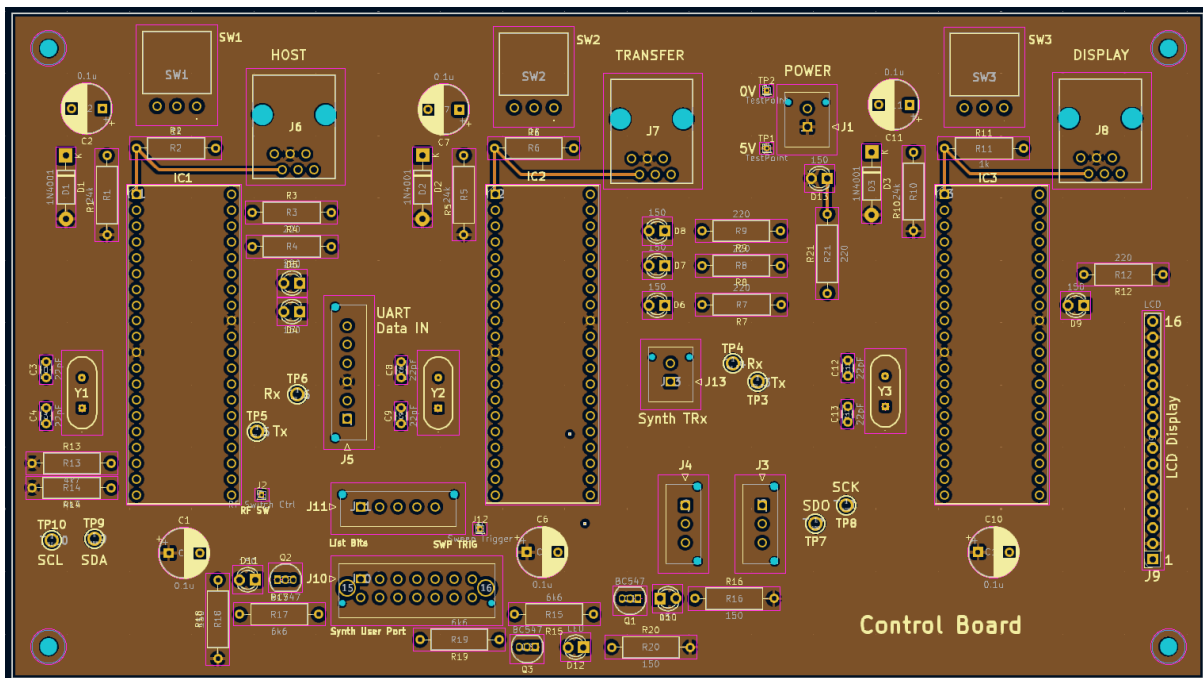


Figure 33 – Control Board PCB 5V Power Plane Layer

Bottom Copper



Figure 35 – 3D View of Designed PCB

Design Rules Checker

Figure 34 shows the output report from the design rules checker tool, as shown below the only errors flagged were a pad intersection and an incomplete thermal relief connection and multiple warnings about isolated copper fill areas. The pad error has been ignored because the footprint is a manufacturer standard and the thermal relief issues is on the header pins for an LCD display, as such thermal relief is not needed. The incomplete fill areas are being used to reduce the amount of chemicals required to produce the PCB and therefore the chemical waste, meaning ignore the warnings have been ignored in this case.

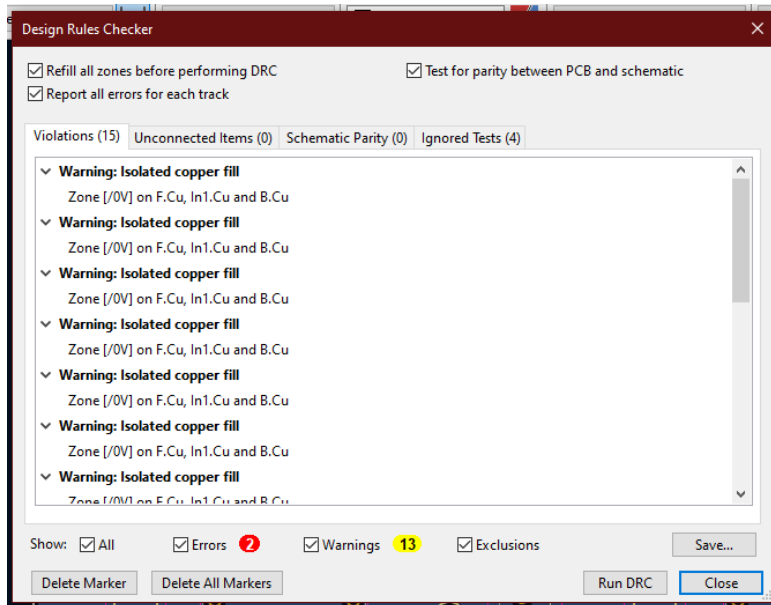


Figure 36 – Design Rules Checker Output

Prototype Testing

Through hole component footprints have been selected as this allows for simple circuit population and assembly requiring no specialist tools, and through hole components can be swapped more easily than surface mount if any are damaged during testing. Many test and probe points have been included in the design throughout the circuit to enable many parts of the circuit to be monitored; the data signals all have probe loops so they can be observed on an oscilloscope and each power input and output can be monitored through the pads on the PCB, the power test points are left as pads to allow the option to solder in header pins or sockets or leave them bare depending on the preference in testing equipment and methodologies.

The MCP2221A IC was removed from the design of the prototype in favour of an FTDI USB-RS232 converter cable, as such the complexity of the design was greatly reduced. This decision was made to remove any error that may occur from incorrect implementation of the IC and to reduce the number of components required for the PCB, allowing for a smaller PCB design with a more consistent data input. A dedicated converter cable removes uncertainty in the data input (from the control laptop) received at the Host PIC and allows for reliable repeatability of transmissions during operation.

The functionality of the prototype was tested in stages building up the capability slowly and making changes where necessary, the aim of testing was to establish the capability and reliability of the data transmission throughout the PCB. The main initial testing revolved around ensuring the interrupts carrying out the data transmissions were reliable and accurate during operation. Each PIC was validated as functioning correctly in isolation before the system is slowly built up and tested further.

This testing methodology ensure that each data protocol interrupt operated correctly and reliably before allowing for a full system integration acceptance test. Having established the operation of the routines the data formatting functions were then further developed to identify the incoming and outgoing instructions. The data formatting functions could not be fully tested in isolation as understanding how the bytes were written into buffers of the receiving PIC was more important and informative.

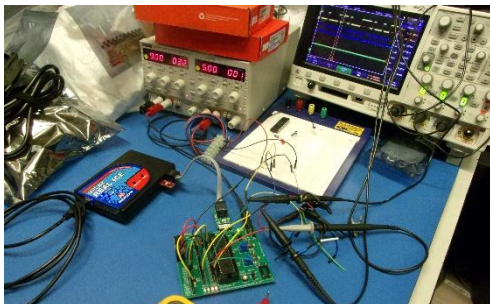


Figure 37 – Bench Testing

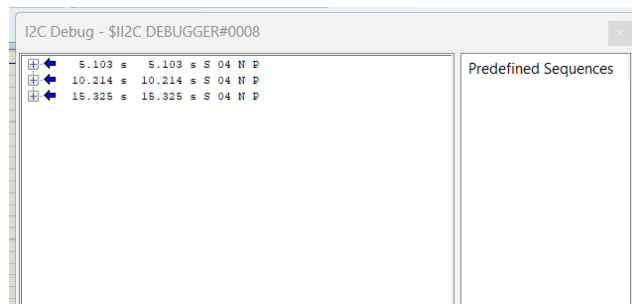


Figure 38 – Proteus I2C Debugger

The Host PIC needs to receive the UART input, modify the all the data appropriately, and retransmit the YIG filter frequency, Frequency Synthesiser commands (via the I²C bus) to both the Transfer and Display PICs and the duty cycle value to the Display PIC only.

Upon receiving the data from the I²C bus the Transfer PIC needs to use the token to identify the data and establish which device instruction it is dealing with. The Transfer PIC also needs to be able to sense check what it has received in an effort to not damage the RF devices by sending data that could cause damage to the device in unexpected ways. The transfer PIC is capable of checking the frequency value for the YIG filters to ensure it does not fall outside the 6-18GHz range the filters are

capable of, but I have not found a sensible method of checking the Frequency Synthesiser commands. As it stands, I am relying on the device itself to match the incoming instruction against its own known database of commands.

To test the data transmission of the system I broke it up into smaller parts to be tested individually before a complete data transmission through the system. The first part is the Host PIC receiving inverted (idle low) UART data and correctly identifying what device data has been received, this is checked by identifying which letter is given as a token to the data being written to the I²C bus. Next, I must ensure the transfer PIC identifies the token and correctly distinguishes which device the data needs to be transferred to.

During testing the prototype functionality, I rewrote some functions and changed the operation of some interrupts. This was to optimise the code I had initially written as some of the checks and case statements became irrelevant as I was combining sections of code and developing my understanding of how the code functioned.

For example, the UARTRxISR() was changed from the [initial approach](#) as a switch case was not needed but a flag to identify the end of the message was required, this was because the program runs faster than a user would be able to type. Meaning that the interrupt had captured the first byte and sent it through the main body of code before another byte could be received, as such the full input was never captured. To resolve this the carriage return received variable was added to act as a flag returned to the main loop from the interrupt to identify when the full input had been written to the rcvBuff and could be used. The spi_transmit() function was converted from a function to an interrupt routine as it was inefficient by wasting processing polling a WHILE loop to establish the status of the buffer to check when it could send the next byte.

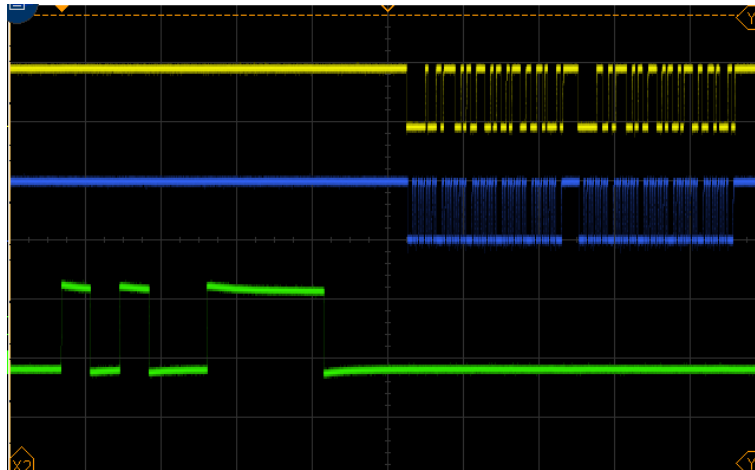


Figure 39 – UART Final Character and I2C Transmission

Having rewritten the code to operate more efficiently I was able to perform a complete data transmission from the control laptop to a transmission to the SPI bus. Figure 39 shows the carriage return character received by the RCREG alongside the full message that is transmitted to the I²C bus, each character typed into the command line is captured by the interrupt and read into the receive buffer for temporary storage. The characters are read until the interrupt routine is terminated upon receiving a carriage return, a flag is then set as the routine is exited which is used in the main loop to trigger the data formatting functions.

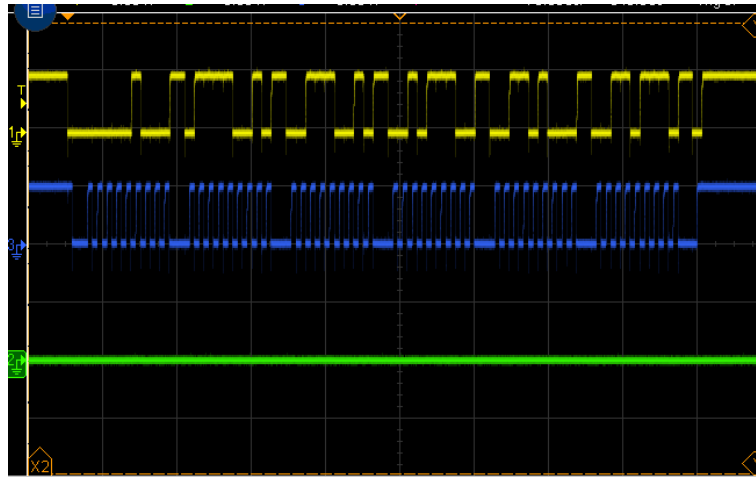


Figure 40 – I2C Frame Data

Figure 40 shows the I²C data transmission to the Transfer PIC via the bus, the data sent is the address (0x04), the inserted identity token (y) and the characters of the input frequency. The characters typed into the command line were 16.7 to represent a desired centre frequency, during code formatting the identity token is inserted to the yigFreq array, and prior to transmission the address is appended to the i2cMsg buffer. This final buffer writes each character into the SSP1 Buffer as a hexadecimal byte based on the ASCII table, the characters are as follows: 'y' – 0x79, '1' – 0x31, '6' – 0x36, '.' – 0x2E, '7' – 0x37.

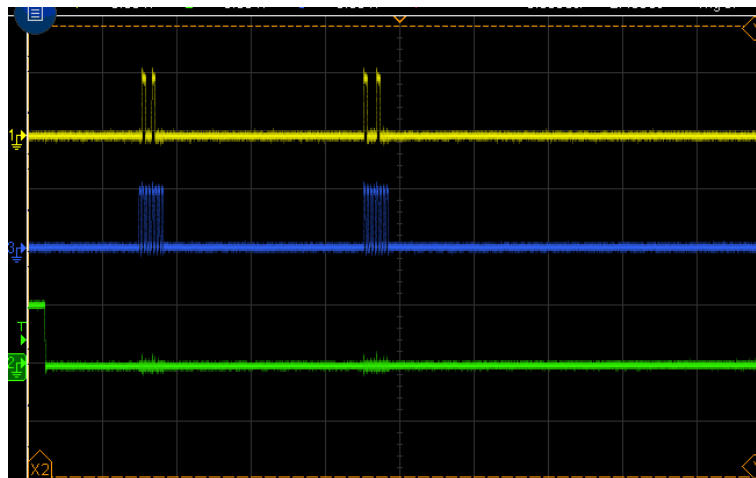


Figure 41 – SPI Data Frame

Figure 41 shows the SPI transmission of the Transfer PIC during a separate test of the data transmissions I set the representative desired frequency to 9.4 producing the bus transmission displayed. The 16-bit instruction required by the YIG filter drivers is shown as two bytes of data on the bus and the chip select line can be seen in green below, the instruction generated by the code is 0x4888.

Project Management

Initially a project Gantt chart was produced to identify key stages of the timeline of events and provide an approximate scale for each process. The [Gantt chart](#) produced during the proposal stage and the [actual timeline](#) of events updated throughout the project can be found in the appendix to show the comparison of the expected and actual task schedules and required processes.

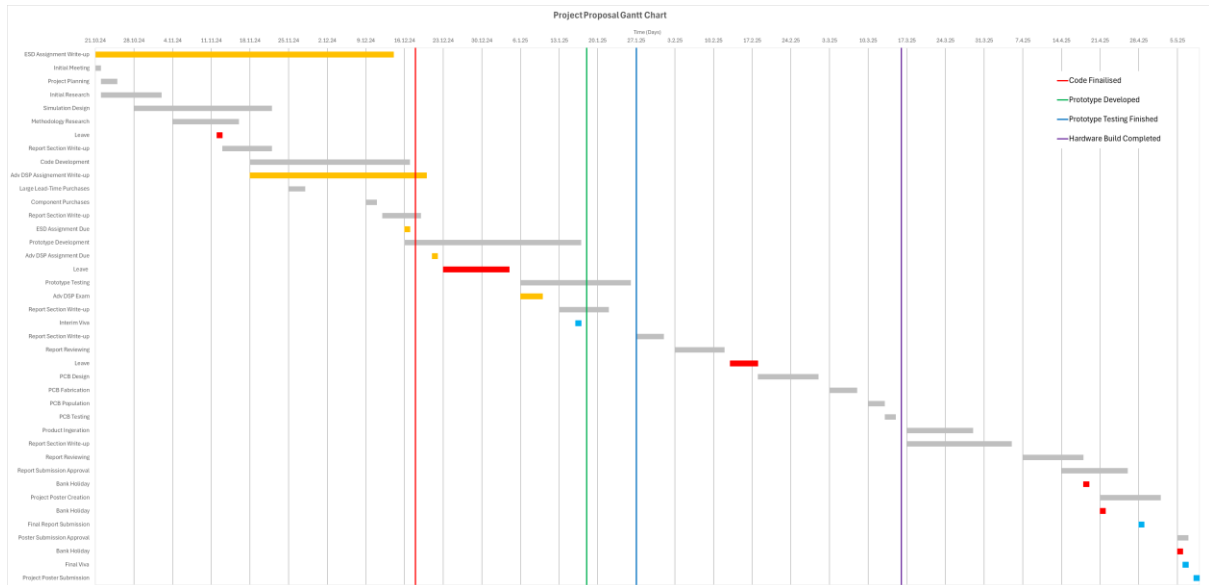


Figure 42 – Proposed Gantt Chart

The biggest concerns were time management issues, such as tasks overrunning, scope creep and dependencies. And other big concerns for the project were skill gaps, integration issues and single points of failure which all needed to be minimised as much as possible to ensure that they didn't become a root cause of the time management issues and contribute to causing the project to overrun the allocated time.

Ethical Concerns

Some thought had to be given to ensuring that legal requirements and ethical standards were met and maintained throughout the duration of this project. These considerations were for the suppliers used during the production of the prototype and for the disposal of the product at the end of its lifecycle.

The suppliers used to order components and tooling and fabricate PCBs had to be reputable and approved for use by the organisation; meaning they have to abide by health and safety legislation and ethical practices. For example, the responsible sourcing of materials and workforce for the production of the equipment as well as the responsible disposal of waste material and chemicals. Another consideration was for the shipping distances of the components being ordered; therefore, a local PCB manufacturer was used to fabricate the PCBs and European electronic components suppliers were utilised wherever possible to reduce the shipping distances and the pollution generated.

Another consideration is for the use and potential misuse of the system under development. The system has been designed with a defined use case (controlling RF devices) and therefore has been implemented in such a way as to ensure that only a specific set of instructions can be given (to the devices) this ensures that the control system is unable to be used for any other purpose.

Control of RF Components within a Passive RF Tracking System

Project Risks

Risk Category	Risks Identified	Probability	Severity	RPN	Mitigations
Technical Risks	Design and Feasibility Issues Ensuring the design is achievable with available resources	3	2	6	Ensure that the designs laid out for the project are feasible and have been approved by a technical authority.
	Technology Limitations Using appropriate tools and technologies that you can manage	1	1	1	Ensure that the appropriate and up to date technologies are available for the duration of the project.
	Integration Problems Making sure all components work together properly	4	3	12	This must be started as early as possible in the project lifecycle to ensure any issues that arise from integration.
Time and Resources	Scope Creep Defining clear project boundaries to avoid endless modifications	4	2	8	Having a clearly defined project brief and regular meetings will allow for the project scope to be set and maintained with minimal project inflation.
	Time Management Balancing the project with other commitments	5	3	15	Time will be managed based off Gantt charts and feedback from project update meetings. Project tasks will be kept separate as much as possible to allow for them to be completed in parallel reducing dependencies.
	Material and Equipment Availability Ensuring you have the necessary parts/tools	3	2	6	Ensure all of the equipment required for the project are procured early to ensure they are available ahead of being required and all components with long lead times are identified and ordered far earlier.
Financial Risks	Budget Overruns Managing costs effectively and avoiding unnecessary expenses	2	2	4	All expenditure will be in line with the corporate policy to ensure best value for money, all procurement will be through approved suppliers and if that is not possible then three quotes must be provided from alternate suppliers.
	Unexpected Costs Accounting for repairs, rework, or additional purchases	4	2	8	An amount of the budget should be allocated for an unexpected purchase as this may arise from component damage, extra PCB manufacturing due to redesigns or costs due to extra work needing to be carried out.
Operational Risks	Skill Gaps Identifying areas where additional learning or outsourcing may be needed	3	4	12	Ensuring that any tasks requiring specialist skills are identified so that any relevant training can be arranged before the task is carried out allowing for any skill gaps to be filled.

Control of RF Components within a Passive RF Tracking System

	Testing and Validation Ensuring proper testing to prevent failures	2	2	4	Making a detailed test procedure to establish the equipment and tests required. This allows for any equipment required to be identified and purchased early in the process.
	Single Point of Failure Avoiding reliance on one critical component or method	3	4	12	Any single point of failures should be identified early to allow for plans to be created and implemented should one be identified allowing for alternative methods or removing dependencies causing the single point of failure.
Safety and Compliance Risks	Workplace Safety Using proper protective gear and following safety procedures	2	1	2	Follow all risk assessments and best practices for the task and workspace to ensure personal safety and the safety of others working nearby.
	Legal & Regulatory Considerations Ensuring compliance with relevant codes	1	1	1	All work carried out will in accordance with all of the relevant health and safety regulations and International and British Standards. All components and manufacturing will be procured through approved suppliers that are known to follow the same or similar regulations.
External and Environmental Risks	Weather & Location Factors If the project involves outdoor work	1	1	1	This is negligible as no of the work will be carried out outdoors with exposure to the elements and all location specific risks are detailed in the workplace risk assessment.
	Supply Chain Delays If waiting for parts or tools from suppliers	3	2	6	Supply chain delays can be minimised by ensuring that all components are ordered early in the process to avoid long lead times and components that are in stock will be favoured wherever possible.

Table 2 – Table of Identified Project Risks

See [Appendix](#) for risk priority number tables

Control of RF Components within a Passive RF Tracking System

Risk Assessment

Hazard	Possible Effects / Harm	Resources Impacted	Controls Measures	Residual Risk Rating		
				Likelihood	Severity	L, M, H
Access / Egress Staff unable to leave in emergency due to injury	Staff stuck in the building during an incident. Potential distress Potential further injury	Staff suffering disability which impacts their reaction to an incident	All staff training provided Line management to provide specialised personalised risk assessments for any staff with disabilities	1	B	L
Emergency Response Evacuation Lockdown	Staff unaware of emergency response procedures may act incorrectly in an incident	All staff	All staff training. Briefing overarching site measures, primarily on emergency procedure Drills of emergency procedure response	1	C	L
Slips, Trips and Falls Cables, spills, obstructed walkways	Tripping over loose cables, bags, boxes, and other walkway obstructions	Lab users Contractors Visitors Equipment being carried	Power sockets rooted to most benches to minimise need for trailing extension cables. All cables in walkways must be taped down with contrasting coloured tape or covered with cable protector. All walkways must be kept clear of items to ensure safe passage for all lab users	2	B	L
Lone working	Injury or harm due to work-related activities	All staff Equipment	Avoid lone working wherever possible. Ensure hazardous work is not carried out while alone. All work carried out should be limited to low-risk activities, such as computer work or operating standard lab equipment	3	B	M
Display Screen Equipment	Musculoskeletal damage from improper posture. Eye strain. Additional stress.	All staff	Staff to read and acknowledge the Display Screen Equipment Visual Display Guidance. If issues are encountered, DSE checklist is to be filled out and seek further advice. Training to be issued to staff.	1	D	L

Control of RF Components within a Passive RF Tracking System

Electrical Safety Laptop PSU (Networked and standalone) Server or static computer equipment Equipment taken on trial / test equipment	Faulty electrical equipment exposing all staff to dangerous voltages (240V) Shocks and burns Discomfort from shocks or jolts as equipment fails	All staff Test Equipment Equipment under test	Staff are to ensure all kit being used is Portable Appliance Tested (PAT) and in date, in accordance with the following schedules: - Laptop Power Supplies – every 2 years - Power Cables – every 2 years “Static” Equipment – every 4 years - Trials Equipment/Test Equipment – every 2 years Training is to be provided by the group management to inform users of: - PAT Requirements and testing frequency - The need to inspect all equipment pre and post use for any signs of damage or wear to any wires - Basic awareness of what indicates damage	1	C	L
Lead and Rosin Ingestion and inhalation	Lead on hands may contaminate food or drink Inhalation of vapourised lead or rosin during soldering	Lab users Those Soldering Those handling Lead solder	So far is reasonably practicable non-rosin and non-lead solder may be used. If lead solder is required for activity, all soldering must be carried out as per the following restrictions: - Non-Rosin based solder is to be used - Extraction equipment must be used - Any respiratory sensitivity must be reported to line and workplace management - Lung function health screening must be undertaken each year Should rosin-based soldering be required it may occur in accordance with this risk assessment and control measures. Additional controls are listed in the COSHH assessment. In summary: - Hands must be thoroughly washed after soldering. This is to be done before any consumption of food or drink. - No food or drink on or near the workspaces where soldering is taking place - Safety glasses or spectacles should be worn during soldering - Holdings of solder should be kept to a minimum - Health surveillance (Lung Function) required This excludes the following circumstances:	1	D	M

Control of RF Components within a Passive RF Tracking System

			- Filling or modifying lead solder joints after their production - Lead joints that have been exposed to the weather for significant amount of time			
Liquid Flux Liquid Flux containing similar risks	Liquid flux on hands may contaminate food or drink. Liquid flux may vaporise	Lab users Those soldering	Liquid flux will need: - Extraction systems to remove fumes - Handling with care – no food or drink on or near the soldering workbench, and hands should be washed after use. Liquid flux must be stored in the COSHH cabinet when not in use.	1	D	M
Electromagnetic Fields (EMF) Either: - EMF sources used in RFCM testing - EMF leaking from RF test equipment - RF control and communication equipment (Radios, Wi-Fi, etc)	Sensory effects from EMF Staff at particular risk Medical implants deactivated or heated by EMF Certain levels of EMF can cause issues for foetal development	All staff Staff at particular risk (medical implants, body worn medical devices, pregnant)	Emission generated in experiments are to be kept below the health effect Exposure Limit Values (ELV) Emitters will be classified (as per Definitions and Source Classification) as follows: - If using a low gain (<5dBi) antenna, and power less than 200mW – Risk Group 2 - If the Effective Isotropic Radiated Power (EIRP) is less than 30dBm – Risk Group 2 - If EIRP is greater than 30dBm a written assessment is required from an EMF expert All staff must undergo the EMF practitioners training course if working directly with or in an area with any EMF sources. Line management must remain aware of any staff at particular risk and ensure that any specialist assessments are produced.	3	A	L

Table 3 – Work-based Risk Assessment

See [Appendix](#) for risk matrix tables

Future Improvement

The final design as of the time of writing this report is a prototype of the control board serving as a testbed for the design process. It is a PCB hosting all the components required to power PICs, transfer data to the devices, and has future capability built in for further testing; this board provides the user the ability to utilise all the electronic and data outputs of the RF devices being controlled. This PCB is powered by a bench power supply because the power management circuit board could not be produced, this circuit would provide the power from two AC-DC 18V SMPS supplying the split voltage supply required by the YIG filters. This was not produced as the components available were not the capable of sensing the split supply voltage, further reasoning includes budget and timeline concerns. Producing multiple iterations of the same type of circuit would have taken too much time and cost more than was necessary; as such this system functionality was removed from the overall prototype development in favour of producing a fully functional circuit at a later date, reducing the overall cost to the project.

Given more time the design could have developed further, producing a smaller circuit capable of fitting into a box that would be mounted onto the PTU. This would be achieved by using surface mount components and removing any unnecessary connections to and from the PICs and RF Devices. Some of the test points could also be removed, as once the code is deemed reliable, consistent, and robust in its data transfers these likely would not be needed as the data transfers would not require testing as frequently as the prototype does.

Along with PCB development the code would be further developed to return device data to the user along the transfer chain, as it is currently unable to do this given the time constraints. Further to this all the existing code would need to be made more robust and efficient in its operation, this would allow for more confidence in the data transfers taking place. An example is the I²C receive interrupt service routine, the PIC can receive bytes from the bus if its address is matched, but a PIC could mistake bus data for its address as the routine is quite simple. A more complex routine would look to identify if a start bit had preceded the byte matching its address as this would be a good AND case to adhere to providing a more robust and reliable routine for the program. The short-term solution was to use available addresses outside of the values of ASCII characters that were being used during data transmissions over the I²C bus.

To turn the prototype into a product I would need to adhere to many internationally accepted standards for PCB design and production, alongside accepted best practices for code development. The international Standards used throughout industry and the programming best practices are explained below:

British Standards Institution – BS EN IEC 61188-7 standardises CAD library construction for PCBs and PCB assemblies.[41] BS EN 61619 manages the testing of insulation fluids for PCBs.[42] BS EN IEC 61188-6-3 standardises the design and use of PCB and PCB assemblies, specifically relating to land pattern design for through-hole components.[43]

Institute for Interconnecting and Packaging Electronic Circuits (IPC) – IPC-A-600 and IPC-A-610 for acceptance of PCBs and assemblies, as well as IPC-2221 to standardise generic board design, and IPC-6012 to provide performance specifications for rigid boards.[44], [45]

International Organization for Standardization provides a framework for quality and environmental management. ISO 9001 provides standards for organizations to establish, implement, maintain, and continuously improve their quality management systems allowing PCB manufacturers optimize internal management, enhance product quality, and strengthen supply chain and risk management

capabilities.^[46] ISO 14001 provides the standard for designing and implementing environmental management systems, covering waste management, resource utilization, and environmental performance monitoring.^[47]

This is also true of the program written to operate this system, it will need to abide by the International Standard ISO/IEC 9899:2024, this standard provides a form and the interpretation of programmes written in the C language. The standard has been adapted from the original ANSI X3.159-1989^[48], allowing for standardisation and therefore best practices to follow, these recommended best practices include naming conventions, function production, and code layout. These best practices include providing meaningful variable and function names which follow naming conventions (Camel, Snake, Kebab, and Pascal), avoiding using overly complex and long functions (a single task per function), avoiding deep nesting of cases as this will become hard to read and follow, and segmentation of blocks into code paragraphs in addition to good use of indentation for control structures to clearly layout the code.^[49]

Conclusion

Having run out of time a formal experiment could not be carried out; this would have aided in understanding the reliability and robustness of the developed prototype. Despite being unable to perform the experiment a test methodology was considered, this would consist of sending many types of messages through the system (acting as user inputs) with the aim to measure the Bit Error Rate for each stage of the data chain and for the system overall. The Bit Error Rate values could then be graphed to generate an accuracy figure for the system as a whole, and for each transmission throughout the system to identify any weaker communication points.

To conclude, further development of this prototype into a product require more team members to be included, one to manage the project and budget, ensuring timelines are met and budgets not exceeded, and another further develop the code into a more reliable and robust system capable of taking user inputs and reliably sending the correct signals to and from the appropriate devices. Allowing for the hardware designs of the power and control PCBs as well as the enclosure housing the devices to be fully developed. This would allow each element of the product to be designed and fabricated in line with all the International Standards required for production.

If the prototype was to be further developed into a production unit, the circuit would be changed to a single PIC programmed with all the necessary functionalities of the prototype circuit. Reduce production costs as fewer expensive microcontrollers would be required (also removing the supporting components) this in turn reduces the overall size of the PCB by reducing the number of components. Thus, further reduces the PCB fabrication costs due to the decreased PCB dimensions. The complexity of the code would be lessened by removing the requirement for inter-PIC communication as the single PIC would receive the user commands and send the instructions to the devices directly. The simplification of the programme needed means it should take less time to produce minimising time and workforce costs during development.

The overall cost increase of having more skilled people working on the project would be negated as the time taken to complete the product would be reduced and it would be completed to a higher standard than the prototype that was produced. Ensuring the team is diversely skilled allows for better work to be carried out more effectively and efficiently.

Reflection

Throughout the duration of this project, I gained microcontroller programming capability and further developed my PCB design and project management skills. I used courses produced by microchip university to develop my abilities in high-speed data communication and implement the lessons learnt into the PCB design, I also utilised the Practical I2C: Introduction, Implementation and Troubleshooting and Debugging Techniques for Serial Communications (I2C/SPI/UART) to assist the development of the data protocol interrupts and debug the functionality. Having completed this project my future learning is to start learning how to write Python programmes for easy rapid development of embedded systems and to complete more training on PCB design, enabling me to design more complex circuits for future work.

During this project, I found the code development to be more difficult and require more time than many other aspects, it also meant that some mistakes were made in an effort to reduce the amount of time spent carrying out other tasks. An example is the time taken to produce and develop the I²C interrupts used to transmit and receive the data, the attention this required meant that time was taken away from the design of the control PCB leading to some elements being forgotten. E.g. The contrast voltage potentiometer on the LCD.

Another issue I encountered was using MPLAB and the Real ICE device, these would both periodically not work for unexplained reasons. The Real ICE device would be unable to communicate with and programme or run code on the PIC, I would then have to power it off and on again multiple times for it to start working again. The age of the ICE meant that I could only use MPLAB X v6.2 or below so when MPLAB updated I was unable to use the device and when I was forced to update my laptop to windows 11 the device stopped working entirely. The issues I came up against with MPLAB IDE were to do with projects not compiling due to make file errors which were ill defined, and meant I had to remake many of the projects multiple times.

The parts of this project that went smoothly were the PCB design (despite some errors), component procurement, and adaption to the necessary changes throughout the project. The PCB was designed quickly and had all the required capability, the fabrication and component procurement were carried out quickly to not impede on the speed of prototype production. The required change in the PIC microcontroller used was handled well, as my decision was made quickly and procurement of the 46K22 chip followed immediately. During code production I drew out flowcharts of how the code was going to function before I started to write the functions, then using the PIC datasheets and sourced examples I was able to utilise interrupts effectively for multiple types of data protocol. I also tested the data sorting functions using an online C++ compiler before implementing them into the main project, this allowed me to use printed statements to effectively debug anything I had written so I had a better understanding if the functions were working as intended and correctly; this ensured that I had confidence in the functions I was implementing into the PIC projects.

References

- [1] admin, 'PIC Microcontroller : Architecture and Its Applications', WatElectronics.com. Accessed: Nov. 01, 2024. [Online]. Available: <https://www.watelectronics.com/pic-microcontroller-architecture-and-applications/>
- [2] thepiyushguptamarketing, 'Pic Microcontroller Guide: Features, Applications & Uses'. Accessed: Nov. 01, 2024. [Online]. Available: <https://fpgainsights.com/fpga/pic-microcontroller/>
- [3] A. Ltd, 'What is RISC?', Arm | The Architecture for the Digital World. Accessed: Nov. 01, 2024. [Online]. Available: <https://www.arm.com/glossary/risc>
- [4] T. Agarwal, 'Introduction to PIC Microcontrollers and its Architecture', ElProCus - Electronic Projects for Engineering Students. Accessed: Nov. 01, 2024. [Online]. Available: <https://www.elprocus.com/introduction-to-pic-microcontrollers-and-its-architecture/>
- [5] 'Von Neumann & Harvard Architectures', DEV Community. Accessed: Dec. 07, 2024. [Online]. Available: <https://dev.to/user-name/von-neumann-harvard-architectures-30f5>
- [6] M. Lab, 'PIC MICROCONTROLLER ARCHITECTURE', Microcontrollers Lab. Accessed: Nov. 01, 2024. [Online]. Available: <https://microcontrollerslab.com/pic-microcontroller-architecture/>
- [7] 'PIC microcontroller tutorial'. Accessed: Nov. 01, 2024. [Online]. Available: https://www.microcontrollerboard.com/pic_microcontroller.html
- [8] B. Admin, 'Which PIC Microcontroller to Choose?', Kanda Electronics Blog. Accessed: Nov. 08, 2024. [Online]. Available: <https://www.kanda.com/blog/microcontrollers/pic-microcontrollers/pic-microcontroller-choose/>
- [9] 'Advantages of PIC 18 Series over PIC 16 series', NBCAFE. Accessed: Nov. 08, 2024. [Online]. Available: <https://www.nbcafe.in/advantages-of-pic-18-series-over-pic-16-series/>
- [10] '39632e.pdf'. Accessed: Dec. 07, 2024. [Online]. Available: <https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/39632e.pdf>
- [11] 'Adafruit Learning System'. Accessed: Dec. 07, 2024. [Online]. Available: <https://learn.adafruit.com/assets/59130>
- [12] 'Difference Between Serial and Parallel Transmission', GeeksforGeeks. Accessed: Oct. 31, 2024. [Online]. Available: <https://www.geeksforgeeks.org/difference-between-serial-and-parallel-transmission/>
- [13] J. Roller, '5 Data Transmission Modes in Computer Networks You Should Know', IEEE Computer Society. Accessed: Oct. 31, 2024. [Online]. Available: <https://www.computer.org/publications/tech-news/trends/5-data-transmission-modes/>
- [14] E. Technology, 'Difference Between Synchronous & Asynchronous Transmission', ELECTRICAL TECHNOLOGY. Accessed: Dec. 07, 2024. [Online]. Available: <https://www.electricaltechnology.org/2020/05/difference-between-synchronous-asynchronous-transmission.html>
- [15] R. Pelayo, 'What is I2C? | Protocol Guide', Microcontroller Tutorials. Accessed: Dec. 07, 2024. [Online]. Available: <https://www.teachmemicro.com/i2c-primer/>

- [16] 'Kumar - Embedded Communications Protocols and Internet of .pdf'. Accessed: Oct. 31, 2024. [Online]. Available: <http://www.gpcet.ac.in/wp-content/uploads/2018/08/UNIT-V.pdf>
- [17] E. Brown, 'What Is Full Duplex Communication?', CallWave. Accessed: Dec. 07, 2024. [Online]. Available: <https://callwave.com/full-duplex-communication/>
- [18] Sebastian, 'Microcontroller Connection Protocols: W1, I2C, SPI, UART', Geek Culture. Accessed: Oct. 31, 2024. [Online]. Available: <https://medium.com/geekculture/microcontroller-connection-protocols-w1-i2c-spi-uart-7625ad013e60>
- [19] 'Overview of 1-Wire Technology and Its Use | Analog Devices'. Accessed: Oct. 31, 2024. [Online]. Available: <https://www.analog.com/en/resources/technical-articles/guide-to-1wire-communication.html>
- [20] R. Solutions, 'Microcontrollers are the brains of these modern digital devices. the primary building blocks of IoT devices and the heart of most modern tech. how do microcontrollers communicate with other devices?', Regami Solutions. Accessed: Nov. 01, 2024. [Online]. Available: <https://www.regami.solutions/post/how-do-devices-communicate-with-microcontrollers-differences-between-spi-i2c-uart-can-and-usb>
- [21] 'Communication network protocols', Farnell. Accessed: Nov. 01, 2024. [Online]. Available: <https://uk.farnell.com/communication-network-protocols-trc-ar>
- [22] yida, 'UART vs I2C vs SPI – Communication Protocols and Uses', Latest Open Tech From Seeed. Accessed: Nov. 01, 2024. [Online]. Available: <https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/>
- [23] T. Agarwal, 'USB Protocol : Architecture, Working Specifications & Applications', ElProCus - Electronic Projects for Engineering Students. Accessed: Nov. 08, 2024. [Online]. Available: <https://www.elprocus.com/usb-protocol/>
- [24] 'How USB Works - Developer Help'. Accessed: Nov. 08, 2024. [Online]. Available: <https://developerhelp.microchip.com/xwiki/bin/view/applications/usb/how-it-works/>
- [25] 'USB Operation: Protocol, Data Transfer & Packets » Electronics Notes'. Accessed: Nov. 10, 2024. [Online]. Available: <https://www.electronics-notes.com/articles/connectivity/usb-universal-serial-bus/protocol-data-transfer.php>
- [26] 'How USB Works: Communication Protocol (Part 2)', CircuitBread. Accessed: Nov. 10, 2024. [Online]. Available: <https://www.circuitbread.com/tutorials/how-usb-works-communication-protocol-part-2>
- [27] 'The Importance of Power and Ground Planes in PCB Design'. Accessed: Nov. 09, 2024. [Online]. Available: <https://www.advancedpcb.com/en-us/blog/the-importance-of-power-and-ground-planes-in-pcb-design/>
- [28] 'The Ultimate PCB Routing Guidelines'. Accessed: Nov. 09, 2024. [Online]. Available: <https://www.nextpcb.com/blog/pcb-routing>
- [29] 'Routing Traces in PCBs: Best Practices'. Accessed: Nov. 09, 2024. [Online]. Available: <https://resources.pcb.cadence.com/blog/2024-routing-traces-in-pcb-best-practices>

- [30] 'UART vs. SPI vs. I2C: Routing & Layout Guidelines', Altium. Accessed: Nov. 08, 2024. [Online]. Available: <https://resources.altium.com/p/i2c-vs-spi-vs-uart-how-layout-these-common-buses>
- [31] 'I2C PCB Layout: Best Practices for Optimal Performance – Hillman Curtis: Printed Circuit Board Manufacturing & SMT Assembly Manufacturer'. Accessed: Nov. 08, 2024. [Online]. Available: <https://hillmancurtis.com/i2c-pcb-layout/>
- [32] 'Tips for Optimal High Speed SPI Layout Routing'. Accessed: Nov. 08, 2024. [Online]. Available: <https://resources.pcb.cadence.com/blog/2019-tips-for-optimal-high-speed-spi-layout-routing>
- [33] 'Is There an SPI Trace Impedance Requirement?', Altium. Accessed: Nov. 10, 2024. [Online]. Available: <https://resources.altium.com/p/there-spi-trace-impedance-requirement>
- [34] 'PCB Layout Design - ESP32-S2 - — ESP Hardware Design Guidelines latest documentation'. Accessed: Nov. 09, 2024. [Online]. Available: <https://docs.espressif.com/projects/esp-hardware-design-guidelines/en/latest/esp32s2/pcb-layout-design.html#uart>
- [35] ian, 'Understanding ICSP for PIC Microcontrollers', Instructables. Accessed: Dec. 07, 2024. [Online]. Available: <https://www.instructables.com/Understanding-ICSP-for-PIC-Microcontrollers/>
- [36] 'In-Circuit Serial Programming (ICSP) Guide', 2003.
- [37] 'ME Labs, Inc. | 1-719-520-5323 | In Circuit Serial Programming with ME Labs programmers'. Accessed: Nov. 06, 2024. [Online]. Available: <https://melabs.com/support/icsp.htm>
- [38] 'SM-Drivers-MLFP-MLFR.pdf'. Accessed: Apr. 25, 2025. [Online]. Available: <https://www.microlambdawireless.com/uploads/pdfs/SM-Drivers-MLFP-MLFR.pdf>
- [39] 'Valon 5019 Frequency Synthesizer', Valon Technology, LLC. Accessed: Jan. 14, 2025. [Online]. Available: <https://www.valonrf.com/5019-frequency-synthesizer-20ghz-542911.html>
- [40] 'PIC18F16Q40 Curiosity Nano Hardware User Guide', 2020.
- [41] 'BS EN 61188-7:2017 | 30 Sep 2017 | BSI Knowledge'. Accessed: Apr. 25, 2025. [Online]. Available: <https://knowledge.bsigroup.com/products/printed-boards-and-printed-board-assemblies-design-and-use-electronic-component-zero-orientation-for-cad-library-construction>
- [42] 'BS EN 61619:1997 | 15 Jul 1997 | BSI Knowledge'. Accessed: Apr. 25, 2025. [Online]. Available: <https://knowledge.bsigroup.com/products/insulation-liquids-contamination-by-polychlorinated-biphenyls-pcbs-method-of-determination-by-capillary-column-gas-chromatography>
- [43] 'BS EN IEC 61188-6-3:2025 | 28 Feb 2025 | BSI Knowledge'. Accessed: Apr. 25, 2025. [Online]. Available: <https://knowledge.bsigroup.com/products/circuit-boards-and-circuit-board-assemblies-design-and-use-land-pattern-design-description-of-land-pattern-for-through-hole-components-tht>
- [44] MCL, 'A Full Guide of the IPC Standards for Printed Circuit Boards', mcl. Accessed: Apr. 25, 2025. [Online]. Available: <https://www.mclpcb.com/blog/ipc-standards-for-pcbs/>
- [45] 'IPC Classes and Complying with IPC Standards for PCB Design', Altium. Accessed: Apr. 25, 2025. [Online]. Available: <https://resources.altium.com/p/complying-with-ipc-standards-for-pcb-design>

[46] 'ISO 9001:2015', ISO. Accessed: Apr. 25, 2025. [Online]. Available: <https://www.iso.org/standard/62085.html>

[47] 'ISO 14001:2015', ISO. Accessed: Apr. 25, 2025. [Online]. Available: <https://www.iso.org/standard/60857.html>

[48] B. Kelechava, 'The Current C Programming Language Standard - ISO/IEC 9899:2024 (C24) - ANSI Blog', The ANSI Blog. Accessed: Apr. 25, 2025. [Online]. Available: <https://blog.ansi.org/c-programming-language-standard-iso-iec-9899-2024/>

[49] 'Coding Standards and Best Practices to Follow | BrowserStack'. Accessed: Apr. 18, 2025. [Online]. Available: <https://www.browserstack.com/guide/coding-standards-best-practices>

Bibliography

'C4PICs - The PIC microcontroller architecture'. Accessed: Apr. 28, 2025. [Online]. Available: https://www.flowcode.co.uk/courses/C4PICs/Course%20Files/PICmicro_reference/The_PIC_microcontroller/The_PIC_microcontroller_architecture.html

'What is a PIC microcontroller? What can it do?' Accessed: Apr. 28, 2025. [Online]. Available: <https://technologystudent.com/pics/picgen1.html>

'PIC16F VS PIC18F: Analysis of PIC Mid-Range 8 bit family and Advanced 8-bits Family', <https://www.cytron.io/tutorial>. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.cytron.io/tutorial/pic16f-vs-pic18fanalysis-pic-mid-range-8-bit-family-advanced-8-bits-family>

Voltshop, 'Communication Protocols', Voltshop. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.voltshop.net/communication-protocols/>

'Difference Between Serial and Parallel Transmission', GeeksforGeeks. Accessed: Oct. 31, 2024. [Online]. Available: <https://www.geeksforgeeks.org/difference-between-serial-and-parallel-transmission/>

'Serial vs Parallel Communication', Newhaven Display. Accessed: Apr. 28, 2025. [Online]. Available: <https://newhavendisplay.com/blog/serial-vs-parallel-communication/>

'Serial communication Basic Knowledge -RS-232C/RS-422A/485-', CONTEC. Accessed: Apr. 28, 2025. [Online]. Available: <https://app-contec-sitecore-prod-cd-gfgjgcbgf9e4cadf.japaneast-01.azurewebsites.net/support/basic-knowledge/daq-control/serial-communicatin>

'Serial | Arduino Documentation'. Accessed: Apr. 28, 2025. [Online]. Available: <https://docs.arduino.cc/language-reference/en/functions/communication/serial/>

'1-Wire Devices | Analog Devices'. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.analog.com/en/product-category/1wire-devices.html#category-detail>

'1-Wire Protocol | Arduino Documentation'. Accessed: Apr. 28, 2025. [Online]. Available: <https://docs.arduino.cc/learn/communication/one-wire/>

'UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices'. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html>

'Universal Asynchronous Receiver-Transmitter (UART) | Arduino Documentation'. Accessed: Apr. 28, 2025. [Online]. Available: <https://docs.arduino.cc/learn/communication/uart/>

'What Is UART: Meaning, Working, Models, and Uses | Spiceworks - Spiceworks', Spiceworks Inc. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.spiceworks.com/tech/networking/articles/what-is-uart/>

'class UART – duplex serial communication bus — MicroPython latest documentation'. Accessed: Apr. 28, 2025. [Online]. Available: <https://docs.micropython.org/en/latest/library/machine.UART.html>

J. Wu, 'A Basic Guide to I2C', 2022.

‘Inter-Integrated Circuit (I2C) Protocol | Arduino Documentation’. Accessed: Apr. 28, 2025. [Online]. Available: <https://docs.arduino.cc/learn/communication/wire/>

S. Santos, ‘ESP32 I2C Communication: Set Pins, Multiple Bus Interfaces and Peripherals | Random Nerd Tutorials’. Accessed: Apr. 28, 2025. [Online]. Available: <https://randomnerdtutorials.com/esp32-i2c-communication-arduino-ide/>

‘<https://www.robot-electronics.co.uk/i2c-tutorial>’. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.robot-electronics.co.uk/i2c-tutorial>

S. Campbell, ‘Basics of the I2C Communication Protocol’, Circuit Basics. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/>

‘Understanding I2C Communication and How to Debug the I2C Protocol?’, Total Phase Blog. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.totalphase.com/blog/2020/07/understanding-i2c-communication-and-how-to-debug-the-i2c-protocol/>

‘I2C - What’s That?’, I2C Bus. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.i2c-bus.org/http://www.i2c-bus.org/i2c-bus/>

‘What is SPI?: The Basics of Serial Peripheral Interface - Keysight Oscilloscope Glossary - Keysight Technologies’. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.keysight.com/used/gb/en/knowledge/glossary/oscilloscopes/what-is-spi-the-basics-of-serial-peripheral-interface>

‘Introduction to SPI Interface | Analog Devices’. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.analog.com/en/resources/analog-dialogue/articles/introduction-to-spi-interface.html>

‘class SPI – a Serial Peripheral Interface bus protocol (controller side) — MicroPython latest documentation’. Accessed: Apr. 28, 2025. [Online]. Available: <https://docs.micropython.org/en/latest/library/machine.SPI.html>

‘Controller Area Network (CAN) Protocol Overview’. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/controller-area-network--can--overview.html>

‘CAN Bus Explained - A Simple Intro [2025]’, CSS Electronics. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial>

‘<https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1745786673736>’. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1745786673736>

AutoPi.io, ‘What is CAN bus: Everything You Need to Know the Protocol’, AutoPi.io. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.autopi.io/blog/can-bus-explained/>

‘What Is USB (Universal Serial Bus)? Types, and Importance - Spiceworks’, Spiceworks Inc. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.spiceworks.com/tech/tech-general/articles/universal-serial-bus/>

T. Agarwal, 'USB Protocol : Architecture, Working Specifications & Applications', ElProCus - Electronic Projects for Engineering Students. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.elprocus.com/usb-protocol/>

R. LeCount, 'USB Data Packets Explained', USB Makers Intl. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.usbmakers.com/usb-data-packets-explained>

'What is a Printed Circuit Board? Make Circuits by Connecting Components', Altium. Accessed: Apr. 28, 2025. [Online]. Available: <https://resources.altium.com/p/what-is-a-pcb>

'PCB Design', ABL Circuits. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.ablcircuits.co.uk/pcb-services/pcb-design/>

S. Campbell, 'A Complete Guide to PCB Design and Layout', Circuit Basics. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.circuitbasics.com/make-custom-pcb/>

'Top 5 PCB Design Layout Guidelines You Need to Know', Altium. Accessed: Apr. 28, 2025. [Online]. Available: <https://resources.altium.com/p/pcb-layout-guidelines>

fx-builder, 'Guide to PCB Power Planes and Best Practices', mcl. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.mclpcb.com/blog/power-plane-pcb-best-practices/>

'Advanced PCB Routing Strategies', AdvancedPCB. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.advancedpcb.com/en-us/blog/advanced-pcb-routing-strategies/>

Proto-Electronics, 'Our Top 10 PCB Routing Tips'. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.proto-electronics.com/blog/top-10-pcb-routing-tips>

'Routing Topology Configuration in PCB Design for Power, Data, and Peripherals', Altium. Accessed: Apr. 28, 2025. [Online]. Available: <https://resources.altium.com/p/routing-topologies-in-your-pcb-and-computer-peripherals>

R. Shashikanth, '11 Best High-Speed PCB Routing Practices', Sierra Circuits. Accessed: Apr. 28, 2025. [Online]. Available: <https://www.protoexpress.com/blog/best-high-speed-pcb-routing-practices/>

Appendix I

Host PIC

main.c

```
/* Main.c file generated by New Project wizard
```

```
*
```

```
* Created: Mon Feb 24 2025
```

```
* Processor: PIC18F46K22
```

```
* Compiler: MPLAB XC8
```

```
*/
```

```
//config bit setup - keyword 'config' must be lower case, parameter in upper case
```

```
// PIC18F16Q40 Configuration Bit Settings
```

```
// 'C' source line config statements
```

```
// CONFIG1H
```

```
#pragma config IESO = ON // Internal/External Oscillator Switchover bit
```

```
#pragma config FCMEN = ON // Fail-Safe Clock Monitor Enable
```

```
#pragma config PRICKEN = ON // Primary Clock is always enabled
```

```
#pragma config FOSC = 8 // Use the internal RC oscillator. Pins RA4 and RA5 are available for I/O. CHANGE TO HS!!!
```

```
// CONFIG2L
```

```
#pragma config BORV = 2 // Brown-out Reset Voltage Selection bits->Brown-out Reset Voltage (VBOR) set to 2.2V
```

```
#pragma config BOREN = SBORDIS // Brown-out Reset enabled (SBOREN bit is ignored)
```

```
#pragma config PWRTEN = OFF // Power-up timer selection bits (PWRT is disabled)
```

```
// CONFIG2H
```

```
#pragma config WDTPS = 32768 // WDT postscale (Divider ratio 1:32768; software control of WDTPS)
```

```
#pragma config WDTEN = OFF // WDT operating mode (WDT enabled regardless of sleep; SWDTEN is ignored)
```

```
// CONFIG3H
```

```
#pragma config MCLR = EXTMCLR // MCLR Enable bit (If LVP = 0, MCLR pin is MCLR; If LVP = 1, RE3 pin function is MCLR )
```

```
// CONFIG4L
```

```
#pragma config DEBUG = OFF // Background Debugger (Background Debugger disabled)
```

```
#pragma config XINST = OFF // Extended Instruction Set Enable bit (Extended Instruction Set and Indexed Addressing Mode disabled)
```

```
#pragma config LVP = OFF // Low Voltage Programming Enable bit (Low voltage programming enabled. MCLR/VPP pin function is MCLR. MCLR configuration bit is ignored)
```

```
#pragma config STVREN = ON // Stack Full/Underflow Reset Enable bit (Stack full/underflow will cause Reset)
```

```
// CONFIG5L
```

```
#pragma config CP3 = OFF // Code Block Protection (not code-protected)
```

```
#pragma config CP2 = OFF
```

```
#pragma config CP1 = OFF
```

```
#pragma config CP0 = OFF
```

```
// CONFIG5H
```

```
#pragma config CPD = OFF // Data EEPROM not code-protected
```

```
#pragma config CPB = OFF // Block Boot not code-protected
```

```
// CONFIG6L
```

```
#pragma config WRT3 = OFF // Write Protection bit (not write-protected)
```

```
#pragma config WRT2 = OFF
```

```
#pragma config WRT1 = OFF
```

```
#pragma config WRT0 = OFF
```

Control of RF Components within a Passive RF Tracking System

```
// CONFIG6H
#pragma config WRTD  = OFF      // Data EEPROM Write Protection bit (not write-protected)
#pragma config WRTB  = OFF      // Boot Block not write-protected
#pragma config WRTC  = OFF      // config register write protection

// CONFIG7L
#pragma config EBTR3 = OFF      // Table Read Protection bit (not protected from table reads)
#pragma config EBTR2 = OFF
#pragma config EBTR1 = OFF
#pragma config EBTR0 = OFF

//CONFIG7H
#pragma config EBTRB = OFF      // Boot Block Table Read Protection bit

// WDTCON
// #pragma config SWDTEN = OFF   // Software Enable or Disable the Watchdog Timer bit

#include <xc.h>
#include "main.h"
#include "interrupts.h"    // include this so the schedule flags can be checked

#define transfAddr 0x04
#define dispAddr 0x06

// Variables
int arrLen = 0;
char crRcvd = 0;
int i2cEnd = 0;
int i2cRdy = 0;

// Arrays
char rcvBuff [15] = {};
char i2cMsg [15] = {};    // make a message to send over the i2c for testing
char dutyCycle [6] = {};
char yigFreq [6] = {};
char synthCmd [11] = {};

// Pointers
char rcvPtr = 0;
char i2cMsgPtr = 0;        // make variable that will contain the address of the start of the i2c message
char dcPtr = 0;
char yigPtr = 0;
char synthPtr = 0;

void __interrupt() all_interrupts(void) {
    if((INTCONbits.TMR0IE == 1) && (INTCONbits.TMR0IF == 1)) {    // if timer0 interrupt is enabled and timer0 interrupt flag
        //if(PIR3bits.TMR0IF == 1) {
            TOISR();                // ...perform timer0 interrupt service routine
        }
        if(PIR2bits.OSCFIF == 1) {                // if the external oscillator has failed, and it will because there isn't one
            PIR2bits.OSCFIF = 0;                // clear the interrupt
        }
        //if((PIE7bits.I2C1TXIE == 1) && (PIR7bits.I2C1TXIF == 1)) {    // if the i2c transmit buffer empty flag is set and the
interrupt is enabled
            if(PIR1bits.SSP1IF == 1) {
                I2CTxISR();                // call the i2c transmit interrupt service routine
            }
            if((PIE1bits.RC1IE == 1) && (PIR1bits.RC1IF == 1)) {
                UARTRxISR();
            }
        }
    }
}

void main(void) {
```

Control of RF Components within a Passive RF Tracking System

```
// Oscillator Configuration
OSCCON = 0xFF; // on SLEEP device Idles / HFINTOSC - (16MHZ) / Device running defined by Fosc of CONFIG1H /
Stable HFINTOSC / System Clk select - INT Osc Block

// Set everything as an input while we get things setup
TRISA = 0xFF; // bit A5 (SPI SS1) set as input (tri-stated)
TRISB = 0xFF;
TRISC = 0xFD; // bit 3 (I2C SCL) and bit 4 (I2C SDA) set as input (tri-stated) / bit 6 (Tx1) and bit 7 (Rx1) set as input
/ C1 set as output
TRISD = 0xEE; // bit 4 (SPI SDO) and bit 0 (SPI SCK) set as output / bit 3 (SPI SS2) and bit 1 (SPI SDI) set as input
(tri-stated)
TRISE = 0x07; // bit 7 (WPUE3) disabled

// set all outputs off/low while we get things setup
LATA = 0x00;
LATB = 0x00;
LATC = 0x00;
LATD = 0x00;
LATE = 0x00;

// turn off all analog inputs, there aren't any in this project and they over ride digital io
ANSELA = 0x00;
ANSELB = 0x00;
ANSELC = 0x00;
ANSELD = 0x00;
ANSELE = 0x00;

//LATAbits.LATA1 = 1; // use this command to address and individual bit, goes for setup registers too.

// Reset Control Register
RCON = 0x00;

// timer0 setup, will be used as a task scheduler
T0CON = 0x03; // timer stopped / 16 bit mode / int instr cycle clk / prescaler assigned - 1:16 pre scaler
T1CON = 0x00; // Fosc/4 as clock source / 1:1 post scaler / disable 2nd osc / sync with Fosc/4 / en R/W in 1 16bit
instr / timer stopped
T3CON = 0x00;
T5CON = 0x00;
T2CON = 0x02; // postscaler 1:1 / timer is off / prescaler is 16
T4CON = 0x02;
T6CON = 0x02;
TMR0H = 0xFF; // Write high byte of timer start value, will actually be written when TMR0L is written
TMR0L = 0x06; // timer period is 4us - 250 counts till overflow gives 1ms task scheduler base tick (INT Fosc =
16MHz)
// Tclk = ((Fosc/4)/prescale)^-1 No. Cnts = 1ms/4us 65535 - cnts + 1 = TMR0

// setup interrupts - not using multi-vector interrupts in the configs above
INTCON = 0xE0; // disable all interrupts for now, disable priority interrupts
// Don't touch INTCON1 - it's read only interrupt status
// Don't touch IVTAD - we will use the default interrupt base
// IVTLOCK = 0x01; // prevent changes to the interrupt vector base address, we don't want this changing due to an
error
// SHADCON is used to modify the interrupt context - don't touch it!

// PIR (interrupt request) registers cleared by default but can be set if peripheral runs in the back ground
// IPR (interrupt priority) register don't do anything due to setting of INTCON0

// setup the i2c in master mode
// set tris registers above
SSP1BUF = 0x00;
SSP1STAT = 0xC0; // disable Slew Rate control, Set I2C to standard speeds (100kHz and 1MHz) and enable SMBus
compliant input logic thresholds
SSP1CON1 = 0x28; // SSP1 Enabled / I2C Master mode, clock = F OSC / (4 * (SSPxADD+1))
```

Control of RF Components within a Passive RF Tracking System

```

SSP1CON2 = 0x00;    // mostly RX statuses, doesn't matter that much for master mode
SSP1CON3 = 0x00;
SSP1ADD = 0x27;     // Fosc - 16MHz Fcy - 4MHz BRG Val - 0x27 Fclock - 100kHz

// setup EUSART port
TXSTA1 = 0x6A;     // 9-bit mode / Tx enabled / Async / Low Speed
RCSTA1 = 0xD0;     // serial port configured / 9-bit mode / Rx Enabled
BAUDCON1 = 0x78;   // Receive and Transmit Data is Inverted (active-low) / 16-bit Baud Rate Generator is used
(SPBRGHx:SPBRGx)
SPBRGH1 = 0x00;    // Baud Rate High value
SPBRG1 = 0x67;    // Baud Rate Low value

INTCONbits.GIE = 1; // enable all interrupts
PIE1bits.RC1IE = 1; // enable
INTCONbits.TMR0IE = 1; // enable timer 0 overflow interrupt, all other interrupts are disabled by default
T0CONbits.TMR0ON = 1; // start timer zero
TRISBbits.TRISB0 = 0; // reset as output
TRISBbits.TRISB1 = 0; // reset as output

while(1) {          // welcome to the main loop, you'll never leave!
    if(schedFlags.heartBeat == 1) { // if the heard beat schedule flag has been set...
        LATBbits.LATB0 = ! LATBbits.LATB0; // switch the on board led, should get about 2 hertz
        schedFlags.heartBeat = 0; // reset the schedule flag to preserve the system timing
        NOP(); // put in no-op to use as a break point
    }
    if(crRcvd == 1) {
        crRcvd = 0;
        arrLen = array_length();
        data_identifier(arrLen);
        load_i2c_buff();
        i2cRdy = 1;
    }
    if(i2cRdy == 1) {
        i2cRdy = 0;
        i2c_transmit(transfAddr);
    }
    if(i2cRdy == 2) {
        i2cRdy = 0;
        i2cMsgPtr = 0;
        i2c_transmit(disfAddr);
    }
    if(i2cEnd == 1 && i2cMsg[0] == transfAddr) {
        i2cEnd = 0;
        i2cRdy = 2;
    }
    else if(i2cEnd == 1 && i2cMsg[0] == disfAddr) {
        i2cEnd = 0;
        i2cRdy = 3;
        clear_i2c_buff();
    }
    else { i2cRdy = 0; }
}

//CLRWDI(); // clears the watchdog timer so we don't get reset
return;
}

void i2c_transmit(char address) {
    PIE1bits.SSP1IE = 1; // enable interrupt on SSP1 buffer empty
    SSP1CON2bits.SEN = 1; // set start bit in the register
    i2cMsg[0] = address; // load the client address into element 0 of the array
}

```

```

int array_length(void) {
    int length = 0;
    for(int i = 0; i < 12; i++) {        // for loop to count no. of chars
        if(rcvBuff[i] == '\0') {        // if element is NULL
            length = i;                // make arrLen equal to i
            break;                    // break from for loop
        }
        else {                        // else continue for loop op
            continue;
        }
    }
    return length;                    // return array length var
}

void data_identifier(int token) {
    token = arrLen - 1;                // use array length minus Null and minus \r
    if(rcvBuff[token] == 0x25) {        // use token int to identify '%'
        dutyCycle[0] = 0x6F;
        for(int j = 0; j <= token; j++) {    // '%' id'd fill duty cycle array
            dutyCycle[j+1] = rcvBuff[j];
            rcvBuff[j] = 0x00;
        }
    }
    else {                            // if '%' not id'd must be a different command
        char c = rcvBuff[0];            // make first element var c
        if(0x40 < c && c < 0x5B) {        // if c is a Capitalised letter
            synthCmd[0] = 0x73;        // insert identity token
            for(int k = 0; k <= arrLen; k++) { // ... then fill the array offset by 1 element
                synthCmd[k+1] = rcvBuff[k];
                rcvBuff[k] = 0x00;
            }
        }
        else if (0x2F < c && c < 0x3A) {    // else if c is a number
            yigFreq[0] = 0x79;        // insert identifier
            for(int l = 0; l <= arrLen; l++) { // .. then fill array offset by 1 element
                yigFreq[l+1] = rcvBuff[l];
                rcvBuff[l] = 0x00;
            }
        }
        else {                        // if not a Capital Letter or number not relevant data
            //cout << "Bad Data" << endl;
            return;
        }
    }
}

void load_i2c_buff(void) {
    if(dutyCycle[0] == 0x6F) {
        for(dcPtr = 0; dcPtr <= arrLen; dcPtr++) {
            i2cMsg[dcPtr + 1] = dutyCycle[dcPtr];
        }
        i2cRdy = 1;
    }
    else if(synthCmd[0] == 0x73) {
        for(synthPtr = 0; synthPtr <= arrLen; synthPtr++) {
            i2cMsg[synthPtr + 1] = synthCmd[synthPtr];
        }
        i2cRdy = 1;
    }
    else if(yigFreq[0] == 0x79) {
        for(yigPtr = 0; yigPtr <= arrLen; yigPtr++) {
            i2cMsg[yigPtr + 1] = yigFreq[yigPtr];
        }
    }
}

```

```

        i2cRdy = 1;
    }
    else {
        //cout << "ERROR" << endl;
        return;
    }
}

void clear_i2c_buff(void) {
    for(int m = 0; m <= arrLen; m++) {
        i2cMsg[m] = 0x00;
    }
}

```

main.h

```

#ifndef main_included
#define main_included

// list of function prototypes
void TOISR(void);
void I2CTxISR(void);
char UARTRxISR(void);
void i2c_transmit(char);
int array_length(void);
void data_idenfier(int);
void load_i2c_buff(void);
void clear_i2c_buff(void);

//list of exertnal variables
extern char i2cMsg [];           // give other functions access to the i2cMessage array and pointer variable
extern char i2cMsgPtr;
extern char rcvBuff [];
extern char rcvPtr;
extern int arrLen;
extern char dutyCycle [];
extern char yigFreq [];
extern char synthCmd [];
extern char crRcvd;
extern int i2cEnd;

#endif

```

interrupts.c

```

#include <xc.h>
#include "interrupts.h"
#include "main.h"

// timer 0 isr variables
union unionScheduleFlags schedFlags;    // creates and instantiation of the schedule flags that that can be used in
these function
char count = 0;                        // interrupt counter
char lastCount = 0;                    //
char flagGen = 0;

unsigned int hbTaskCtr = 0xFE0B;
unsigned int i2cTaskCtr = 0xCF2B;
void TOISR (void) {                    // interrupt gets called every 100us
    TOCONbits.TMR0ON = 0;              // stop the timer
    INTCONbits.TMR0IF = 0;             // clear timer0 interrupt flag
    TMR0H = 0xFF;                      // rest the timer registers, high first due to 16 bit write
    TMR0L = 0x05;
}

```

Control of RF Components within a Passive RF Tracking System

```
LATCbits.LATC2 =! LATCbits.LATC2;    // Test Pin

// SCHEDULE THE HEARBEAT TASK (likely to become master flash in the future)
hbTaskCtr++;                        // increment the heart beat task counter
if(hbTaskCtr == 0) {                // check if the counter 0
    schedFlags.heartBeat = 1;        // perform blink routine
    hbTaskCtr = (unsigned int)0xFE0B; // restore counter value
}

i2cTaskCtr++;
if(i2cTaskCtr == 0) {
    schedFlags.i2cTx = 1;
    i2cTaskCtr = (unsigned int)0xCF2B;
}

TOCONbits.TMR0ON = 1;               // set the timer going again
}

char rcvPtr = 0;
char crRcvd = 0;
char UARTRxISR(void) {
    if(RCSTAbits.FERR == 1) { // if there is a frame error
        rcvBuff[rcvPtr] = RCREG1;
        rcvBuff[rcvPtr] = 0x00;
    }
    else {
        if (RCREG1 == '\r') { // would be an else but XC8 didn't allow a breakpoint if it was
            rcvPtr = 0;        // ...and the pointer
            crRcvd = 1;
        }
        else {
            rcvBuff[rcvPtr] = RCREG1; // put the next character in the transmit buffer
            rcvPtr++;
        }
    }
    return crRcvd;
}

char TxStageCntr = 0;               // setup a switch expression
int i2cEnd = 0;
void I2CTxISR(void) {
    PIR1bits.SSP1IF = 0;            // clear the interrupt flag
    switch (TxStageCntr) {
        case 0:
            SSP1BUF = i2cMsg[i2cMsgPtr]; // put the next character in the transmit buffer
            i2cMsgPtr++;                 // increment the pointer
            TxStageCntr++;
            break;
        case 1:
            if(SSP1CON2bits.ACKSTAT == 0) { // if the ACK is received from client
                if(i2cMsg[i2cMsgPtr] != '\0') { // check for last bit of data
                    SSP1BUF = i2cMsg[i2cMsgPtr]; // put the next character in the transmit buffer
                    i2cMsgPtr++;                 // increment array element pointer
                }
                else {
                    SSP1CON2bits.PEN = 1; // send stop bit
                    TxStageCntr++;        // increment switch expression
                    i2cMsgPtr = 0;        // reset array element pointer
                }
            }
            else {
                SSP1CON2bits.PEN = 1; // send stop bit
                TxStageCntr++;        // increment switch expression
            }
        }
    }
}
```

Control of RF Components within a Passive RF Tracking System

```
        i2cMsgPtr = 0;           // reset array element pointer
    }
    break;
case 2:
    TxStageCntr = 0;             // reset switch expression
    PIE1bits.SSP1IE = 0;         // disable the interrupt
    i2cEnd = 1;
    break;
default:
    SSP1CON2bits.PEN = 1;        // send stop
    TxStageCntr = 2;             // in fail carry out stage 2
}
}

                                interrupts.h
#ifndef interrupts_included
#define interrupts_included

// define union for the schedule flags
union unionScheduleFlags{
    unsigned char theFlags;
    struct{
        unsigned driveDisp    :1; // 32m2
        unsigned unused_2     :3; // 2nd block of unused bits
        unsigned updateDisp   :1; // 64ms, div down to update disp @6Hz
        unsigned i2cTx        :1; // 128ms
        unsigned heartBeat    :1; // 256ms (msb)
    };
};
extern union unionScheduleFlags schedFlags;

#endif

interrupts.h
#ifndef interrupts_included
#define interrupts_included

// define union for the schedule flags
union unionScheduleFlags{
    unsigned char theFlags;
    struct{
        unsigned driveDisp    :1; // 32m2
        unsigned unused_2     :3; // 2nd block of unused bits
        unsigned updateDisp   :1; // 64ms, div down to update disp @6Hz
        unsigned i2cTx        :1; // 128ms
        unsigned heartBeat    :1; // 256ms (msb)
    };
};
extern union unionScheduleFlags schedFlags;

#endif
```

Appendix II

Transfer PIC

main.c

```
/* Main.c file generated by New Project wizard
```

```
*
```

```
* Created: Mon Feb 24 2025
```

```
* Processor: PIC18F46K22
```

```
* Compiler: MPLAB XC8
```

```
*/
```

```
//config bit setup - keyword 'config' must be lower case, parameter in upper case
```

```
// PIC18F16Q40 Configuration Bit Settings
```

```
// 'C' source line config statements
```

```
// CONFIG1H
```

```
#pragma config IESO = ON // Internal/External Oscillator Switchover bit
```

```
#pragma config FCMEN = ON // Fail-Safe Clock Monitor Enable
```

```
#pragma config PRICKEN = ON // Primary Clock is always enabled
```

```
#pragma config FOSC = 8 // Use the internal RC oscillator. Pins RA4 and RA5 are available for I/O. CHANGE TO HS!!!
```

```
// CONFIG2L
```

```
#pragma config BORV = 2 // Brown-out Reset Voltage Selection bits->Brown-out Reset Voltage (VBOR) set to 2.2V
```

```
#pragma config BOREN = SBORDIS // Brown-out Reset enabled (SBOREN bit is ignored)
```

```
#pragma config PWRTEN = OFF // Power-up timer selection bits (PWRT is disabled)
```

```
// CONFIG2H
```

```
#pragma config WDTPS = 32768 // WDT postscale (Divider ratio 1:32768; software control of WDTPS)
```

```
#pragma config WDTEN = OFF // WDT operating mode (WDT enabled regardless of sleep; SWDTEN is ignored)
```

```
// CONFIG3H
```

```
#pragma config MCLR = EXTMCLR // MCLR Enable bit (If LVP = 0, MCLR pin is MCLR; If LVP = 1, RE3 pin function is MCLR )
```

```
// CONFIG4L
```

```
#pragma config DEBUG = OFF // Background Debugger (Background Debugger disabled)
```

```
#pragma config XINST = OFF // Extended Instruction Set Enable bit (Extended Instruction Set and Indexed Addressing Mode disabled)
```

```
#pragma config LVP = OFF // Low Voltage Programming Enable bit (Low voltage programming enabled. MCLR/VPP pin function is MCLR. MCLR configuration bit is ignored)
```

```
#pragma config STVREN = ON // Stack Full/Underflow Reset Enable bit (Stack full/underflow will cause Reset)
```

```
// CONFIG5L
```

```
#pragma config CP3 = OFF // Code Block Protection (not code-protected)
```

```
#pragma config CP2 = OFF
```

```
#pragma config CP1 = OFF
```

```
#pragma config CP0 = OFF
```

```
// CONFIG5H
```

```
#pragma config CPD = OFF // Data EEPROM not code-protected
```

```
#pragma config CPB = OFF // Block Boot not code-protected
```

```
// CONFIG6L
```

```
#pragma config WRT3 = OFF // Write Protection bit (not write-protected)
```

```
#pragma config WRT2 = OFF
```

```
#pragma config WRT1 = OFF
```

```
#pragma config WRT0 = OFF
```

```
// CONFIG6H
```

Control of RF Components within a Passive RF Tracking System

```
#pragma config WRTD    = OFF        // Data EEPROM Write Protection bit (not write-protected)
#pragma config WRTB    = OFF        // Boot Block not write-protected
#pragma config WRTC    = OFF        // config register write protection

// CONFIG7L
#pragma config EBTR3   = OFF        // Table Read Protection bit (not protected from table reads)
#pragma config EBTR2   = OFF
#pragma config EBTR1   = OFF
#pragma config EBTR0   = OFF

//CONFIG7H
#pragma config EBTRB   = OFF        // Boot Block Table Read Protection bit

// WDTCON
//#pragma config SWDTEN = OFF      // Software Enable or Disable the Watchdog Timer bit

#include <xc.h>
#include "main.h"
#include "interrupts.h"    // include this so the schedule flags can be checked
#include <stdio.h>
#include <math.h>

// variables
double freq = 0;
char i2cTransfStat = 0;
char processI2C = 0;
char spiEnd = 0;
int arrLen = 0;

// arrays
char yigFreq [6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
char yigInstr [3] = {0x00, 0x00, 0x01};
char synthCmd [11] = {};
char readString[11] = {0x00, 0x80, 0xFF, 0x80, 0x00};
char i2cMSG[15] = {};

// Pointers
char synthPtr = 0;
char yigPtr = 0;
char i2cMsgPtr = 0;    // make variable that will contain the address of the start of the i2c message
char readStringPtr = 0;

void __interrupt() all_interrupts(void) {
    if((INTCONbits.TMR0IE == 1) && (INTCONbits.TMR0IF == 1)) {    // if timer0 interrupt is enabled and timer0 interrupt flag
        //if(PIR3bits.TMR0IF == 1) {
            TOISR();    // ...perform timer0 interrupt service routine
        }
        if(PIR2bits.OSCFIF == 1) {    // if the external oscillator has failed, and it will because there isn't one
            PIR2bits.OSCFIF = 0;    // clear the interrupt
        }
        //if((PIE7bits.I2C1TXIE == 1) && (PIR7bits.I2C1TXIF == 1)) {    // if the i2c transmit buffer empty flag is set and the
interrupt is enabled
            if(PIR1bits.SSP1IF == 1) {
                I2CRxISR();    // call the i2c receive interrupt service routine
            }
            if((PIE1bits.TX1IE == 1) && (PIR1bits.TX1IF == 1)) {
                UARTTxISR();
            }
            if((PIE3bits.SSP2IE == 1) && (PIR3bits.SSP2IF == 1)) {
                SPITxISR();
            }
        }
    }
}
```

Control of RF Components within a Passive RF Tracking System

```
void main(void) {
    // Oscillator Configuration
    OSCCON = 0xFF;    // on SLEEP device Idles / HFINTOSC - (16MHZ) / Device running defined by Fosc of CONFIG1H /
    Stable HFINTOSC / System Clk select - INT Osc Block

    // Set everything as an input while we get things setup
    TRISA = 0xFF;    // bit A5 (SPI SS1) set as input (tri-stated)
    TRISB = 0xFF;
    TRISC = 0xFD;    // bit 3 (I2C SCL) and bit 4 (I2C SDA) set as input (tri-stated) / bit 6 (Tx1) and bit 7 (Rx1) set as input
    / C1 set as output
    TRISD = 0xEE;    // bit 4 (SPI SDO) and bit 0 (SPI SCK) set as output / bit 3 (SPI SS2) and bit 1 (SPI SDI) set as input
    (tri-stated)
    TRISE = 0x07;    // bit 7 (WPUE3) disabled

    // set all outputs off/low while we get things setup
    LATA = 0x00;
    LATB = 0x00;
    LATC = 0x00;
    LATD = 0x00;
    LATE = 0x00;

    // turn off all analog inputs, there aren't any in this project and they over ride digital io
    ANSELA = 0x00;
    ANSELB = 0x00;
    ANSELC = 0x00;
    ANSELD = 0x00;
    ANSELE = 0x00;

    //LATAbits.LATA1 = 1;    // use this command to address and individual bit, goes for setup registers too.

    // Reset Control Register
    RCON = 0x00;

    // timer0 setup, will be used as a task scheduler
    T0CON = 0x03;    // timer stopped / 16 bit mode / int instr cycle clk / prescaler assigned - 1:16 pre scaler
    T1CON = 0x00;    // Fosc/4 as clock source / 1:1 post scaler / disable 2nd osc / sync with Fosc/4 / en R/W in 1 16bit
    instr / timer stopped
    T3CON = 0x00;
    T5CON = 0x00;
    T2CON = 0x02;    // postscaler 1:1 / timer is off / prescaler is 16
    T4CON = 0x02;
    T6CON = 0x02;
    TMR0H = 0xFF;    // Write high byte of timer start value, will actually be written when TMR0L is written
    TMR0L = 0x06;    // timer period is 4us - 250 counts till overflow gives 1ms task scheduler base tick (INT Fosc =
    16MHz)
    // Tclk = ((Fosc/4)/prescale)^-1    No. Cnts = 1ms/4us    65535 - cnts + 1 = TMR0

    // setup interrupts - not using multi-vector interrupts in the configs above
    INTCON = 0xE0;    // disable all interrupts for now, disable priority interrupts
    // Don't touch INTCON1 - it's read only interrupt status
    // Don't touch IVTAD - we will use the default interrupt base
    // IVTLOCK = 0x01;    // prevent changes to the interrupt vector base address, we don't want this changing due to an
    error
    // SHADCON is used to modify the interrupt context - don't touch it!
    INTCONbits.TMR0IE = 1;    // enable timer 0 overflow interrupt, all other interrupts are disabled by default
    // PIR (interrupt request) registers cleared by default but can be set if peripheral runs in the back ground
    // IPR (interrupt priority) register don't do anything due to setting of INTCON0

    // setup the i2c in master mode
    // set tris registers above
    SSP1BUF = 0x00;    // clear the SSP1 Buffer just incase it has data in it
    SSP1STAT = 0xC0;    // disable Slew Rate control, Set I2C to standard speeds (100kHz and 1MHz) and enable SMBus
    compliant input logic thresholds
```

Control of RF Components within a Passive RF Tracking System

```

SSP1CON1 = 0x26;    // SSP1 Enabled / I2C Client 7-bit Address
SSP1CON2 = 0x01;    // enable clock stretching on the client to give it thinking time
SSP1CON3 = 0x00;    //
SSP1ADD = 0x04;     // PIC Address

// setup SPI master mode
SSP2BUF = 0x00;
SSP2STAT = 0x80;     // Input data sampled at end of data output time / Transmit on Idle to active clock state /
Rest is I2C only / CKE = 0
SSP2CON1 = 0x20;     // SPI port enabled / CKP = 0 / SPI Host mode, CLK = Fosc/4
SSP2CON2 = 0x00;
SSP2CON3 = 0x00;
SSP2ADD = 0x27;     // Fosc - 16MHz Fcy - 4MHz BRG Val - 0x27 Fclock - 100kHz

// setup EUSART port
TXSTA1 = 0x60;     // 9-bit mode / Tx enabled / Async / Low Speed
RCSTA1 = 0xD0;     // serial port configured / 8-bit mode
BAUDCON1 = 0x08;    // 16-bit Baud Rate Generator is used (SPBRGHx:SPBRGx)
SPBRGH1 = 0x00;     // Baud Rate High value
SPBRG1 = 0x67;     // Baud Rate Low value

INTCONbits.GIE = 1; // enable all interrupts
PIE1bits.SSP1IE = 1;
PIE3bits.SSP2IE = 1;
TOCONbits.TMR0ON = 1; // start timer zero
TRISAbits.TRISA2 = 0; // reset as output
TRISBbits.TRISB0 = 0;

yig1Sel = 1;

while(1) {
    // welcome to the main loop, you'll never leave!
    if(schedFlags.heartBeat == 1) { // if the heard beat schedule flag has been set...
        LATBbits.LATB0 = ! LATBbits.LATB0; // switch the on board led, should get about 2 hertz
        schedFlags.heartBeat = 0; // reset the schedule flag to preserve the system timing
        NOP(); // put in no-op to use as a break point
        arrLen = array_length();
        data_sorting(i2cMSG[1]);
    }
    if(yigFreq[0] > 0x00) {
        freq = convert_2_float();
        yig_freq_instr(freq); // convert frequency to 16-bit Hex Instruction
    }
    if(yigInstr[0] > 0x00 || yigInstr[1] > 0x00) {
        yig1Sel = 0; // yig 1 chip select set low
        spi_transmit(yig1Sel); // transmit YIG filter instructions
    }
    if(spiEnd == 1) {
        yig1Sel = 1;
        spiEnd = 0;
    }
    if(synthCmd[0] > 0x00) {
        PIE1bits.TX1IE = 1; // enable the Tx Interrupt
        TXREG1 = synthCmd[synthPtr]; // Send first character
        synthCmd[synthPtr] = 0x00;
        synthPtr++;
    }
}
//CLRWDI(); // clears the watchdog timer so we don't get reset
return;
}

int array_length(void) {
    int length = 0;

```

```

for(int i = 0; i < 15; i++) {      // for loop to count no. of chars
    if(i2cMSG[i] == '\0') {      // if element is NULL
        length = i;             // make arrLen equal to i
        break;                  // break from for loop
    }
    else {                       // else continue for loop op
        continue;
    }
}
return length;                  // return array length var
}

void data_sorting(char id) {
    switch(id) {
        case 's':
            for(synthPtr = 0; synthPtr <= arrLen; synthPtr++) {
                synthCmd[synthPtr] = i2cMSG[synthPtr+2];
                i2cMSG[synthPtr] = 0x00;
            }
            i2cMSG[0] = 0x00;
            i2cMSG[1] = 0x00;
            synthPtr = 0;
            break;
        case 'y':
            for(yigPtr = 0; yigPtr <= arrLen; yigPtr++) {
                yigFreq[yigPtr] = i2cMSG[yigPtr+2];
                i2cMSG[yigPtr] = 0x00;
            }
            i2cMSG[0] = 0x00;
            i2cMSG[1] = 0x00;
            yigPtr = 0;
            break;
        default:
            return;
            break;
    }
}

double convert_2_float() {
    double f = strtod(yigFreq, NULL);      // convert the char array to a float
    if(f <= 6 || f >= 18) {                // sense check the calculated value
        f = 6;                             // if 6 > f > 18 set f as 6 (boot freq)
    }
    else {
        f = f;                             // if okay allow f value
    }
    for(int i = 0; i < 7; i++) {
        yigFreq[i] = 0x00;
    }
    return f;
}

void yig_freq_instr(double freq1) {        // optional second freq
    double yig1_16bit = 65535 * ((freq1 - 6) / (18 - 6)); // GHz freq to 16bit value equation / 6 and 18GHz are the max
    and min freq
    //double YIG2_16bit = 65535 * ((freq2 - 6) / (18 - 6)); // optional second freq equation, duplicate process with
    YIG2 identifier for multiple YIGs @ diff Freqs
    short yig1_16bit_round = round(yig1_16bit);
    char yig1_upper = (yig1_16bit_round >> 8); // separate rounded value (8 MSBs)
    char yig1_lower = (yig1_16bit_round >> 0); // separate rounded value (8 LSBs)
    yigInstr[0] = yig1_upper; // Load 8 MSBs into first position of array
    yigInstr[1] = yig1_lower; // Then load 8 LSBs into next position
}

```

Control of RF Components within a Passive RF Tracking System

```
void spi_transmit(int cs) {
    PIE3bits.SSP2IE = 1;           // enable the interrupt
    yigPtr = 0;                     // ensure pointer is reset
    yigInstr[2] = '\0';             // append NULL char to yig instruction array
    if(!cs) {                       // if Chip Select line is held low
        SSP2BUF = yigInstr[yigPtr]; // load first byte into shift register
        yigInstr[yigPtr] = 0x00;    // clear the character from the array
        yigPtr++;                   // increment the pointer
    }
}

main.h
#ifndef main_included
#define main_included

#define transfAddr 0x04
#define yig1Sel     LATAbits.LATA2
#define yig2Sel     LATDbits.LATD3

// list of function prototypes
void TOISR(void);
void I2CRxISR(void);
void UARTTxISR(void);
void SPITxISR(void);
int array_length(void);
void data_sorting(char);
double convert_2_float(void);
void yig_freq_instr(double);
void spi_transmit(int);

//list of external variables
extern char i2cMsg [];              // give other functions access to the i2cMessage array and pointer variable
extern char i2cMsgPtr;
extern char synthCmd [];
extern char synthPtr;
extern char readString[];
extern char readStringPtr;
extern char i2cTransfStat;
extern char i2cMsgBytes;
extern char i2cAddr;
extern char i2cMSG[];
extern char processI2C;
extern char yigFreq [];
extern char yigInstr [];
extern char yigPtr;
extern char spiStageCntr;
extern char spiEnd;
extern int arrLen;

#endif

interrupts.c
#include <xc.h>
#include "interrupts.h"
#include "main.h"

// timer 0 isr variables
union unionScheduleFlags schedFlags; // creates and instantiation of the schedule flags that that can be used in
these function
char count = 0;
char lastCount = 0;
char flagGen = 0;
```

Control of RF Components within a Passive RF Tracking System

```

unsigned int hbTaskCtr = 0xFE0B;

void TOISR (void) {           // interrupt gets called every 100us
    TOCONbits.TMR0ON = 0;     // stop the timer
    INTCONbits.TMR0IF = 0;    // clear timer zero interrupt flag
    TMR0H = 0xFF;             // rest the timer registers, high first due to 16 bit write
    TMR0L = 0x05;

    LATCbits.LATC2 = ! LATCbits.LATC2;    // Test Pin

    // SCHEDULE THE HEARBEAT TASK (likely to become master flash in the future)
    hbTaskCtr++;               // increment the heart beat task counter
    if(hbTaskCtr == 0) {       // check if the counter 0
        schedFlags.heartBeat = 1;    // perform blink routine
        hbTaskCtr = (unsigned int)0xFE0B;    // restore counter value
    }
    TOCONbits.TMR0ON = 1;      // set the timer going again
}

char i2cProgress = 0x00;      // count how far the i2c has got through its message
char inByteCtr = 0;

void I2CRxISR(void) {         // the i2c did something, let's deal with it
    PIR1bits.SSP1IF = 0;      // clear the interrupt flag, the most important part of the interrupt
    if(inByteCtr == 0) {       // see if the start bit was detected, if it was the buffer should contain the address
        i2cMSG[inByteCtr] = SSP1BUF;    // empty SSP2BUF into i2cMSG array
        if (i2cMSG[inByteCtr] == transfAddr) {    // if the value of the element equals the address
            inByteCtr++;           // increment the pointer of the array
        }
        else {
            processI2C = 2;        // if the address is not matched set the I2C process to 2
        }
        SSP1CON1bits.CKP = 1;     // release the clock so the transfer can continue
    }
    else {                     // if inByteCtr > 1
        i2cMSG[inByteCtr] = SSP1BUF;    // write buffer to pointer location
        inByteCtr++;               // increment pointer
        if (inByteCtr == 15) {       // expected amount of data
            processI2C = 1;
            inByteCtr = 0;
        }
        SSP1CON1bits.CKP = 1;      // release the clock so the transfer can continue
    }
}

void UARTTxISR(void) {
    if (synthCmd[synthPtr] == '\0') {    // check to make sure the last character in the message hasn't been sent, if not...
        PIE1bits.TX1IE = 0;            // disable the interrupt
        synthPtr = 0;                   // ...and the pointer
    }
    else {
        TXREG1 = synthCmd[synthPtr];    // put the next character in the transmit buffer
        synthCmd[synthPtr] = 0x00;
        synthPtr++;                     // increment the pointer
    }
}

char spiEnd = 0;
void SPITxISR(void) {
    PIR3bits.SSP2IF = 0;                // reset the interrupt flag
    if (yigPtr == 2) {                  // check for last character in array
        yigPtr = 0;                     // reset the pointer
        spiEnd = 1;                     // flag to rest of code interrupt finished
        PIE3bits.SSP2IE = 0;           // disable the interrupt
    }
}

```

```
}  
else {  
    SSP2BUF = yigInstr[yigPtr];    // put the next character in the transmit buffer  
    yigInstr[yigPtr] = 0x00;       // clear the value from the array  
    yigPtr++;                      // increment the pointer  
}  
}
```

interrupts.h

```
#ifndef interrupts_included  
#define interrupts_included  
  
// define union for the schedule flags  
union unionScheduleFlags{  
    unsigned char theFlags;  
    struct{  
        unsigned driveDisp    :1; // 32m2  
        unsigned unused_2     :3; // 2nd block of unused bits  
        unsigned updateDisp   :1; // 64ms, div down to update disp @6Hz  
        unsigned unused       :1; // 128ms  
        unsigned heartBeat    :1; // 256ms (msb)  
    };  
};  
extern union unionScheduleFlags schedFlags;  
  
#endif
```

Appendix III

Display PIC

main.c

```
/* Main.c file generated by New Project wizard
```

```
*
```

```
* Created: Mon Feb 24 2025
```

```
* Processor: PIC18F46K22
```

```
* Compiler: MPLAB XC8
```

```
*/
```

```
//config bit setup - keyword 'config' must be lower case, parameter in upper case
```

```
// PIC18F16Q40 Configuration Bit Settings
```

```
// 'C' source line config statements
```

```
// CONFIG1H
```

```
#pragma config IESO = ON // Internal/External Oscillator Switchover bit
```

```
#pragma config FCMEN = ON // Fail-Safe Clock Monitor Enable
```

```
#pragma config PRICKEN = ON // Primary Clock is always enabled
```

```
#pragma config FOSC = 8 // Use the internal RC oscillator. Pins RA4 and RA5 are available for I/O. CHANGE TO HS!!!
```

```
// CONFIG2L
```

```
#pragma config BORV = 2 // Brown-out Reset Voltage Selection bits->Brown-out Reset Voltage (VBOR) set to 2.2V
```

```
#pragma config BOREN = SBORDIS // Brown-out Reset enabled (SBORV bit is ignored)
```

```
#pragma config PWRTEN = OFF // Power-up timer selection bits (PWRT is disabled)
```

```
// CONFIG2H
```

```
#pragma config WDTPS = 32768 // WDT postscale (Divider ratio 1:32768; software control of WDTPS)
```

```
#pragma config WDTEN = OFF // WDT operating mode (WDT enabled regardless of sleep; SWDTEN is ignored)
```

```
// CONFIG3H
```

```
#pragma config MCLR = EXTMCLR // MCLR Enable bit (If LVP = 0, MCLR pin is MCLR; If LVP = 1, RE3 pin function is MCLR )
```

```
// CONFIG4L
```

```
#pragma config DEBUG = OFF // Background Debugger (Background Debugger disabled)
```

```
#pragma config XINST = OFF // Extended Instruction Set Enable bit (Extended Instruction Set and Indexed Addressing Mode disabled)
```

```
#pragma config LVP = OFF // Low Voltage Programming Enable bit (Low voltage programming enabled. MCLR/VPP pin function is MCLR. MCLR configuration bit is ignored)
```

```
#pragma config STVREN = ON // Stack Full/Underflow Reset Enable bit (Stack full/underflow will cause Reset)
```

```
// CONFIG5L
```

```
#pragma config CP3 = OFF // Code Block Protection (not code-protected)
```

```
#pragma config CP2 = OFF
```

```
#pragma config CP1 = OFF
```

```
#pragma config CP0 = OFF
```

```
// CONFIG5H
```

```
#pragma config CPD = OFF // Data EEPROM not code-protected
```

```
#pragma config CPB = OFF // Block Boot not code-protected
```

```
// CONFIG6L
```

```
#pragma config WRT3 = OFF // Write Protection bit (not write-protected)
```

```
#pragma config WRT2 = OFF
```

```
#pragma config WRT1 = OFF
```

```
#pragma config WRT0 = OFF
```

Control of RF Components within a Passive RF Tracking System

```
// CONFIG6H
#pragma config WRTD  = OFF      // Data EEPROM Write Protection bit (not write-protected)
#pragma config WRTB  = OFF      // Boot Block not write-protected
#pragma config WRTC  = OFF      // config register write protection

// CONFIG7L
#pragma config EBTR3 = OFF      // Table Read Protection bit (not protected from table reads)
#pragma config EBTR2 = OFF
#pragma config EBTR1 = OFF
#pragma config EBTR0 = OFF

//CONFIG7H
#pragma config EBTRB = OFF      // Boot Block Table Read Protection bit

// WDTCON
#pragma config SWDTEN = OFF     // Software Enable or Disable the Watchdog Timer bit

#include <xc.h>
#include "main.h"
#include "interrupts.h"    // include this so the schedule flags can be checked
#include "LCD.h"
#include <stdio.h>
#include <math.h>

// variables
double freq = 0;
char i2cTransfStat = 0;
char processI2C = 0;
int length = 0;
int arrLen = 0;

// arrays
char yigFreq [6] = {};
char synthCmd [15] = {};
char dutyCycle[5] = {};
char i2cMsg [15] = {};
char string [11];

// Pointers
char synthPtr = 0;
char yigPtr = 0;
char dcPtr = 0;
char i2cMsgPtr = 0;      // make variable that will contain the address of the start of the i2c message

void __interrupt() all_interrupts(void) {
    if((INTCONbits.TMR0IE == 1) && (INTCONbits.TMR0IF == 1)) {    // if timer0 interrupt is enabled and timer0 interrupt flag
        //if(PIR3bits.TMR0IF == 1) {
            TOISR();          // ...perform timer0 interrupt service routine
        }
        if(PIR2bits.OSCFIF == 1) {          // if the external oscillator has failed, and it will because there isn't one
            PIR2bits.OSCFIF = 0;          // clear the interrupt
        }
        //if((PIE7bits.I2C1TXIE == 1) && (PIR7bits.I2C1TXIF == 1)) { // if the i2c transmit buffer empty flag is set and the
        //interrupt is enabled
        if(PIR1bits.SSP1IF == 1) {
            I2CRxISR();          // call the i2c receive interrupt service routine
        }
    }
}

void main(void) {
    // Oscillator Configuration
    OSCCON = 0xFF;      // on SLEEP device Idles / HFINTOSC - (16MHZ) / Device running defined by Fosc of CONFIG1H /
    Stable HFINTOSC / System Clk select - INT Osc Block
}
```

Control of RF Components within a Passive RF Tracking System

```
// Set everything as an input while we get things setup
TRISA = 0xFF;    // bit A5 (SPI SS1) set as input (tri-stated)
TRISB = 0xFF;
TRISC = 0xFD;    // bit 3 (I2C SCL) and bit 4 (I2C SDA) set as input (tri-stated) / bit 6 (Tx1) and bit 7 (Rx1) set as input
/ C1 set as output
TRISD = 0x90;    // bit 1 (RS), 2 (En), 4-7 (D4-D7) set as outputs
TRISE = 0x07;    // bit 7 (WPUE3) disabled

// set all outputs off/low while we get things setup
LATA = 0x00;
LATB = 0x00;
LATC = 0x00;
LATD = 0x00;
LATE = 0x00;

// turn off all analog inputs, there aren't any in this project and they over ride digital io
ANSELA = 0x00;
ANSELB = 0x00;
ANSELC = 0x00;
ANSELD = 0x00;
ANSELE = 0x00;

//LATAbits.LATA1 = 1;    // use this command to address and individual bit, goes for setup registers too.

// Reset Control Register
RCON = 0x00;

// timer0 setup, will be used as a task scheduler
T0CON = 0x03;    // timer stopped / 16 bit mode / int instr cycle clk / prescaler assigned - 1:16 pre scaler
T1CON = 0x00;    // Fosc/4 as clock source / 1:1 post scaler / disable 2nd osc / sync with Fosc/4 / en R/W in 1 16bit
instr / timer stopped
T3CON = 0x00;
T5CON = 0x00;
T2CON = 0x02;    // postscaler 1:1 / timer is off / prescaler is 16
T4CON = 0x02;
T6CON = 0x02;
TMR0H = 0xFF;    // Write high byte of timer start value, will actually be written when TMR0L is written
TMR0L = 0x06;    // timer period is 4us - 250 counts till overflow gives 1ms task scheduler base tick (INT Fosc =
16MHz)
// Tclk = ((Fosc/4)/prescale)^-1    No. Cnts = 1ms/4us    65535 - cnts + 1 = TMR0

// setup interrupts - not using multi-vector interrupts in the configs above
INTCON = 0xE0;    // disable all interrupts for now, disable priority interrupts
// Don't touch INTCON1 - it's read only interrupt status
// Don't touch IVTAD - we will use the default interrupt base
// IVTLOCK = 0x01;    // prevent changes to the interrupt vector base address, we don't want this changing due to an
error
// SHADCON is used to modify the interrupt context - don't touch it!
INTCONbits.TMR0IE = 1;    // enable timer 0 overflow interrupt, all other interrupts are disabled by default
// PIR (interrupt request) registers cleared by default but can be set if peripheral runs in the back ground
// IPR (interrupt priority) register don't do anything due to setting of INTCON0

// setup the i2c in master mode
// set tris registers above
SSP1BUF = 0x00;    // clear the SSP1 Buffer just incase it has data in it
SSP1STAT = 0xC0;    // disable Slew Rate control, Set I2C to standard speeds (100kHz and 1MHz) and enable SMBus
compliant input logic thresholds
SSP1CON1 = 0x26;    // SSP1 Enabled / I2C Client 7-bit Address
SSP1CON2 = 0x01;    // enable clock stretching on the client to give it thinking time
SSP1CON3 = 0x00;    //
SSP1ADD = dispAddr;    // PIC Address
```

Control of RF Components within a Passive RF Tracking System

```

INTCONbits.GIE    = 1;    // enable all interrupts
PIE1bits.SSP1IE   = 1;
T0CONbits.TMR0ON  = 1;    // start timer zero
TRISAbits.TRISA2  = 0;    // reset as output
TRISBbits.TRISB0  = 0;
LATAbits.LATA2    = 1;    // latch pin high

LCD_init();
LCD_configure_cursor_blink(1);
LCD_cursor_set(1,1);

while(1) {
    // welcome to the main loop, you'll never leave!
    if(schedFlags.heartBeat == 1) {
        // if the heard beat schedule flag has been set...
        LATBbits.LATB0 = ! LATBbits.LATB0;    // switch the on board led, should get about 2 hertz
        schedFlags.heartBeat = 0;    // reset the schedule flag to preserve the system timing
        NOP();    // put in no-op to use as a break point
        arrLen = array_length();
        read_message();
    }
    if(string[0] == dispAddr) {
        switch (string[1]) {
            // switch case IDs the 1st element of array "string"
            case 'y':
                LCD_write_string(string);
                break;
            case 'o':
                LCD_write_string(string);
                break;
            case 's':
                LCD_write_string(string);
                break;
            default:
                LCD_write_string("Invalid Input!");
                break;
        }
    }
}
//CLRWDWT();    // clears the watchdog timer so we don't get reset
return;
}

int array_length(void) {
    int length = 0;
    for(int i = 0; i < 15; i++) {
        // for loop to count no. of chars
        if(i2cMsg[i] == '\0') {
            // if element is NULL
            length = i;
            // make arrLen equal to i
            break;
            // break from for loop
        }
        else {
            // else continue for loop op
            continue;
        }
    }
    return length;
    // return array length var
}

void read_message(void) {
    for(int i = 0; i <= arrLen; i++) {
        string[i] = i2cMsg[i];
        i2cMsg[i] = 0x00;
    }
}

main.h
#ifndef main_included

```

Control of RF Components within a Passive RF Tracking System

```
#define main_included

#define dispAddr    0x06

// list of function prototypes
void TOISR(void);
void I2CRxISR(void);
int array_length(void);
void read_message(void);

//list of external variables
extern char i2cMsg[]; // give other functions access to the i2cMessage array and pointer variable
extern char i2cMsgPtr;
extern char synthCmd [];
extern char synthPtr;
extern char readString[];
extern char readStringPtr;
extern char i2cTransfStat;
extern char i2cMsgBytes;
extern char i2cAddr;
extern char processI2C;

#endif

                                interrupts.c

#include <xc.h>
#include "interrupts.h"
#include "main.h"

// timer 0 isr variables
union unionScheduleFlags schedFlags;    // creates and instantiation of the schedule flags that that can be used in
these function
char count = 0;
char lastCount = 0;
char flagGen = 0;

unsigned int hbTaskCtr = 0xFE0B;

void TOISR (void) {              // interrupt gets called every 100us
    TOCONbits.TMR0ON = 0;        // stop the timer
    INTCONbits.TMR0IF = 0;       // clear timer zero interrupt flag
    TMR0H = 0xFF;                // rest the timer registers, high first due to 16 bit write
    TMR0L = 0x05;

    LATCbits.LATC2 = ! LATCbits.LATC2;    // Test Pin

    // SCHEDULE THE HEARBEAT TASK (likely to become master flash in the future)
    hbTaskCtr++;                  // increment the heart beat task counter
    if(hbTaskCtr == 0) {          // check if the counter 0
        schedFlags.heartBeat = 1;    // perform blink routine
        hbTaskCtr = (unsigned int)0xFE0B; // restore counter value
    }
    TOCONbits.TMR0ON = 1;         // set the timer going again
}

char i2cProgress = 0x00;         // count how far the i2c has got through its message
char inByteCtr = 0;
void I2CRxISR(void) {            // the i2c did something, let's deal with it
    PIR1bits.SSP1IF = 0;         // clear the interrupt flag, the most important part of the interrupt
    if(inByteCtr == 0) {          // see if the start bit was detected, if it was the buffer should contain the address
        i2cMsg[inByteCtr] = SSP1BUF; // empty SSP2BUF into i2cMsg array
        if (i2cMsg[inByteCtr] == dispAddr) { // if the value of the element equals the address
            inByteCtr++;          // increment the pointer of the array
        }
    }
}
```

```

    }
    else {
        processI2C = 2;          // if the address is not matched set the I2C process to 2
    }
    SSP1CON1bits.CKP = 1;        // release the clock so the transfer can continue
}
else {                          // if inByteCtr > 1
    i2cMsg[inByteCtr] = SSP1BUF; // write buffer to pointer location
    inByteCtr++;                // increment pointer
    if (inByteCtr == 15) {      // expected amount of data
        processI2C = 1;
        inByteCtr = 0;
    }
    SSP1CON1bits.CKP = 1;        // release the clock so the transfer can continue
}
}
}

```

Interrupts.c

```

#include <xc.h>
#include "interrupts.h"
#include "main.h"

// timer 0 isr variables
union unionScheduleFlags schedFlags; // creates and instantiation of the schedule flags that that can be used in
these function
char count = 0;
char lastCount = 0;
char flagGen = 0;

unsigned int hbTaskCtr = 0xFE0B;

void T0ISR (void) { // interrupt gets called every 100us
    TOCONbits.TMR0ON = 0; // stop the timer
    INTCONbits.TMR0IF = 0; // clear timer zero interrupt flag
    TMR0H = 0xFF; // rest the timer registers, high first due to 16 bit write
    TMR0L = 0x05;

    LATCbits.LATC2 = ! LATCbits.LATC2; // Test Pin

    // SCHEDULE THE HEARBEAT TASK (likely to become master flash in the future)
    hbTaskCtr++; // increment the heart beat task counter
    if(hbTaskCtr == 0) { // check if the counter 0
        schedFlags.heartBeat = 1; // perform blink routine
        hbTaskCtr = (unsigned int)0xFE0B; // restore counter value
    }
    TOCONbits.TMR0ON = 1; // set the timer going again
}

char i2cProgress = 0x00; // count how far the i2c has got through its message
char inByteCtr = 0;
void I2CRxISR(void) { // the i2c did something, let's deal with it
    PIR1bits.SSP1IF = 0; // clear the interrupt flag, the most important part of the interrupt
    if(inByteCtr == 0) { // see if the start bit was detected, if it was the buffer should contain the address
        i2cMsg[inByteCtr] = SSP1BUF; // empty SSP2BUF into i2cMsg array
        if (i2cMsg[inByteCtr] == dispAddr) { // if the value of the element equals the address
            inByteCtr++; // increment the pointer of the array
        }
    }
    else {
        processI2C = 2; // if the address is not matched set the I2C process to 2
    }
    SSP1CON1bits.CKP = 1; // release the clock so the transfer can continue
}
}

```

Control of RF Components within a Passive RF Tracking System

```
else {
    // if inByteCtr > 1
    i2cMsg[inByteCtr] = SSP1BUF; // write buffer to pointer location
    inByteCtr++; // increment pointer
    if (inByteCtr == 15) { // expected amount of data
        processI2C = 1;
        inByteCtr = 0;
    }
    SSP1CON1bits.CKP = 1; // release the clock so the transfer can continue
}
}
```

interrupts.h

```
#ifndef interrupts_included
#define interrupts_included

// define union for the schedule flags
union unionScheduleFlags{
    unsigned char theFlags;
    struct{
        unsigned driveDisp :1; // 32m2
        unsigned unused_2 :3; // 2nd block of unused bits
        unsigned updateDisp :1; // 64ms, div down to update disp @6Hz
        unsigned unused :1; // 128ms
        unsigned heartBeat :1; // 256ms (msb)
    };
};
extern union unionScheduleFlags schedFlags;

#endif
```

Libraries

LCD.c

```
/*
 * MIT License
 *
 * Copyright (c) 2022 Ahmet Burak Irmak (https://youtube.com/Microesque)
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */

/*
 * If you want to navigate this file easier use the shortcuts (netbeans default).
 * "Ctrl + Shift + +" -> Expand all code blocks
 * "Ctrl + Shift + -" -> Collapse all code blocks
 */
```

Control of RF Components within a Passive RF Tracking System

```
*/

#include "LCD.h"
#include <math.h>

#define LCD_BUSY_DELAY      60          //Minimum delay required by short commands[uS]
#define LCD_BUSY_DELAY_EXTENDED (2300-LCD_BUSY_DELAY) //Extra delay required by the longer commands[uS]

#define LCD_CMD_FUNCTION_SET 0x28 //Function Set(4-bit, 2 line, 5x8 dots)
#define LCD_CMD_DISPLAY_CONTROL 0x0C //Display On/Off Control(Display on, cursor off, blinking off)
#define LCD_CMD_ENTRY_MODE 0x06 //Entry Mode Set(Increment, shift off)
#define LCD_CMD_CLEAR 0x01 //Clear Display
#define LCD_CMD_RETURN_HOME 0x02 //Return Home(or shift reset)
#define LCD_CMD_SHIFT_LEFT 0x18 //Shift Left
#define LCD_CMD_SHIFT_RIGHT 0x1C //Shift Right
#define LCD_CMD_CURSOR_INCREMENT 0x14 //Increment Cursor
#define LCD_CMD_CURSOR_DECREMENT 0x10 //Decrement Cursor
#define LCD_CMD_CGRAM_SET 0x40 //Set CGRAM Address

#define LCD_CMD_DDRAM_LINE_1 (0x80-1) //DDRAM address for line-1(-1 from row starting from 1 instead of 0)
#define LCD_CMD_DDRAM_LINE_2 (0xC0-1) //DDRAM address for line-2(-1 from row starting from 1 instead of 0)
#define LCD_CMD_DDRAM_LINE_3 (0x80+19) //DDRAM address for line-3(+19 from 3rd line being the continuation of line-1)
#define LCD_CMD_DDRAM_LINE_4 (0xC0+19) //DDRAM address for line-4(+19 from 4th line being the continuation of line-2)

/* Initialize the global variables */
static uint8_t LCD_display_control = LCD_CMD_DISPLAY_CONTROL;
static uint8_t LCD_entry_mode = LCD_CMD_ENTRY_MODE;

/**
 * Pulses the clock line of LCD\n
 * <STRONG>{!!!Meant for internal use only!!!}</STRONG>
 */
void LCD_pulse_clock(){

    LCD_EN_LAT = 1;          //Set the "EN" pin
    __delay_us(LCD_BUSY_DELAY); //Give the required delay
    LCD_EN_LAT = 0;          //Clear the "EN" pin
}

/**
 * Sends an 8-bit command (in 4-bit mode)\n
 * <STRONG>{!!!Meant for internal use only!!!}</STRONG>
 * @param command: 8-bit command to be sent
 */
void LCD_write_command(char command){

    LCD_RS_LAT = 0; //Enter command mode

    /*----- Load higher order 4-bits -----*/
    LCD_D7_LAT = (command & 0b10000000) >> 7;
    LCD_D6_LAT = (command & 0b01000000) >> 6;
    LCD_D5_LAT = (command & 0b00100000) >> 5;
    LCD_D4_LAT = (command & 0b00010000) >> 4;

    /*----- Send a clock pulse -----*/
    LCD_pulse_clock();

    /*----- Load lower order 4-bits -----*/
    LCD_D7_LAT = (command & 0b00001000) >> 3;
```

Control of RF Components within a Passive RF Tracking System

```
LCD_D6_LAT = (command & 0b00000100) >> 2;
LCD_D5_LAT = (command & 0b00000010) >> 1;
LCD_D4_LAT = (command & 0b00000001) ;

/*----- Send a clock pulse -----*/
LCD_pulse_clock();
}

/**
 * Sends an 8-bit data (in 4-bit mode)\n
 * <STRONG>(!!!Meant for internal use only!!!)</STRONG>s
 * @param data: 8-bit data to be sent
 */
void LCD_write_data(char data){

    LCD_RS_LAT = 1; //Enter data mode

    /*----- Load higher order 4-bits -----*/
    LCD_D7_LAT = (data & 0b10000000) >> 7;
    LCD_D6_LAT = (data & 0b01000000) >> 6;
    LCD_D5_LAT = (data & 0b00100000) >> 5;
    LCD_D4_LAT = (data & 0b00010000) >> 4;

    /*----- Send a clock pulse -----*/
    LCD_pulse_clock();

    /*----- Load lower order 4-bits -----*/
    LCD_D7_LAT = (data & 0b00001000) >> 3;
    LCD_D6_LAT = (data & 0b00000100) >> 2;
    LCD_D5_LAT = (data & 0b00000010) >> 1;
    LCD_D4_LAT = (data & 0b00000001) ;

    /*----- Send a clock pulse -----*/
    LCD_pulse_clock();
}

/**
 * Initializes the LCD\n
 * <u>LCD configured as:</u>\n
 * - Function set: 4-bit, 2-line, 5x8 dots\n
 * - Display: On\n
 * - Cursor display: Off\n
 * - Cursor blink: Off\n
 * - Cursor on write: Increment, no shift\n
 */
void LCD_init(){

    //----- Pin configurations -----
    LCD_RS_TRIS = 0; LCD_RS_LAT = 0;
    LCD_EN_TRIS = 0; LCD_EN_LAT = 0;
    LCD_D4_TRIS = 0; LCD_D4_LAT = 0;
    LCD_D5_TRIS = 0; LCD_D5_LAT = 0;
    LCD_D6_TRIS = 0; LCD_D6_LAT = 0;
    LCD_D7_TRIS = 0; LCD_D7_LAT = 0;

    /*******//
    /*** Sequence for "Initializing 4-bit interface" from datasheet ***/
    /*******//

    //----- Wait for more than 40ms -----
    __delay_ms(60);

    //----- Load 0b0011 in command mode -----
```

Control of RF Components within a Passive RF Tracking System

```
LCD_D7_LAT = 0; LCD_D6_LAT = 0; LCD_D5_LAT = 1; LCD_D4_LAT = 1;

//----- Send a clock pulse -----
LCD_pulse_clock();

//----- Wait for more than 4.1ms -----
__delay_ms(6);

//----- Send a clock pulse -----
LCD_pulse_clock();

//----- Wait for more than 100us -----
__delay_us(150);

//----- Send a clock pulse -----
LCD_pulse_clock();

//----- Load 0b0010 in command mode(4-bit mode start) -----
LCD_D7_LAT = 0; LCD_D6_LAT = 0; LCD_D5_LAT = 1; LCD_D4_LAT = 0;

//----- Send a clock pulse -----
LCD_pulse_clock();

//----- Configure important settings -----
LCD_write_command(LCD_CMD_FUNCTION_SET); //Function Set(4-bit, 2 line, 5x8 dots)
LCD_write_command(LCD_display_control); //Display On/Off Control(Display on, cursor off, blinking off)
LCD_write_command(LCD_entry_mode); //Entry Mode Set(Increment, shift off)
LCD_clear(); //Clear the display
}

/**
 * Sets the cursor location
 * @param row: Row position(1-4)
 * @param column: Column position(1-20)
 */
void LCD_cursor_set(uint8_t row, uint8_t column){

    /* Offset the column address with the corresponding row */
    if(row == 1) column += LCD_CMD_DDRAM_LINE_1;
    if(row == 2) column += LCD_CMD_DDRAM_LINE_2;
    if(row == 3) column += LCD_CMD_DDRAM_LINE_3;
    if(row == 4) column += LCD_CMD_DDRAM_LINE_4;

    /* Set the cursor location with the new address */
    LCD_write_command(column);
}

/**
 * Increments the current cursor position by 1
 */
void LCD_cursor_increment(){

    /* Send the cursor increment command */
    LCD_write_command(LCD_CMD_CURSOR_INCREMENT);
}

/**
 * Decrements the current cursor position by 1
 */
void LCD_cursor_decrement(){

    /* Send the cursor decrement command */
    LCD_write_command(LCD_CMD_CURSOR_DECREMENT);
}
```

```

}

/**
 * Enables/disables character display on screen\n
 * (The LCD will still work and you can still do write operations,
 * only the characters will/won't be displayed)
 * @param state: 1 to enable, 0 to disable
 */
void LCD_configure_lcd_display(uint8_t state){

    /* Change the gobal display control variable accordingly */
    if(state){
        LCD_display_control |= 0b00000100;
    }
    else{
        LCD_display_control &= 0b11111011;
    }

    /* Send the new display control command */
    LCD_write_command(LCD_display_control);
}

/**
 * Turns on/off the line displayed below characters on cursor location
 * @param state: 1 to turn on, 0 to turn off
 */
void LCD_configure_cursor_display(uint8_t state){

    /* Change the gobal display control variable accordingly */
    if(state){
        LCD_display_control |= 0b00000010;
    }
    else{
        LCD_display_control &= 0b11111101;
    }

    /* Send the new display control command */
    LCD_write_command(LCD_display_control);
}

/**
 * Turns on/off the character blinking on cursor location
 * @param state: 1 to turn on, 0 to turn off
 */
void LCD_configure_cursor_blink(uint8_t state){

    /* Change the gobal display control variable accordingly */
    if(state){
        LCD_display_control |= 0b00000001;
    }
    else{
        LCD_display_control &= 0b11111110;
    }

    /* Send the new display control command */
    LCD_write_command(LCD_display_control);
}

/**
 * Clears the whole screen and resets cursor to 1,1
 */
void LCD_clear(){

```

```

/* Send the clear command(automatically sets cursor to 1,1) */
LCD_write_command(LCD_CMD_CLEAR);

/* Give the extended delay needed by the longer command */
__delay_us(LCD_BUSY_DELAY_EXTENDED);

/* Re-send the entry mode, otherwise clear command resets the id to increment */
LCD_write_command(LCD_entry_mode);
}

/**
 * Resets the shifting and resets cursor to 1,1
 */
void LCD_shift_reset(){

    /* Send the return home command(automatically sets cursor to 1,1) */
    LCD_write_command(LCD_CMD_RETURN_HOME);

    /* Give the extended delay needed by the longer command */
    __delay_us(LCD_BUSY_DELAY_EXTENDED);
}

/**
 * Shifts the whole screen left by 1\n
 * (Shifting the screen also shifts the cursor locations!)\n
 * <STRONG>Note: Characters off-screen will be displayed on a different line!</STRONG>
 */
void LCD_shift_left(){

    /* Send the shift left command */
    LCD_write_command(LCD_CMD_SHIFT_LEFT);
}

/**
 * Shifts the whole screen right by 1\n
 * (Shifting the screen also shifts the cursor locations!)\n
 * <STRONG>Note: Characters off-screen will be displayed on a different line!</STRONG>
 */
void LCD_shift_right(){

    /* Send the shift right command */
    LCD_write_command(LCD_CMD_SHIFT_RIGHT);
}

/**
 * Configures if the cursor increments or decrements on a write command
 * @param state: 1 to increment, 0 to decrement
 */
void LCD_configure_entry_id(uint8_t state){

    /* Change the gobal entry mode variable accordingly */
    if(state){
        LCD_entry_mode |= 0b00000010;
    }
    else{
        LCD_entry_mode &= 0b11111101;
    }

    /* Send the new entry mode command */
    LCD_write_command(LCD_entry_mode);
}

/**

```

```

* Enables/disables screen shift on a write command\n
* (Screen will be shifted left when cursor is incrementing, shifted right otherwise)
* @param state: 1 to enable, 0 to disable
*/
void LCD_configure_entry_shift(uint8_t state){

    /* Change the gobal entry mode variable accordingly */
    if(state){
        LCD_entry_mode |= 0b00000001;
    }
    else{
        LCD_entry_mode &= 0b11111110;
    }

    /* Send the new entry mode command */
    LCD_write_command(LCD_entry_mode);
}

/**
* Adds a custom character to the "CGRAM" and resets cursor to 1,1\n
* (Max. 8 custom characters can be added/used at a time, limited
* by the driver chip itself!)\n
* (Character font needs to be 5x8!)\n
* @param *pattern: Binary pattern of the character\n
* (Can be generated by googling "LCD custom character
* generator".)\n
* @param location: CGRAM address to put the character(0-7)
*/
void LCD_add_character(char *pattern, uint8_t address){

    /* Set the CGRAM address with the passed offset(address).
    * Multiply the offset by 8(shift left by 3), since each
    * "CGRAM location" has 8 addresses for 8 rows of character.*/
    LCD_write_command( LCD_CMD_CGRAM_SET + (uint8_t)(address<<3) );

    /* Send the pattern */
    for(uint8_t i=0 ; i < 8 ; i++) {
        LCD_write_data(pattern[i]);
    }

    /* Set cursor to 1,1 to point the cursor back from "CGRAM" to "DDRAM" */
    LCD_cursor_set(1, 1);
}

/**
* Writes a string to the current cursor location\n
* <STRONG>!!!The string has to be null terminated otherwise this function will fail or loop forever!!!</STRONG>
* @param *string: Character array that contains the string(max. 255 characters)
*/
void LCD_write_string(char *string){

    /* Write all characters one by one until you reach "NULL" */
    for(uint8_t i=0 ; string[i] != '\0' ; i++){
        LCD_write_data(string[i]);
    }
}

/**
* Writes an ASCII character to the current cursor location
* @param character: Character to be written
*/
void LCD_write_char(char character){

```

```

/* Write the character */
LCD_write_data(character);
}

/**
 * Writes an integer variable to the current cursor location(max. 32-bit)\n
 * (Negative numbers will be treated as positive!)
 * @param variable: Variable to be written
 * @param back_offset: If the offset is bigger than the
 * total digit count, the difference amount of empty spaces
 * will be put before writing the variable\n
 * (Allows for right to left writing while keeping the first
 * digit stationary)\n
 * (Cursor will still point to the right side of the variable)\n
 * (Leave as 0 if you want the variable to be written as is)
 */
void LCD_write_variable(int32_t variable, uint8_t left_offset){

    int32_t temp_variable; //Temporary variable to work on
    uint8_t digits = 0;    //Variable to hold the number of total digits

    /*----- Turn negative numbers into positive -----*/
    if(variable < 0) variable *= -1;

    /*----- Calculate the total digits -----*/
    temp_variable = variable;    //Copy the variable
    while(1){                    //...
        digits++;                //Increment the digit(do first since even 0 will have 1 digit)
        if(temp_variable < 10) break; //If the number has 1 digit left, break
        temp_variable /= 10;      //Shift the number to the right
    }

    /*----- Give the offset before writing the variable -----*/
    while(1){
        if(left_offset <= digits) break; //If the offset is equal to digits *or less*, break
        LCD_write_data(' ');           //Put an empty space
        left_offset--;                  //Decrement the offset
    }

    /*----- Write the variable on screen -----*/
    while(1){
        temp_variable = variable;        //Copy/re-copy the variable
        for(uint8_t i = 1; i < digits; i++){ //Shift the temporary variable *digits* times
            temp_variable /= 10;          //...
        }                                //...
        LCD_write_data( (temp_variable%10) + '0' ); //Write the digit(with character 0 as offset)
        digits--;                          //Decrement the *digits*(to move onto next)
        if(digits == 0) break;              //If *digits* reaches 0, break
    }
}

/**
 * Writes a float variable to the current cursor location\n
 * (Negative numbers will be treated as positive!)
 * @param variable: Variable to be written
 * @param back_offset: If the offset is bigger than the total
 * digit count(including integer digits, float digits and 1
 * more for the '.' character), the difference amount of
 * empty spaces will be put before writing the variable\n
 * (Allows for right to left writing while keeping the first
 * digit stationary)\n
 * @param f_digit: How many digits of the fractional part to
 * be written\n

```

```

* (Too high value will cause the variable to overflow!)
*/
void LCD_write_float(float variable, uint8_t left_offset, uint8_t f_digit){

    /*----- Calculate the left offset for the integer part.(-1 is for the '.' character) -----*/
    /*----- (Making sure that left offset doesn't wrap back to 255.) -----*/
    if( left_offset >= (f_digit+1) ){
        left_offset = (left_offset-f_digit) - 1;
    }
    else{
        left_offset = 0;
    }

    /*----- Turn negative numbers into positive -----*/
    if(variable < 0) variable *= -1;

    /*----- Seperate the integer and fractional parts using "modff" from "math.h" -----*/
    float integer;          //Define a variable for the integer part
    float fractional;        //Define a variable for the fractional part
    fractional = modff(variable, &integer); //Seperate the integer and fractional parts

    /*----- Write the integer part -----*/
    LCD_write_variable( (int32_t)integer, left_offset ); //Write the integer part with given offset
    LCD_write_char('.'); //Place a dot before the fractional part

    /*----- Write the fractional part -----*/
    for(uint8_t i=0; i < f_digit; i++){ //Limit the fractional digits
        fractional *= 10; //Shift fraction to left
        LCD_write_data( (uint8_t)((int32_t)fractional%10 + '0') ); //Write the digit(with character 0 as offset)
    }
}

```

LCD.h

```

/*
* MIT License
*
* Copyright (c) 2022 Ahmet Burak Irmak (https://youtube.com/Microesque)
*
* Permission is hereby granted, free of charge, to any person obtaining a copy
* of this software and associated documentation files (the "Software"), to deal
* in the Software without restriction, including without limitation the rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included in all
* copies or substantial portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*/

/*
* This library is made for communicating with LCDs (Liquid Crystal Display)
* with PIC microcontrollers.
*

```

Control of RF Components within a Passive RF Tracking System

* The library is written to communicate with LCDs that are controlled by
* the HD44780U chip from hitachi. It's written for the XC8 compiler, and is
* intended to be used with MPLAB X IDE.
*
* All library functions are blocking, so there is no need for any peripherals.
*
* Works with the common 4x20 and 2x16 LCD displays (should work with any LCD
* that uses HD44780U).
*
* For a comprehensive tutorial on how to use the library visit
* "<https://youtube.com/Microesque>".
* Actual video: <https://www.youtube.com/watch?v=pR942IY54oo>
*/

```
#ifndef LCD_H
#define LCD_H

/*****
/***** Update the definitions below accordingly *****/
/*****

#define LCD_RS_TRIS TRISAbits.TRISA0 //TRIS bit of the pin connected to "RS"
#define LCD_EN_TRIS TRISAbits.TRISA1 //TRIS bit of the pin connected to "EN"
#define LCD_D4_TRIS TRISAbits.TRISA2 //TRIS bit of the pin connected to "D4"
#define LCD_D5_TRIS TRISAbits.TRISA3 //TRIS bit of the pin connected to "D5"
#define LCD_D6_TRIS TRISAbits.TRISA4 //TRIS bit of the pin connected to "D6"
#define LCD_D7_TRIS TRISAbits.TRISA5 //TRIS bit of the pin connected to "D7"

#define LCD_RS_LAT LATAbits.LATA0 //LAT bit of the pin connected to "RS"
#define LCD_EN_LAT LATAbits.LATA1 //LAT bit of the pin connected to "EN"
#define LCD_D4_LAT LATAbits.LATA2 //LAT bit of the pin connected to "D4"
#define LCD_D5_LAT LATAbits.LATA3 //LAT bit of the pin connected to "D5"
#define LCD_D6_LAT LATAbits.LATA4 //LAT bit of the pin connected to "D6"
#define LCD_D7_LAT LATAbits.LATA5 //LAT bit of the pin connected to "D7"

#define _XTAL_FREQ 64000000 //Frequency of the sytem clock

/*****
/***** Necessary includes *****/
/*****

#include <xc.h>
#include <stdint.h>

/*****
/***** Function prototypes *****/
/*****

void LCD_init(void);
void LCD_cursor_set(uint8_t, uint8_t);
void LCD_cursor_increment(void);
void LCD_cursor_decrement(void);
void LCD_configure_lcd_display(uint8_t);
void LCD_configure_cursor_display(uint8_t);
void LCD_configure_cursor_blink(uint8_t);
void LCD_clear(void);
void LCD_shift_reset(void);
void LCD_shift_left(void);
void LCD_shift_right(void);
void LCD_configure_entry_id(uint8_t);
void LCD_configure_entry_shift(uint8_t);
void LCD_add_character(char*, uint8_t);
```

```
void LCD_write_string(char*);  
void LCD_write_char(char);  
void LCD_write_variable(int32_t, uint8_t);  
void LCD_write_float(float, uint8_t, uint8_t);  
  
#endif /* LCD_H */
```

Appendix IV

Redundant Functions

```

spi_transmit()
void spi_transmit(int cs) {
    while(!cs) {
        if(yigInstr[yigPtr] != '\0') {
            if(SSP2BUF == 0) {
                SSP2BUF = yigInstr[yigPtr];
                yigPtr++;
            }
            else {
                while(SSP2BUF > 0) {
                    ;
                }
            }
        }
        else {
            return;
        }
    }
    yigPtr = 0;
}

```

// while the Chip Select line is held low
 // if next char is not NULL...
 // check Buffer Full Status Bit
 // if empty fill with current element
 // increment the pointer
 // while buffer is full...
 // hold here
 // reset pointer to 0

The function was setup in the main loop:

```

yig1Sel = 0;
spi_transmit(yig1Sel);
yig1Sel = 1;

```

// yig 1 chip select set low
 // transmit YIG filter instructions
 // yig1 chip select set high

UARTRxISR()

```

char stageCntr = 0;
void UARTRxISR(void) {
    if(RCSTAbits.FERR == 1) {                // if there is a frame error
        rcvBuff[rcvPtr] = RCREG1;
        rcvBuff[rcvPtr] = 0x00;
        return;
    }
    else {
        switch(stageCntr) {
            case 0:
                if(PIR1bits.RCIF == 1) {      // if transmit SR bit is set buffer is empty
                    stageCntr++;              // increment the counter
                }
                break;
            case 1:
                if (rcvBuff[rcvPtr] != '\r') { // check to make sure the last character in the message hasn't been sent
                    RCREG1 = rcvBuff[rcvPtr]; // ... if not, put the next character in the transmit buffer
                    rcvPtr++;                  // increment the pointer
                }
                else {
                    stageCntr++;              // increment the counter
                }
                break;
            case 2:
                PIE1bits.RC1IE = 0;          // disable the interrupt
                stageCntr = 0;                // reset the counter
                rcvPtr = 0;                   // ...and the pointer
                break;
            default:
                stageCntr = 2;                // go to case 2
        }
    }
}

```

Appendix V

Data Sheets

RF Devices

[Frequency Synthesiser](#)

[YIG Filter](#)

[RF Switch](#)

Microcontroller

[Microchip PIC18F16Q40](#)

[Microchip PIC18F16Q40 Curiosity Nano](#)

[Microchip PIC18F46K22](#)

Programmable Oscillator

[Analog Devices LTC6904](#) (Not Used)

Voltage Regulators

[STMicroelectronics L805ACV](#)

[STMicroelectronics L812ACV](#)

[STMicroelectronics L815CV](#)

[STMicroelectronics L915CV](#)

Comparators

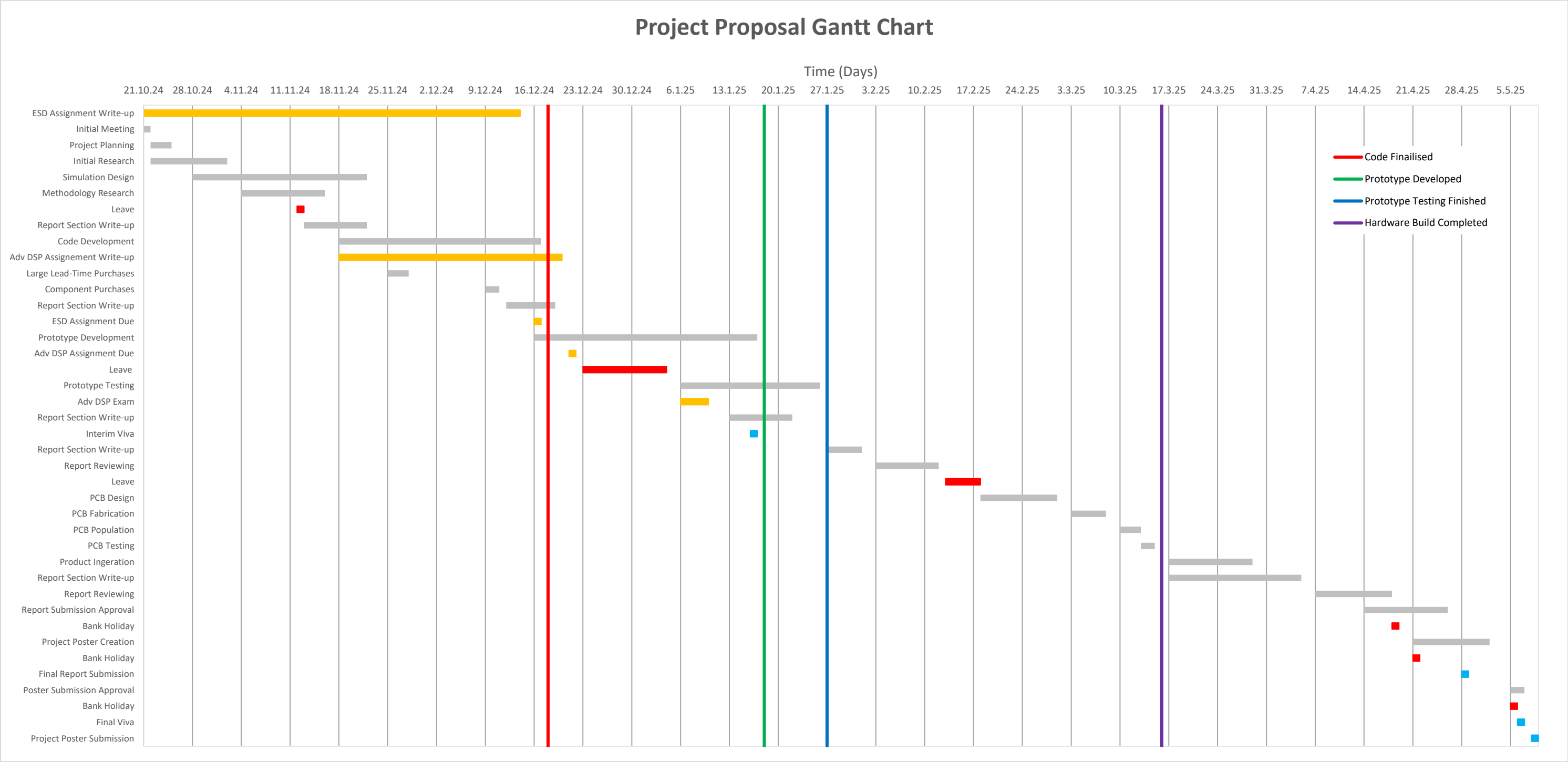
[Texas Instruments LM393P](#)

USB-Serial IC

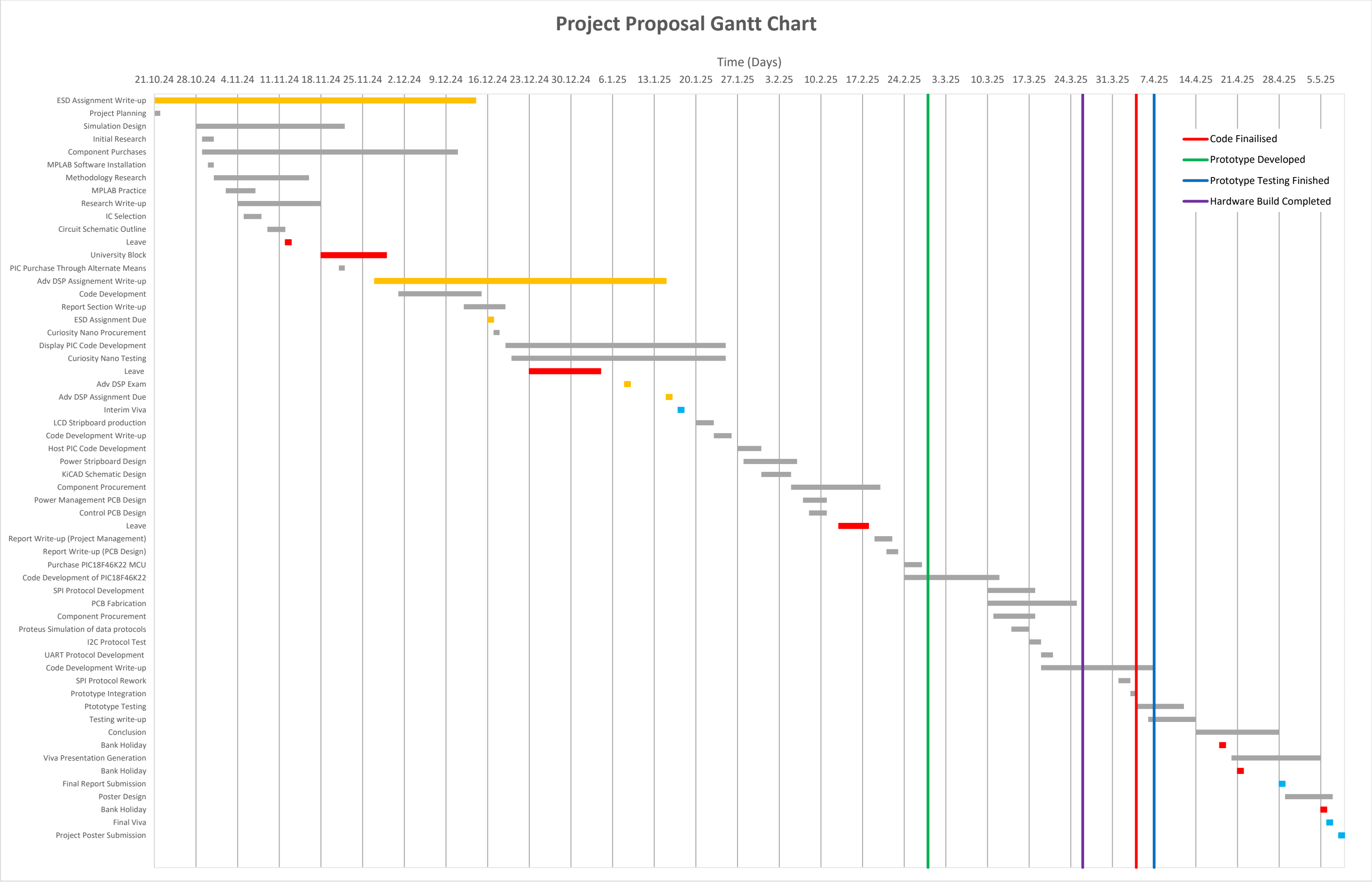
[Microchip MCP2221A](#)

[FTDI USB-RS232-WE-1800-BT-5.0](#) – [FTDI FT232R USB UART IC](#)

Appendix VI
Proposed Gantt Chart



Actual Project Timeline



Appendix VII

Project Risk Priority Number Tables

Probability	Impact
1	May occur but only in exceptional circumstances
2	Not likely to occur in normal circumstances
3	May occur at some time
4	Expected to occur at some time
5	Expected to occur regularly under normal circumstances

Severity	Impact
1	Minor impact to prototype production or functionality.
2	Some impact to prototype production or functionality.
3	Impacts the prototype production or functionality.
4	Significant impact to the prototype production or functionality.
5	Safety hazard created.

Appendix VIII

Risk Matrix Tables

Likelihood	1 Very Low	2 Low	3 Medium	4 High	5 Very High
Probability of Occurrence	May occur but only in exceptional circumstances	Not likely to occur in normal circumstances	May occur at some time	Expected to occur at some time	Expected to occur regularly under normal circumstances

Severity	Health & Safety	Property & Equipment	Environment
A Minor	No likely injuries or Injuries requiring First Aid.	Non-disruptive unplanned property damage or programme disruption.	Small-scale contained spillage.
B Moderate	Minor non-disabling injury requiring professional medical advice.	Minor unplanned property damage or programme disruption.	Minor localised spillage or release requiring action at local level to contain and clear effectively.
C Major	Superficial Injuries or acute illness resulting in temporary discomfort or disability.	Damage or plant and equipment up to £5k causing disruption to programme.	Uncontained spillage or release causing with pollutant confined to the site. No damage to the local environment and no breach of any statutory limit.
D Severe	Lacerations, burns, concussion, sprains, chronic illness or permanent disablement. RIDDOR reportable injuries /diseases	Extensive damage or loss of plant, equipment, and/or facilities, up to £50k causing extensive disruption to programme.	Localised contamination with limited intervention required to remediate. Breach of statutory limits, with potential for enforcement action.
E Critical	Fatality or serious life changing injury with major effect on quality of life.	Complete loss of equipment, plant and/or infrastructure greater than £50k with loss of programme Significant / permanent or long-term damage to the environment.	Significant sustained remediation required. Regulatory enforcement action likely.

Risk Matrix		Probability of Likelihood				
		(1) Very Low	(2) Low	(3) Medium	(4) High	(5) Very High
Severity	(E) Critical	Medium	Medium	High	High	High
	(D) Severe	Low	Medium	Medium	High	High
	(C) Major	Low	Medium	Medium	Medium	High
	(B) Moderate	Very Low	Low	Medium	Medium	Medium
	(A) Minor	Very Low	Very Low	Low	Low	Medium

Control of RF Components within a Passive RF Tracking System

Residual Risk Rating	Action required	Review	Authorisation
Very Low	This represents a low residual risk; where the risk is expected to be effectively managed, and no further controls are needed other than monitoring. However, this is dependent on the control measures being applied and maintained.	Local line management	Local line management
Low	Consider only simple, low-cost additional controls.	Local line management	Local line management
Medium	Consider a range of additional controls to manage the residual risk OR accept that risks have been adequately controlled and are demonstrably 'As Low as Reasonably Practicable' (ALARP)	Local line management HS&E review required Where residual risk for very low likelihood, high severity hazards remain at Medium, Local line management must make department/ Divisional managers aware and pass up for review as required.	Division Manager Delegated authority
High	Action must be taken to review the proposed/existing controls. Work should not commence or should be suspended pending the outcome of the review. Where all possible mitigation has been exhausted and all risks demonstrably ALARP, seek 'Executive Authorisation' to proceed.	HS&E review required	Executive or delegated authority