Technical Manual
Yavin IV Defence System

Team Dirac

November 4, 2014

Chapter 1

Introduction

1.1 Document Identification

This document describes the design and development of the "Yavin IV Orbital Tracking System". This document and design brief is prepared by Dirac Defence Limited for assessment in MTRX3700, year 2014. The was approved by lieutenants Reid and Bell, and small scale testing initiated.

1.2 System Overview

This document outlines a proposed and prototyped design in response to the Rebel Alliance Commander Rye's request for a defence system to combat the imminent threat posed by The Empire, and their Death Star weapons platform. This system is to effectively, efficiently and easily track a space-based planetary annihilator, approximately the size of a small moon.
The system described in this paper is the small scale prototype for stage one of implementation and testing prior to contract approval and large scale deployment. Yavin IV Defence System is designed to provide accurate, low cost tracking for Death Stars and other similar objects.

1.3 Document Overview

This technical document provides a detailed description of the design process and requirements of the various modules of the Orbital Tracking System.

Section one provides the main body of the technical document and outlines the development process of the system.
Chapter one introduces the system and document, details the technical jargon used and addresses reference documents used.
Chapter two describes the system requirements, operational scenarios, module

design and module requirements.
Chapter three details the user interface design and interactions with different user classes.
Chapter four specifies the hardware design, validation and maintenance.
Chapter five details the software design process, architecture and preconditions for use.
Chapter six describes the performance of the system and future development.
Chapter seven outlines the safety implications of the system.
Chapter eight addresses conclusions and provides final final analysis of the system.

Section two contains the supporting documents, calculations, DOxygen documentation and code listings.

1.4 Reference Documents

The present document is prepared on the basis of the following reference documents, and should be read in conjunction with it.

"MTRX3700 Mechatronics 3, Major Project: Multi Sensor Death Star Tracker". David Rye, Sydney, 2014.

1.4.1 Acronyms and Abbreviations

Acronym	Meaning
OTS	Orbital Tracking System, the system under development
ADC	Analogue-to-Digital Converter
Stuff	Meaning of Stuff

Chapter 2

System Description

This section is intended to give a general overview of the basis for the Yavin IV Defence System system design, of its division into hardware and software modules, and of its development and implementation.

2.1 Introduction

The Yavin IV Orbital Tracking System's primary function is to track the Galactic Empire's Death Star system in real-time. It consists of the following two major modules, tracking and user interface.

Tracking This module takes data from the range and pan-tilt modules unfinished

-Range –Ultrasound –Infrared –Temperature

–Pan-Tilt

Menu System –LCD –Local User Interface –Serial

2.2 Operational Scenarios

The system is designed to work in two different modes depending on the class of user, factory mode and end-user mode.
The end user mode is primarily designed to be implemented as the main operational mode, and offers automatic and manual tracking, serial and local user input and output. The factory mode allows technically trained users to calibrate various settings, change sample rates, show statistics and display raw readings, in addition to the full functionality available in end user mode.
Prior to distribution, the system will be calibrated in factory mode and upon distribution, the system will be initialised in user mode. This is to ensure the system is in optimal operating condition and so that the user may not inadvertently modify critical settings.
The factory mode is protected by a physically isolated electrical line and may be initialised by inserting and turning the factory key.

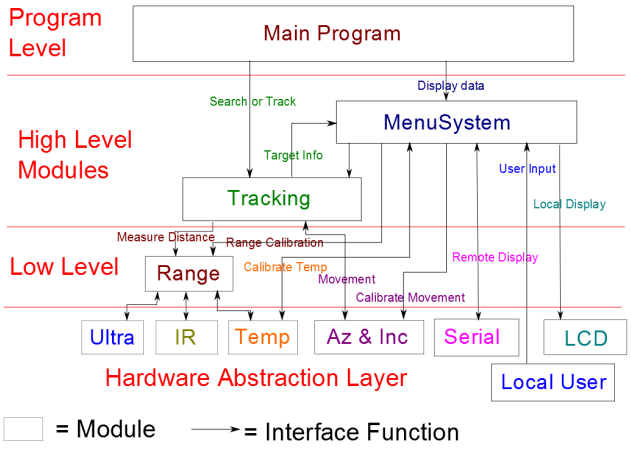

Figure 2.1: Conceptual Diagram of the module breakdown and interaction between modules.

2.3 System Requirements

The operational scenarios considered place certain requirements on the Yavin IV Defence system, and on the modules that comprise it.

The system's primary objective is to locate and track an enemy deathstar, then provide relevant information corresponding to its whereabouts via a serial and local interface. The secondary requirements are the system is to be powered by a 9V PP3 DC battery, implemented on a Microchip PIC18F4520 microcontroller, programmed in "C" language and/or natural assembly language, peripheral electronics shall be constructed on veroboard.

2.4 Module Design

¡Describe the breakdown of the design into functional modules. Each module probably contains both software and hardware. Then include a section like the following 2.5 for each module. Not all of the sub-headings may be relevant for each module.¿

The system was broken down into a number of independent modules which contain their own private variables, functions etc. Fig. 2.1 gives an approximate diagrammatic representation of the way the modules fit together.

2.5 Module Requirements: Serial

2.5.1 Functional Requirements

Inputs

The module must be capable to receiving characters from a user program and transmitting them over serial. The ability to send strings is considered an extension of the basic functionality.
The system must be capable of receiving and storing characters over the serial line, and reporting them back to the user program when requested.

Processes

The module must be capable of performing the following processes:

- Receiving data over serial line
- Sending data over serial line
- Storing data to be transmitted
- Storing data received
- Interface with the user program

Outputs

The module must be capable of returning the following outputs:

- exact characters received over the serial line in the correct order.
- Whether the system has received anything
- Data Characters over the serial line

Timing

The serial module must be capable of:

- Storing characters as soon as they are received over serial
- Retaining received characters until they are handled
- Retaining transmission characters until they can be transmitted

2.5.2 Non-Functional Requirements

Performance

The serial module should have the following performance characteristics:

- Very fast ISR's - to affect background code, and other waiting interrupts as little as possible
- Very low ISR latency - So no characters are missed, and the the module transmits almost as soon as possible.

Interfaces

The following interface requirements are desirable:

- Complete isolation/modularisation (e.g. no global interrupts) - the buffers are not accessible to the rest of the program
- Very simple, intuitive interface functions taking 1 or no arguments that are appropriately named.
- As simple operation as possible - E.g. configureSerial() then transmit().

Design Constraints

The design of the serial module was constrained by the following:

- Only High and Low ISR's on the PIC - Needed a public ISR function that is called when a serial interrupt is fired - Reduced modularity and interrupt response
- Very little memory on the PIC - buffers were restricted to 30 characters

2.6 Conceptual Design: Serial

2.6.1 Description

The serial module takes care of all communication (transmit and receive) over the serial UART (rs-232) port.

2.6.2 Overall Design

The serial module was designed to be as simple and intuitive to use as possible. For this reason the final design ended up being very similar to the serial on an arduino.
There are two inbuilt circular buffers: a transmit buffer and a receive buffer. Any inputs to the serial module for transmission are simply placed into the transmission buffer to be transmitted at the next available opportunity. Any data received over serial is automatically pushed onto the receive buffer to be

used by the program.
The buffers are completely hidden from the rest of the program, and the user simply interacts with the buffers in an intuitive manner to send and receive data

2.6.3 Detailed Description

The serial module is INTERRUPT DRIVEN. This means that any background code can be running while the module is transmitting and/or receiving data over the serial line, and no serial data should ever be missed, overlooked or cut out.
The module contains two circular buffers: a transmit buffer and a receive buffer. These buffers are NOT accessible by the rest of the program. Rather, the module provides a public function transmit() which takes a string, and places it into the transmit buffer. Anything in this buffer is then transmitted character by character when the transmit ready interrupt fires.
Whenever a character is received over serial it is stored in the received buffer by an interrupt. Again this buffer is NOT accessible to the rest of the program. Rather it provides a number of functions to interact with it. The most commonly used of which is the readString() function, which returns everything in the buffer up to a carriage return (e.g. a line of input entered by the user).
This serial module also allows users the opportunity to remove or change data they have already transmitted. If a backspace is received, then instead of storing it in the receive buffer it will remove the last received character from the buffer if that character is not a Carriage Return, Newline, or Escape operator. This enables a much more more user friendly system as otherwise there would be no way to fix any syntax error without pressing enter, getting an error and starting again. Furthermore, without this feature, if a user did backspace and change an input it would result in completely unexpected behaviour.
Fig. 2.2 shows a conceptual diagram of the function of the serial module.

2.6.4 Rational for Design

The rational for the serial module is fairly self explanatory: any decent serial module requires buffers to help prevent data loss, with the ideal implementation being circular buffers. The buffers are kept isolated for modularity and good programming practise, while it also serves to greatly simplify the usage and novice users are not confused by access to buffers and other data structures. The rest of the interface was then chosen to be as simple and intuitive as possible.

2.6.5 Implemented Functionality

Implemented Basic Functionality

The serial module implements the following basic functionality:

- Functioning receive and transmit serial interrupts

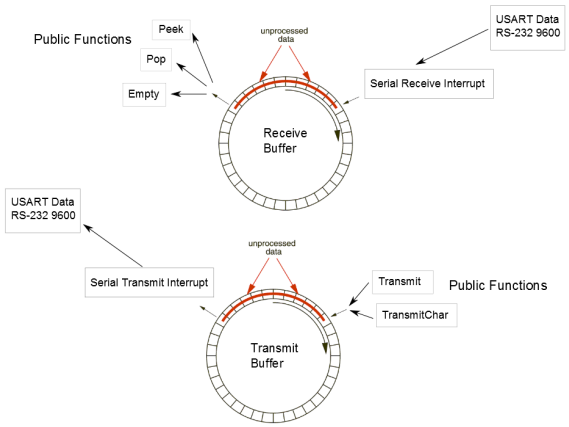

Figure 2.2: Shows Conceptual Diagram of Serial Module -¿ Received input stored by interrupts into circular buffer to await pop commands. Transmit data pushed onto buffer which is transmitted via interrupts

- Configure function to set up the module
- Separate circular buffers to store data received and data to be transmitted
- Public Function to add data to the transmission buffer
- Public Function to read from received buffer, and check if anything has been received

Implemented Additional Functionality

In addition to the required functionality above, the serial module also offers the following functionality:

- Push Null terminated strings to the transmit buffer
- Check if carriage return, or esc has been received
- Pop an entire string from the receive buffer up to a carriage return
- Clear the buffers
- Receiving backspace characters removes the last characters from the buffer (if not CR or ESC)
- Read a string form program memory and transmit
- Peek - Read character without removing from buffer
- Indicate if transmit buffer is empty (all messages sent)

2.6.6 Assumptions Made

The module assumes that the buffers will never be overfilled, i.e. is able to transmit data faster than being written into buffer, or that input is being handled in a timely fashion. Failing this, it is assumed that the oldest data (which is overwritten) is the least meaningful, and that loosing some data will not create catastrophic error in the system. It is recommended to include a wait if sending large blocks of text over serial.

2.6.7 Constraints on Serial Performance

The main constraints on the serial module performance are:

- Baud Rate
- Interrupt Latency

The baud rate sets a maximum rate data can be transferred, which along with the buffer length restricts the rate at which data can be written to the transmit buffer without an overflow occurring. The interrupt latency is the time delay between the interrupt firing and the actual event. This is generally very small (we found 300μs), but a high latency could miss characters being received.

2.6.8 Interface

Refer to the Interface Manual For detailed explanations of the interface functions and how to use them.

List of inputs

The following are inputs that can be sent to the serial module for transmission:

- Strings (RAM char array) through transmit()
- Strings (ROM char array) through sendROM()
- Characters through transmitChar()

In addition, the serial module has serial input from whatever terminal it is communicating to. This is via TTL level logic rs-232 at 9600 baud. The terminal communicates at rs-232 level logic, which is then converted by an adapter on the minimal board.

List of Outputs

The following are outputs that can be requested from the serial module following reception:

- Characters from receivePop(), receivePeek()

- Strings from readString()

The serial module also outputs TTL level logic at 9600 baud rs-232 encoding which is then converted to rs-232 level logic to the terminal.

2.7 Module Requirements: Pan Tilt

2.7.1 Functional Requirements

Inputs

The Pan Tilt module must be capable of taking the following inputs:

- The desired direction

Processes

The Pan Tilt module must be capable of performing the following processes:

- Storing the current direction
  PDM
- Converting a given direction into a PDM
- Continuously generating the control PDM to send to the servo motors

Outputs

The program must be capable of the following outputs:

- PDM control signal to the servo's
- Current direction back to the user program

Timing

The Pan Tilt module must conform to the following timing specifications:

- Module must be interrupt driven
- Must have very low interrupt latency
- PDM's must be offset so that the interrupts for elevation and azimuth don't interfere and block each other
- Delay after movement function to allow the servo's to move to that position

Failure Modes

The Pan Tilt module includes the following assurances against failure:

- New delay information is only set at the end of a cycle to guarantee the PDM frequency
- Validation function to ensure that the high time to within the specified range in the datasheet

2.7.2 Non-Functional Requirements

Performance

The module should have the following performance characteristics to perform well:

- Very low interrupt latency
- Adjustment for interrupt latency

Interfaces

The following interface characteristics are desirable:

- Complete Isolation/modularity so that the rest of the program does not have access to any of the functionality of the module, but merely sets the direction of the Pan-Tilt module
- Very simple, intuitive interface functions taking 1 or no arguments that are appropriately named.
- Very simple module operation, such as configuration and move to desired location

Design Constraints

The design of the Pan Tilt module was primarily constrained by the use of the input capture CCP1 for the ultrasonic sensor to capture the return signal. This meant that the remaining CCP module had to be used to generate both PDM's, which could only be done in software. Had an additional CCP module been available, we could have used the hardware PWM mode, which would give much better precision and guarantee in the generation of the PDM's as there would be no influence from software interrupt latency. Also this would make the design much simpler.

2.8 Conceptual Design: Pan Tilt

2.8.1 Description

The Pan Tilt module is responsible for interfacing and driving the pan tilt mechanism. This primarily consists of generating the PDM signals required to send dictate the position of the servo's.

2.8.2 Overall Design

The overall design of the system is to convert any inputted direction into a set of delays required to generate the PDM's. These PDM's are then realised by timing interrupts running off the calculated delays.

2.8.3 Detailed Design

This module uses a single output compare module to create the delays necessary to generate the PDM's of the desired duty cycle. Due to the interrupt latency of approximately 300μs, the PDM's are staggered so that the interrupt calls will never be closer than approximately 0.04 seconds. The full available duty cycle (1000μs) is divided over the given angular range. There is also an angular offset for calibration reasons. The Delay object is then created to define the necessary delays to create the desired PDM. NOTE: This module is interrupt driven, so the Interrupt.c file must be included in the project for it to work.
A data Flow diagram of the Pan Tilt Module is shown in Fig. **??**.
The conversion from inputted angle to outputted PDM delay is simply to divide the full high time (1000μS to 2000μS) over a stored angular arc. The angular arc along with a reference offset completely define the calibration of he pan tilt mechanism.
The dataflow through the pan tilt module is depicted in Fig. 2.3.

2.8.4 Implemented Functionality

Implemented Basic Functionality

The Pan Tilt module includes the following basic functionality:

- Configure function to set up the module
- Move function to move to any valid position
- Get Direction function to return the current direction

Implemented Additional Functionality

The following additional functionality has been implemented for the Pan Tilt Module

- Incremental move function

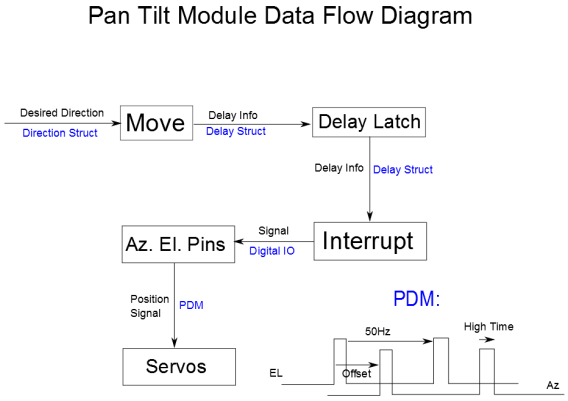

Figure 2.3: Dataflow diagram showing the transition from inputted direction to servo direction

- Increment fine function for greater precision
- Updated function to indicate whether a new delay setting has actually been written into the system yet

2.8.5 Assumptions Made

The largest assumption made is that the clock frequencies in the code match those of the actual clock. Common.h defines the clock frequencies of the PIC-DEM and Minimal Boards which can be switched between. But there is no way for the system to verify the frequency of the clock, and if the clock is not the same then the PDM frequency will not be 50Hz, which can damage the servo's if faster.
The module does not assume that the interrupts fire instantly after the timing event, but that there is an approximately constant latency time, which is found experimentally, included as a # define, and tuned.

2.8.6 Constraints on Pan Tilt Performance

The Pan Tilt module is restricted to the servo range of motion. Also the actual position is entirely dependant on the calibration as they merely take a PDM input.

2.8.7 Interface

Refer to the Interface Manual For detailed explanations of the interface functions and how to use them.

List of Inputs

The Pan Tilt Module takes the following inputs:

- Absolute Direction to which to move (as Direction struct)
- Incremental Direction in which to move (as Direction struct)
- Calibration Directions (as direction structs)
- Configurable system settings as chars

List of Outputs

The Pan Tilt Module returns the following outputs:

- The current direction (as Direction struct)
- Position commands to servo's (as PDM's)
- Current configurable system settings values (as chars)

2.9 Module Requirements: Range

2.9.1 Description

The range module uses the IR, Ultrasonic and temperature sensors to take range measurements.

2.9.2 Hardware

The range module makes use of the following hardware:

- 600 series Polaroid Ultrasonic range sensor
- TL851 Sonar Ranging Control
- Infra Red range sensor

2.9.3 Functional Requirements

Inputs

The range module takes no real inputs

Processes

The range module requires the following processes to function properly

- Sample the Ultrasonic and IR sensors
- Convert the sensor output to a range
- Fuse the ranges from different sensors

Outputs

The output of the range module is the range to the target (in mm) as an unsigned int. The module also returns the state of the detected target as an enumeration.

Timing

The range module uses an input capture module to record the precise time the ultrasonic echo signal returns, which is then used for the range calculation. An interrupt is then used to indicate that the echo has returned, but precise timing for this is not required, as the echo return is stored in hardware when the echo is detected.

Failure Modes

The system waits for the echo signal to return, but there is a possibility that the echo will never return. For this reason the system has a timeout feature so that if the signal does not return within a specified timeout then the range module returns nothing found. This means that the range module does not get stuck in the range module waiting for the echo signal to return.

Implemented Basic Functionality

The range module implements the follow functionality:

- Configure the range module
- Return the range to the target

2.9.4 Non-Functional Requirements

Performance

Interfaces

Refer to the Technical Users Guide for a detailed explanation of the module interface

Implemented Additional Features

The range module also implements the following additional functionality

- Fuse the ranges taking the distance into account - US better for long ranges, IR better for short
- Categorise the target state based on which sensors return a reading
- Range calibrating functions

2.10 Conceptual Design: Range

The Ultrasonic sensor is interrupt driven, which means that we can be performing other actions while waiting for the echo return. Thus the system uses this time to sample the IR and temperature. The module also allows any number of ultrasonic samples per range estimate, and the IR sensor is sampled continuously while the ultrasonic sensor is being sampled. The module also fuses the ranges returned by the respective sensors based on the range, so that IR is used more at short ranges, and not at all at long ranges. The module also sets a target state that can take a number of states depending on which sensors detect a target. This means the module can differentiate between when the sensors are within the Ultrasonic cone, but not in the IR, or when it is within the Ultrasonic cone, but out of IR range. This is then used for the searching and tracking.

Assumptions Made

Interface

2.11 Interrupts

2.11.1 Description

The PIC18F4520 has 2 ISR's, so an interrupt framework whereby each module defines an ISR function, and a macro which determines whether to call its ISR function. The ISR's are then included in their own file, and included here as a semi-module

2.11.2 Functional Requirements

Inputs

For the interrupt framework to function each module must define a macro which checks the interrupt flags for all the interrupts associated with the module.

Processes

The interrupt framework must be capable of performing the following processes:

- Check each module interrupt macro
- Ability to call the associated function depending on macro results

Outputs

The interrupt framework has no real outputs, but directs program execution to the appropriate module ISR

Timing

There are no hard timing requirements for the interrupt framework itself, but the high priority interrupts in particular need to be called as soon after the event as possible.

Implemented Basic Functionality

The interrupt framework includes the following basic functionality:

- Ability to distribute interrupt execution to any module
- Priorities (high and low)

2.11.3 Non-Functional Requirements

Performance

The following characteristics are desirable for performance reasons:

- All interrupts should be as fast and efficient as possible so as to interfere with background code, and other interrupts as little as possible.
- Any extended functionality should be placed into functions not called by the interrupts
- Use of priorities to ensure precision for some modules such as the pan tilt which rely on it

Implemented Additional Features

There are no additional features implemented for the interrupt framework

2.11.4 Conceptual Design

For convenience interrupt macros have been defined in Common.h for each possible interrupt call. Querying one of these macros will return true if that interrupt fired an interrupt. These macros look like: TX_INT, CCP1_INT.
Each module defines a macro (in its header file) which indicates which interrupts that module is using, by or-ing together the previously mentioned macros. When an interrupt fires it checks these macros for each module, and if true calls a 'module scope ISR'. These macros look like: SERIAL_INT, RANGE_INT etc. These module scope ISR's are just functions defined within each interrupt driven module, which are called whenever an interrupt associated with that macro is called. Thus these functions act as ISR's for the module without conflicting with anything else in the program.

Assumptions Made

Interface

There is no public interface for the interrupt framework and nothing can call any of its functions. However other modules must define a macro and a service routine which are used in the interrupt module

2.12 Module Requirements: Circular Buffers

2.12.1 Functional Requirements

Inputs

In order to function correctly the circular buffers sub-module must be capable of accepting the following inputs:

- characters to push onto a buffer
- a buffer to operate on

Processes

In order to function correctly the circular buffers sub-module must be capable of performing the following processes:

- Initialise buffer
- Increment and modulus pointers
- Push characters
- Pop characters
- Peek characters
- Full and empty checking functionality

Outputs

To function correctly the circular buffers sub-module must be capable of returning the following outputs:

- Characters stored in the buffer
- The current filled state of the buffer (full or empty)

Failure Modes

The circular buffer functionality has some inbuilt safe-guards against accidental misuse such as:

- Cannot pop an empty buffer
- Pushing full buffer overwrites oldest data

Unfortunately the C language does not facilitate private object elements, so the user program could manually alter any elements in the buffer, which could likely result in errors. This could be remedied by making circular buffers a full module, with an actual source file, although this would reduce efficiency.

2.13 Conceptual Design: Circular Buffers

2.13.1 Description

The circular buffers sub-module is a header containing everything required to utilise the circular buffer functionality.

2.13.2 Overall Design

Circular buffers are a mechanism used in several different modules, and for this reason the circular buffer definitions and functionality are included in a sub-module like fashion. This consists only of an include-able header which contains all the struct definitions and macro defines needed to fully utilise the buffers.

2.13.3 Detailed Design

The circular buffers header contains the circularBuffer struct definition, which is essentially an array with head and tail pointers which tell the system where to insert and read from the buffer. The header also defines the BUFFERLENGTH (which can be re-defined), and macros such as incMod to increment and modulus with the BUFFERLENGTH. It then contains all the standard circular buffer functionality such as push, pop, peek, empty functionality in the form of efficient macros, specifically designed to induce as little latency in the serial interrupts as possible.
This implementation of circular buffers overcomes the problem of differentiating between full and empty states by not incrementing the tail pointer if it would coincide with the head pointer, essentially reducing the buffer size by one.
Such a design facilitates speed, while still allowing the entire functionality to be alterable and update-able from a single place.

2.13.4 Implemented Functionality

The circular buffer sub-module implements the following functionality:

- incMod
- empty
- full
- peek
- push
- pop
- init - Initialises (clears) the circular buffer

2.13.5 Interface

The circular buffers sub-module has no real interface with the entire functionality being defined in the header to be included.

2.14 Module Requirements: Temp

2.14.1 Functional Requirements

Inputs

The temp module must be capable of taking the following inputs:

- Reference temperature to which to calibrate

Processes

The temp module must be capable of the following processes:

- Configuring the system to sample the temperature
- Sampling the temperature sensor
- converting the sensor output to celcius
- Calibrating the temperature sensor

Outputs

The temp module must be capable of returning the following outputs:

- The temperature in Degrees
- The raw temperature in Degrees

2.14.2 Non-Functional Requirements

Performance

For performance the temperature module stores the last temperature sampled, so that any call to the temperature module can simply return this value, meaning the temperature need only be sampled once a minute or so as the temperature does not vary quickly.

Interfaces

Please refer to the attached Module Descriptions document

2.15 Conceptual Design: Temp

2.15.1 Description

The temperature module is responsible for sampling, storing and calibrating the temperature sensor, primarily for the ultrasonic range calculation and user output.

2.15.2 Overall/Detailed Design

The system revolves around the temperature sampling. There is a static offset which is calculated from the calibration function and gets added to any sensor reading. The static variable in the temperature module (storing the last temperature sampled) gets updated every time the temperature is read. Calling the getTemp() function simply returns this value without sampling the temperature.

2.15.3 Implemented Functionality

The temperature module implements the following functionality:

- Temperature reading
- Calibrating temperature
- Raw temperature
- configure temp module

2.15.4 Assumptions Made

The primary assumption made in the temp module is that the system temperature will not change very quickly, which means that the temperature does not need to be sampled each time the temperature value is used. Instead it simply uses a previously sampled value, and ressamples the temperature at semi regular intervals.

2.15.5 Constraints on Pan Tilt Performance

The main constraint on the performance of the temperature module is the accuracy of the temperature sensor itself, which is only accurate to within half a degree Celcius.

2.15.6 Hardware

The temperature module makes use of the following hardware:

- Temperature sensor

Originally amplifiers were planned to set the analogue to digital converter to the same voltage range as the output of the temperature sensor. But this was deemed unnecessary because the resolution of the ADC was already greater than that of the temperature sensor.

Pin Assignments

The following pins are used to connect to the temperature module hardware:

- Output of Temp sensor to AN1 Pin
- Temp V_{CC} to general purpose 5v bus
- Temp GND to common ground

2.15.7 Interface

Please refer to the attached Module Descriptions document

2.16 Module Requirements: Tracking

2.16.1 Functional Requirements

Inputs

There are no real external inputs to the tracking module, as the functions are iterative they perform a single iteration and return. Thus they are simply called continuously whenever the system is detecting a target. The following inputs are requested by the tracking module from other modules such as range:

- Current system state
- Target Range
- Target State
- Pan Tilt Direction

Processes

The tracking module is required to perform the following processes:

- Move the Pan Tilt module in a search pattern
- Use sensor data to detect targets
- Use algorithms and sensors to track moving targets

Outputs

The tracking module is required to return the following outputs:

- The target details (range, azimuth, elevation)
- whether it still has a target
- the next system state

2.16.2 Non-Functional Requirements

Performance

In order to perform well (defined as accurate and reliable track) the system should have the following characteristics:

- Quite fast tracing step
- Fine adjustments when target is fixed
- Coarse adjustments when target is unknown
- Efficient tracking algorithm

Interfaces

The system should have the following interface characteristics:

-

2.17 Conceptual Design: Tracking

2.17.1 Description

The tracking module contains the high level tracking and search algorithms used by the system. It then uses the Pan Tilt and Range modules to enact these modules.

Overall Design

The searching algorithm uses a simple raster scan variant which scans the azimuthal range, then increments (or decrements) the elevation and scans back the other way. When the system detects any object with either the ultrasonic or the infra red then it enters the tracking mode.
The tracking mode uses a simple weighted average technique, where the system increments the pan tilt direction up, down, left and right of the know target, and averages the directions based on the returns it samples at each of those points. The result is the new target position.

Detailed Design

The Tracking module contains two primary functions: search and track. Each of these functions is 'incremental' in nature. This means a single call will perform a single 'step', and these functions are called continuously to perform a constant action.
The search algorithm is based on a simple raster like scan. The system simply increments sweeps the azimuth, incrementing the elevation each time it hits the max or min azimuth. If it gets to the limits of the elevation then it reverses the direction of its increment variable, making the system scan the other way.
As mentioned previously the tracking algorithm is a simple weighted average. The algorithm is highly configurable with surprising ease, but currently it is configured so as to take sampled above, below, to the left and to the right of the know target location. If the system gets an ultrasonic return (i.e. in the large ultrasonic cone) then it assigns a weighting of 1, adding that direction to the sum. If it returns an IR reading it multiplies the direction by 8, and adds it, no return has no weighting. Then it divides by the sum of the weights to get the new target location.
This means that if it is far from the target, then only 1 direction will get returns and the system will necessarily move toward the target. If it is almost on the target, then the preferential return of ultrasonic and/or IR samples to one side will serve to tune the know position.

Assumptions Made

There are a number of assumptions made in this module these are as follows:

- The searching algorithm does not 'miss' any targets (i.e. scans faster than they can move)
- The target is approximately stationary compared to the speed of the tracking algorithm (so that all 5 samples can be considered for the target at a fixed location)
- That the sensors do not exhibit any false positives or false negatives

Some of these assumptions may seem a little ambitious, however, in a real world scenario they seem to be sufficient to create a working system, albeit not an ideal one.

Constraints on the Tracking Performance

Below are some of the more major constraints on the performance of the tracking algorithm:

- The maximum sampling rate of the ulrasonic - limited the overall tracking speed
- The resolution of the servo's - made it difficult to accurately track distant targets

Hardware

The tracking system itself made use of no dedicated hardware, it was simply a high level software module, however it did make use of many other modules utilising a range of hardware, such as the range and pan tilt modules.

Interface

Refer to the attached Module Descriptions document

2.18 Module Requirements: Local User Interface

2.18.1 Functional Requirements

For a functioning local user interface, the following requirements must be met.

Inputs

- The user must be able to select options suitable to the system in an intuitive manner. This can be done using a potentiometer, encoder, or left/right buttons. This system uses a potentiometer which the user can rotate to choose an option.
- The user must be able to confirm/select their input using a button on the device.
- The user must be able to cancel and return from their selection using a button on the device.
- The buttons must only trigger a signal once when pressed, and should not incorrectly return a signal. This can be done using a de-bouncing circuit and a pull-down resistor.

Processes

The Local User Interface module must map the physical actions of the user to software outputs which the menu system can analyse and use. The necessary processes are:

- Adding button presses to a circular buffer to be retrieved and handled by the menu system so that order is conserved.
- Indicating when the buffer is empty, or contains an escape character.
- Process the potentiometer value from the microcontroller's ADC, and scale the value into discrete steps for a given number of steps.

Outputs

- The module must be able to indicate when a confirm or escape button is pressed.
- The module must be able to indicate which button was pressed, and the order buttons are pressed.
- The module must be able to output the converted value from the potentiometer.
- The hardware for the interface must also contain the physical LCD module and the circuitry needed to support it.

Timing and Performance

The local interface must act immediately when a button is pressed by using an interrupt subroutine, so that user may press buttons as fast as they would like. Readings from the potentiometer must be performed at a fast enough pace so that the user does not notice an unresponsive system. This is usually on the order of approximately 10 samples per second.

2.18.2 Non-Functional Requirements

Interface

Aside from functional requirements, there are other requirements for the system to operate the best it can. In terms of hardware, the local user interface must be ergonomic and robust for the user to easily be able to handle the device. The system should also be adequately labelled so that any user can use the system without prior training.

The software interface must be intuitive, and be based upon the circular buffer module used above.

Design Restraints

The local user interface must be constructed from available and moderately priced components. This is so that different prototypes of the user interface can be constructed cheaply, and in the future if a client is in need of repairs, the time and monetary cost of repairing the device is not expensive.

As per the other modules, the software for the local interface must be programmed in C and run on the PIC18F4520 microcontroller. Due to the size of this module, it is ideal that the module is lightweight and can be built upon the circular buffer module.

2.19 Conceptual Design: Local User Interface

2.19.1 Hardware

The local interface design uses a potentiometer and two buttons to act as the control scheme for the system. The two button approach had been a design choice since the early stages of the project, as having a confirm/select button and a cancel/return button is common on many systems, and is very intuitive.

For selecting a value, multiple input peripherals were considered, such as a potentiometer, rotary encoder, and keypad. For the final selection, the potentiometer had been chosen as in the input device, as team members were more familiar with how to write software using one, where the encoder and keypad would require more time and energy invested into it to the same response.

The design for the local user interface hardware began initially with a one box approach, with the LCD and buttons placed near the servo motors. However, our second design included a controller based approach so that the user can freely operate the controls free from moving the sensors. Following this style of approach, we designed the enclosure to be hand-held like a controller, with the buttons and potentiometer being easily accessible to the user's thumbs.

2.19.2 Software

The software model for the Local User interface is very similar to the serial module. As buttons are pressed, they get entered into a circular buffer and processed from the oldest button press to the newest. When the escape button is pressed, a separate flag is raised so that the system can immediately return from it's current menu.

A button press is indicated by using the external interrupts, which are connected to pins RB0 and RB1 on Port B, with a different interrupt for each button. These were set up to trigger on a rising edge of a new signal, and did not seem to have any bouncing issues, so a bouncing circuit was not necessary. A pull-down resistor was still added to make sure TTL logic levels were still achieved.

For the potentiometer, a simple division algorithm was used to divide the ADC output from a number between 0 and 1023, to a number between 0 and

a value dependant on each menu. This splits the input into a number of even sections to be used by the Menu System.

2.20 Module Requirements: Menu Subsystem

2.20.1 Functional Requirements

Inputs

The menu subsystem must be capable of taking the following inputs in order to fulfil its purpose:

- User inputs from the Serial and Local Interface modules
- A pointer to a tracking state to be initialised to, so that the menu can modify or read whether the system is tracking, searching, or waiting for user input.
- Inputs from the Range and PanTilt modules for the current value of the system settings that a user can modify.
- Digital logic level from a physical key switch attached to a pull-down resistor circuit to signal Factory mode is to be used.

Processes

The menu subsystem must be capable of performing the following processes in order to fulfil its purpose:

- The menu system must be able to switch between a local, remote serial and factory serial user mode.
- Parse and convert user data from the Serial and Local User Interface modules into integer values which the menu system and other modules can effectively use.
- Switch to another menu when commanded to by the user.
- Convert system values to ASCII messages to display.
- Convert messages in the system's ROM memory bank to RAM so that the Serial module can display them.

Outputs

The menu subsystem must be capable of returning the following outputs in order to fulfil its purpose:

- Messages and structure for the LCD module to display to show the local user interface.

- Messages for the Serial module to display to form the serial menus.
- Error messages when the user inputs incorrect values.
- User input values to the corresponding modules to change system settings.

2.20.2 Non-Functional Requirements

Performance

The menu subsystem should have the following characteristics in order to operate efficiently:

- The system should be able to quickly respond to user input and quickly display messages, so that the user does not view the system as unresponsive and broken. A response time of 100 milliseconds or less is desired.
- The serial interface for the user must be easy to understand and use commonly used keys or commands.
- The user should be able to intuitively know where options are, and how to navigate between different menus.
- The menus should be aesthetically pleasing and consistent between menus.

Interfaces

The menu subsystem should have the following interface characteristics in order to operate efficiently:

- The menu system must be able to access the functions from the Range and PanTilt modules which can access and change system variables, such as the maximum system range.
- Tracking data from the Tracking module should be easily available to the menu system so that it can display the data to the user.
- Range data from the infra-red and ultrasound sensors should be easily available to be displayed in the "Show Raw Data" menu.

2.21 Conceptual Design: Menu Subsystem

2.21.1 Description

The menu subsystem is responsible for coordinating all the user inputs and outputs, storing the current menu state, and all the system strings and prompts to send to the users. It also interfaces with the other modules to change system values and relay important information to the user. In essence, it turns the basic Hardware interface modules that are the serial and LCD modules and uses them to create the user interface to the entire system.

2.21.2 Overall Design

The menu system operates based on a handful of general functions that get called and do different things based on what the current menu is. Every cycle of the main program, the menu gets serviced, which involves the following events. First, there is a check for if the user has submitted any commands over serial for Remote/Factory users, or via the local interface for local users. If so, the input is read and converted to an integer, which is then used differently in each menu.

The concept of the menu system revolves are instances of a menu object, which contain function pointers to other functions, so that a general function can be called from the main loop of the program. This was implemented after a menu system was created which used a series of long switch statements, however these were found to be expensive to the program memory, and hence the system was redesigned.

2.21.3 Detailed Design

Menu Objects

Each menu object contains a list of variables which is necessary for the menu to interact with the current system. These will be discussed below.

Menu ID: Each menu is given an identification number which is unique to each menu. This is used to identify which menu the current menu is within switch statements, as the menu objects themselves canot be compared easily.

Serial Message: Each menu contains a message to be displayed when the menu is first entered. These messages are often instructions to the user as to how to operate this specific module, and are often unique to each menu.

LCD Title: Each menu has a title, which is often within the 16 character limit on the LCD so that it can be used for both serial displays and LCD displays.

Minimum Value: The minimum value integer represents the minimum input value that the user can enter for this menu. This is often 1 for menus which navigate to sub-menus, but is also acts as the lower bound for any menus which allow the user to change a system setting.

Maximum Value: The maximum value integer represents the maximum input value that the user can enter for this menu. For menus which navigate to sub-menus, this value is the number of options the user has to navigate. This value also acts as the upper bound for any menus which allow the user to change a system setting.

Increment: The increment value is used to indicate the value of the offset used when counting from the minimum value to the maximum value. This dictates how many possible states there are between the two boundary numbers. For a navigation menu, this is always 1. However for other options, especially for changing the range of the device, the increment value is quite large at 50 millimetres per division. This value is necessary for the local user interface as the potentiometer can only rotate within a fixed arc, so decreasing the number of possible states allows the user to sacrifice accuracy for being able to choose an approximate value in a much less finicky manner.

Serial Display Function This is a function pointer used to indicate which function should be called when the menu is entered via a serial command. There is a generic method which only prints the serial message variable from above, however other alternatives are created for displaying navigation menus with long lists of possible choices, and displaying messages for commands which set system variables, as the upper and lower bounds are also displayed, along with the setting's current value.

LCD Display Function Another function pointer, this time used to indicate how the menu is to display text on the second line of the LCD hardware, depending on the current position of the potentiometer. The user does not have to confirm their selection for this text to appear or change, but this function merely shows them what they may want to select. For instance, this is used to allow the user to cycle through sub-menu choices as they navigate the menu system. This function takes an integer representation as to what the potentiometer is currently pointing at, moved to within the bounds of the maximum and minimum values.

Confirm Function The confirm function handles how the menu behaves once a user has selected the value they would like, whether by the serial interface or the local interface. These functions also take a numeric input of what the user has selected. These are broken down into two sets of functions.

The first is used when the user is trying to change a system setting. The function uses the menu's ID to call the corresponding function in another module, such as calling the "setMaxRange(int)" function from the Range module if the menu was for setting the maximum range. These are done using a switch statement, rather than using many very similar small functions, as the memory usage was approximately the same, and a switch statement assists in making code easier to read.

The second type of confirm function is for the navigation menus, which change the current menu depending on the user's input. The act of changing the menu changes a system variable internally, as well as clearing the serial display, and calling the Serial Display function so that the new menu is shown. Changing the menu also changes the message on the top level of the LCD. Using this method allows the menus to not be called directly from within each other,

which reduces the load on the program stack, as the current menu finishes before the next menu is set.

Return Function This function simply allows the user to navigate up a level in the menu by setting the menu to whichever menu should be above it in the menu hierarchy. These functions are accessed by the user pressing the back key on the local interface, or the escape key (or the on-screen option) in the serial interface. These are hard coded so that one specific function always returns to each navigation menu, as having a general function passing in an argument did not seem to have a great effect on the system's memory usage.

User inputs

Talk about why we use numbers, intToAscii, asciiToInt etc.

User inputs were designed in a manner so that inputs from both the serial interface and the local interface could be handled in one place in the same way, to keep the user experience consistent and reduce memory load on the microcontroller, at the cost of a small amount of processing time. The local user interface's simple input design meant that input information was limited to two commands for confirming and returning, and a range of values on a potentiometer.The values from the potentiometer in the Local Interface module map a result to a number of discrete steps. These steps are then scaled to the appropriate values for this menu by passing in the total number of discrete states by the following formula:

$$NumberOfStates = (maxVal - minVal)/increment \tag{2.1}$$

These discrete states are then scales back to the minimum and maximum numbers by:

$$Value = minVal + (increment * adcResult) \tag{2.2}$$

To set up a similar system on the serial interface, the user controls were bound so that the enter key acted in the same way as the confirm button on the local interface, and similarly the escape key was mapped to the return button. These two keys were chosen as they are used often by computer programs for confirming and cancelling values that the user may want to enter, and are isolated from the input step for giving commands.

The user must be able to enter commands to be confirmed, and this is achieved on the local interface by turning the potentiometer to receive an integer value. This could not be done on the serial interface, so instead a numbered list is presented to the user upon entering a menu. This serves three purposes: the user can easily see exactly which commands are available to them from the current menu, the user does not need to memorise text commands to user the system, and finally ASCII number input is straight forward to convert into the integer value that the menus are expecting.

To convert to and from integers to ASCII character strings, a custom function was written to handle user input, and the itoa and sprintf functions from the c library were used to form the string equivalent of the number. The created function separates the user input into each digit, checks for a negative sign, then multiplies each digit by the corresponding power of ten to sum to the same number in the integer format.

Error Handling

Most errors in the user interface stem from the user inputting the incorrect value into the system. This can only happen in the serial interface, as the local interface input is already constricted to correct values. To handle incorrect user input, the user is first instructed as to how to correctly give the input over the system, and if an incorrect number is entered, an error message is displayed asking the user to input a number within the minimum and maximum values for that menu. This error case also handles any non-numeric input received, as this is detected during the input parsing step.

Factory Mode

Factory mode functions were added to their appropriate navigation menus once the user had switched to factory mode. The user interface only showed these settings and allowed these settings to be selectable when in factory mode. Switching between factory mode and remote user mode was performed by inserting and turning a key into the device. This process was set to always bring the user back to the main menu, as it firstly reset the system and refreshed the interface to show the user the new options, but also did not allow the user to switch out of factory mode whilst in a factory setting, which would have unexpected side effects.

2.21.4 Assumptions

The menu module assumes that the lower level modules are functioning correctly, especially the Serial and Local User Interface modules. If these modules do not function as designed, the menu would not be able to interact with the user as intended, as the side effects of a user's command would be unknown.

Secondly, the menu system also assumes that the hardware is connected properly. In Remote mode, it is assumed that a RS-232 cample is connected to the system, and to a computer running a terminal program with a serial transmission rate of 9600 bites per second. In Local mode, it is under the assumption that the local interface controller is connected and not damaged.

2.21.5 Constraints On the Menu Subsystem

As mentioned in the assumptions section above, as the Menu system is the highest level of abstraction for the user, if any module beneath it does not

operate correctly, the menu system would also not perform as expected, as the user would attempt to control the system via the menu, and another action would be taken.

2.21.6 Interface

See the Interface Manual document for a full list of the public functions from this module.

2.22 Module Requirements: LCD

2.22.1 Functional Requirements

Inputs

The LCD module requires three internal inputs written to operate effectively, a command (what operation is required to execute) or data, the command type (data or control instruction) and whether it is a read or write instruction.
This information is encoded in two control pins and eight data pins, Register Select, Read/Write and data. Once the instruction mode is determined, the command/data is decoded. The command set and its corresponding options are encoded with the most significant numbered bit determining the command, and the following bits are the particular settings altered.
These commands and data read/writes require specific understanding of the LCD and it's associated hardware, so for high level use and protection against input errors, two subroutines were written, a string write and a character write. The configuration settings were pre-configured and kept internal to the module to ensure no accidental changes could be made.

Processes

The LCD module requires the following processes to function properly

- Loop through a given string, writing individual characters
- Appropriately increment the data address location for character display

Outputs

The LCD system is an output device and is required to consistently and clearly display requested characters for the user. It must be able to display the characters/strings when user input calls them, and do so without noticeable delay or flicker.
The LCD is to display information on a 5x7 dot matrix, within a 2x16 limit.

Timing

The LCD has two timing requirements, the first being in regards to usability. The display must be able to update at a rate so the user does not experience any latency. This is achieved by implementing short write sequences with minimal computation.
Secondly, as the LCD processor has no internal 'queue', timing for instructions must be met to ensure there is no command overrun error. This can be achieved in two ways, calling the LCDbusy() subroutine, which waits until the current instruction has executed, or more simply, byu implementing a delay function. The latter was chosen, as errors with the busy function could not be overcome, and the delay function was simple to implement.

2.22.2 Implemented Basic Functionality

This module is able to write any character or string to the LCD display effectively, easily and without noticeable delay.

2.23 Non-Functional Requirements

Performance

The LCD should be able to display required characters without delay and correctly. It should also be adaptable and easy to use, such that no technical knowledge is required to call the write function.

Interfaces

The external interfaces to this module are three subroutines, configLCD, lcdWriteChar and lcdWriteString. Through these functions, the system can interface with the entire program and the external environment.

2.24 Conceptual Design: LCD

2.24.1 Description

The LCD module is designed to perform all interfacing with the LCD, as well as formatting inputed strings etc.

2.24.2 Hardware

The LCD module makes use of the following hardware:

- 1602 LCD

Pin Assignments

Pin No.	Label	Description
1	Vss	Ground
2	Vdd	Supply Voltage
3	VO	Contrast adjustment voltage
4	RS	Register select signal
5	R/W	Read / write select signal
6	E	Operation (read/write) enable
7	DB0	Low byte data bit
8	DB1	Low byte data bit
9	DB2	Low byte data bit
10	DB3	Low byte data bit
11	DB4	High byte data bit
12	DB5	High byte data bit
13	DB6	High byte data bit
14	DB7	High byte data bit
15	A	Positive LED backlight (Anode)*
16	K	Negative LED backlight (Cathode)*

* Backlighting connections for Z 7011 Only

Figure 2.4: LCD Pin Assignments - shows the meaning of each of the pins on the 1602 LCD

Pin Assignments

Fig. 2.4 shows the meaning of each of the pins on the LCD hardware package. Fig. 2.5 shows which microcontroller pins are connected to the LCD. As shown in Fig. 2.5 we have used the LCD 8 bit mode for simplicity and efficiency as we have no shortage of free digital pins. Were the system more complex, and digital pins were in shorter supply this could be changed to the 4 pin mode.
The data pins DB0-DB7 are connected to the whole PORTD and the control pins (RS, R/W and E) are connected to RC4, RC5 and RA4. VL which is connected to the GND via a 10K ohms potentiometer is the contrast voltage pin which controls the bias voltage. The BLA and BLK are the backlights pins which are not connected for our LCD since there is no backlights in our LCD mode.

2.24.3 Software

Timing

Fig. 2.6 shows the timing required to correctly interface with the LCD hardware.

Reading Operation

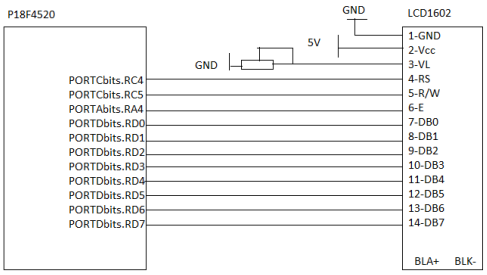

Figure 2.5: LCD Wiring Diagram - Shows how the 1602 LCD is wired up, and which pins are used to interface to it using the 8 bit mode

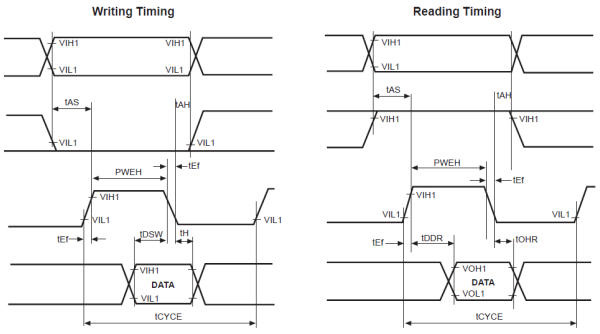

Figure 2.6: The LCD module requires precise timing, particularly for the initial configuration. This figure shows a timing diagram for the device taken from the datasheet

Command	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Remarks
Display Clear	0	0	0	0	0	0	0	0	0	1	Clears Display
Return Home	0	0	0	0	0	0	0	0	1	X	Cursor Moves to 1st Digit
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	SH	I/D=0 Cursor moves left I/D=1 Cursor moves right SH=0 Display is not shifted SH=1 Display is shifted
Display ON/OFF	0	0	0	0	0	0	1	D	C	B	D=0 Display off, D=1 Display on C=0 Cursor on, C=1 Cursor off B=0 Blinking off, B=1 Blinking on
Display Shifting & Cursor Motion	0	0	0	0	0	1	S/C	R/L	X	X	SC=0 Cursor moves, SC=1 Display shifts R/L=0 Left shift, R/L=1 Right shift
Set Display Function	0	0	0	0	1	DL	N	F	X	X	DL=0 4 bit interface, DL=1 8 bit interface N=0 1 line display, N=1 2 line display F=0 5x7 dots, F=1 5x10 dots
Set CGRAM Address	0	0	0	1	CG5	CG4	CG3	CG2	CG1	CG0	DB5-DB0 : CGRAM Address CGRAM Add. corresponds to Cursor Add.
Set DDRAM Address	0	0	1	DD6	DD5	DD4	DD3	DD2	DD1	DD0	DB6-DB0 : DDRAM Address
Read Busy Flag & Address Ctr	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	DB6-DB0 : Address Counter (AC) BF=0 Ready, BF=1 Busy
Write Data to CGRAM / DDRAM	1	0	DATA7	DATA6	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0	DB7-DB0 : Data Bits for Write
Read Data CGRAM / DDRAM	1	1	DATA7	DATA6	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0	DB7-DB0 : Data Bits Read

Figure 2.7: LCD Control and Display Commands

- Reading the commands: RS=0, R/W=1, E=1 to 0. When the pin E jumps from the high level voltage to the low level voltage, the commands can be read from the LCD data bus.
- Reading the characters: RS=1, R/W=1, E=1 to 0. When the pin E jumps from the high level voltage to the low level voltage, the characters can be read from the LCD data bus.

Writing Operation

- Writing the commands: RS=0, R/W=0, E=1 to 0. When the pin E jumps from the high level voltage to the low level voltage, the commands can be written to the LCD display module.
- Writing the characters: RS=1, R/W=0, E=1 to 0. When the pin E jumps from the high level voltage to the low level voltage, the characters can be written to the LCD display module.

Programming the LCD

Fig. 2.7 shows the interface of how to send commands to the LCD.

Writing the commands to the LCD First check whether the Busy Flag is 0. If it is 1, stay and wait until it changes to 0; if it is 0, set the RW and RS to 0 and the E to 1 first, let PORTD equal to the command, have a proper delay and set the E to 0 then the command is written in.

Writing the characters to the LCD First check whether the Busy Flag is 0. If it is 1, stay and wait until it changes to 0; If it is 0, set the RW to 0, the RS and E to 1 first, let PORTD equal to the ASCII of the character, have a proper delay and set the E to 0 then the character is written in.

Checking the Busy Flag DB7 (connected to PORTD7) is the Busy Flag bit which is used to check whether the LCD is busy. DB7 is 1 presents that the LCD is busy and cannot receive any commands or data sent from the controller. Set the RW and RS to 0 and E to 1 following with a small delay. Check if the busy flag is 1, set the E to low and return 1 which means the LCD is not ready. Otherwise, set E to low and return 0 which means the LCD is ready for receiving the data.

Configuring the LCD Set all the pins to output and set all the control pins to 0. A configuring procedure can be:

- Write 0x38, which sets the 1602 to 8 bits, 2 lines and 5*7 dots
- Write 0x0F, which sets the display on, cursor on and blinking on
- Write 0x06, which sets the 1602 as when write in a new character, the cursor shifts rightwards and the screen doesn't shift.
- Write 0b01, Clear the screen, set the cursor to the first digit and set the address counter to 0.

It is worth mentioning that since the first 3 commands has a fast operation speed which is about $40*10^{-3}$ ms while the clearing screen command takes around 1 ms. Therefore a proper delay is needed after the clearing screen command is written.

Writing characters to the LCD This subroutine was created to ensure usability and simplicity for LCD character write operations. To use this function, one must:

- Pass the desired character to be written.
- Select the line (1 or 2) to write to.
- Select the column (1 through 16) to write to.

A local variable is set up with a value equal to the selected line. The screen write location is set by calling the instruction write. Next, the data is sent through and the character displayed on the screen. Finally, a delay is called to ensure there is no potential overwrite.

Writing strings to the LCD This subroutine was created so that a user could display an entire string on the LCD without requiring specific technical knowledge. To use this function, one must:

- Pass a pointer to the start of the desired string
- Select the line to display the string (1 or 2).

A local variable is set to the location of the initial write location. A loop is entered which sets the location on the screen to print the current character. The string pointer is incremented and the next character displayed. Finally a delay is initialised and the LCD cursor returned to the start of the line.

2.25 Module: Usability Software (EEPROM)

2.25.1 Description

An EEPROM read/write module was added to the system to increase user friendliness and system deploy-ability. EEPROM allows for values to be saved in a non-volatile memory location, such that when power cycled, the system will retain important calibration settings. This reduces the maintenance required on the system, and limits the needed interactions of technically trained users to change functional

2.25.2 Functional Requirements

The EEPROM module must be able to save values in a non-volatile memory location, and retrieve them at a later stage.

2.25.3 Inputs

The EEPROM module requires inputs of the storage address and the data to be saved.

2.25.4 Processes

The EEPROM module uses the inbuilt C18 EEP.h library to achieve its objectives, utilising the Writebeep and Readbeep functions. These functions require char type data for write input and returns a char for read outputs. As the data being stored will predominantly be int type, this value was broken into high and low part bytes prior to writing and concatenated into an int following reads.

2.25.5 Outputs

Following a successful read from an EEPROM address, an integer is returned and can be used as required in the program.

2.25.6 Implementation

This module was implemented through two public subroutines, sendEep and readEep. SendEep takes an integer number and an integer address, stores the data in two consecutive memory bytes. readEep takes an integer address, reads the value at that address into two characters, concatenates the number and returns an integer value.

Chapter 3

User Interface Design

The Yavin IV ODS has three main user interface points, the remote terminal interface, the local user interface and finally the system housing.
The remote interface is implemented using 'Teraterm' PC terminal software. This point of contact allows the user to enter commands and receive return values via keyboard and screen. The user will view a menu-select style set of options, which allows for a very shallow learning curve. If any input which is not an available command is selected, a friendly error note is displayed indicating the issue to the user.
The remote interface also allows users to configure factory settings if the factory lock is disabled. Factory is only available in remote mode, as it allows for a more developed command system, and an overview of all options.
The local user interface is designed with ease of use and simplicity as it's primary function. The local system consists of an hand-held interface with one output LCD, two input buttons and one potentiometer select dial. This design was inspired by video game controllers which many users will feel comfortable and familiar with.
The button functionality includes "Up one level" on the left of the controller, and "Select" on the right, whilst the dial allows users to cycle through menu and value options. Having only three options ensures that users cannot encounter any input errors, as none are possible.
The system housing encases the important hardware for then system, and thus is completely inaccessible for the user. However, the user interacts with the housing to change the battery, turn on the system, enable factory mode, and connect serial. The battery is changed by opening a compartment on the side of the box, uncapping the battery and replacing it with a new one. The system is turned on by flicking the red switch atop the housing. Factory mode is enabled by inserting the (correct) key into the keyhole atop the box and turning, enabling the electrical connection. Finally, the serial connection is a simple RS-232 female header on the outside of the box.
The entire system was tied together with Rebel Alliance style decals to aid in cohesion and user unity.

3.1 Classes of User

There are two different types of users that this system caters for, end users, and technical users. An end user is the expected operator of the system and is reasoned to have no technical know-how. It is expected that this user will be able to interpret and use the system correctly the first time it is encountered. Due to this, when initialised, the system defaults to this class.
The second class of users are technical users. These are users who are expected to conduct testing, calibration and possible repairs to the system. This user is able to access the factory mode through a physical key switch which enables additional options on the remote interface. These additional options include calibration for the pan-tilt unit, ultrasound and infrared ranging modules and the temperature sensor.

3.2 Interface Design: End User

3.2.1 User Inputs and Outputs

As an end user, there are two possible input and output points, the local hand-held interface and the remote serial interface.
The user may input commands into the system locally by scrolling through options presented on the LCD with the dial, may select options with the select button, and can escape/go up with the "Up a level" button.
The user may input commands into the system remotely by entering numbers via a keyboard which correspond to options displayed by menus on the terminal.

3.2.2 Input Validation and Error Trapping

The system was designed so that minimal errors were possible. Whilst using the local interface, there are no potential errors that can be tripped by user input. This allows the user to feel safe and confident with the product, furthering user-machine cohesion. In the remote interface, a user may inadvertently press a key which does not match any given option, and a friendly error notification is displayed.
In both cases, any 'selection error' (when an input is entered but the user re-asses their choice), the "go up" command/button is available to cancel the choice.

3.3 Interface Design: Technical User

The interfacing for the technical user is no different to that of the end user, there are simply more options available on the remote terminal.

3.4 Menu Design

3.4.1 Menu Structure

Different menu structures were considered for interfacing with the user over a serial connection and via the local interface's LCD screen. Early in product design, we had reached the idea of using a potentiometer or encoder with buttons as the method of user input when not using serial. We had also decided that it was best to share a menu system between the remote and local user modes to keep the system consistent, which helps the user feel lees confused or jarred when switching between user modes.

From these requirements, the menu could have either been designed and implemented in a tiered system with sub-menus to navigate between, or as a large cyclic menu. Due to the limited rotation of a potentiometer, the large cyclic menu concept was rejected, as each option would only have a few degrees of potentiometer rotation, which would make the ability to choose a specific option difficult. A tiered system was chosen for both the local and remote user interfaces. However, concepts from the cyclic design were used for choosing options from within a sub-menu using the potentiometer, as it was more feasible with a limited number of choices per sub-menu.

The next design decision came in how the functions of the system were to be broken up into sub-menus. Possible choices included splitting the functionality between an Autonomous Tracking sub-menu which displayed possible relevant menu options, and Manual Tracking which allowed the user to go to specific azimuth and elevation angles. Another option involved splitting the system menus into Autonomous Tracking, which just showed the user the current range and angles, then spilt the remaining functions based on their areas, such as user functions focused around changing the options to do with the azimuth angle, elevation angle, and range. This was the final design used, as it was more intuitive as to how to find the option that a user specifically wanted. However, this meant that navigating between options in different sub-menus took more time, such as switching between the "Go To Azimuth" and "Go To Elevation" functions meant that the user needed to navigate through 3 sub-menus.

The final decision in menu structure decision that had to be made was where to include the factory options if the system was put into Factory mode. Either all the factory settings could have their own sub-menu which only appeared when Factory mode was entered, or the factory settings could be placed within their relevant menus, such as placing the "Calibrate Azimuth" function inside the azimuth menu. The final descision was made to place the factory settings in their corresponding menus for consistency, as all possible settings to do with one area of the system should consistently be under the same menu. The downside to this choice is again that navigating between different factory settings takes longer than the alternative, and that the factory settings are not easily identified in one location.

The final menu structure can be seen in 3.1.

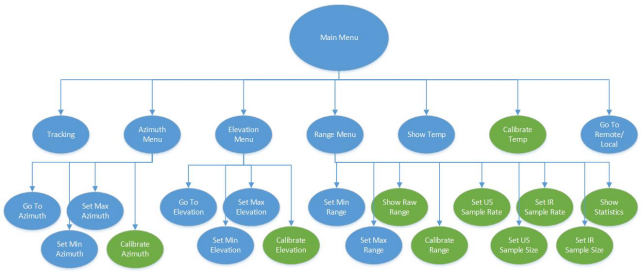

Figure 3.1: Menu Diagram

3.4.2 Menu Appearance

Due to the limited display capabilities of the LCD hardware being used in the system, an intuitive design had to be constructed to convey information to a local user. The LCD module has the ability to write 16 characters to two lines independently. With this, the local user interface was designed so that the top row of the LCD always shows the user their current menu, such as "Azimuth Options" or "Set max range". This allows the second line of the display to be used to show the changes that the user is making, such as displaying sub-menu titles as the user scrolls through potential choices.

The serial interface is less limited by physical constraints, but was also important to design carefully. A similar style of approach was used; the user must always be shown their current location, along with all the possible options that they can take. This is simpler on the serial interface as many options can be displayed at once, so all correct serial inputs are shown for navigation menus, whilst ranges and current values are displayed in functions which change system settings. This allows to the user to more easily make correct choices as to how to interact with the system, and lets the user easily view their current settings.

For the physical appearance of the menu itself in the user's terminal program, the menu itself will always appear in the top left of the program's window. This is done by using a generic clear display command which is common to many terminal programs. This was chosen to reduce the possibility of the serial display looking incorrect in terminal programs other than TeraTerm, in which the serial display was tested, as the program will automatically clear whatever was written previously.

Each menu is framed by a filler character, the plus sign "+". This is done to easily bring attention to the current menu's title, and separate the user input section of the terminal from the printed messages. The menu title and other menu messages are all printed indented to the centre left of the terminal. This is again so that other terminal programs will be more likely to display the user interface correctly, on the small chance that some of the leftmost display is unreadable or not as aesthetically pleasing.

An example of a serial menu can be seen in **??**.

Finally, the certain functional menus such as the "Autonomous Tracking"

menu and the "View Raw Range" menu must be able to refresh the menu as new data is collected. This was performed using another general terminal command, this time to clear the current line. This allowed for data to be constantly overwritten, so that the user is not flooded with tracking data.

Chapter 4

Hardware Design

4.1 Scope of the Local User interface System Hardware

The local user controller consists of an LCD display, 2 buttons and a potentiometer, all mounted inside a black box with a cable running to the main sensor enclosure.

4.2 Hardware Design

4.2.1 Power Supply

Power Source

The system is designed to run off a 9v battery, but can be powered from any 9v power supply rated for 3A. The system is battery operated, and thus does not facilitate any protective earth. There are also no conductive exposed surfaces and the entire thing is encased in an insulating enclosure that is not designed to be open-able by users.
The voltage regulators used had a min/max voltage rating of 6V/20V from testing, which lay within our operating conditions.

Safety Features

Currently the prototype system does not make use of any type of internal fusing or circuit breaker functionality. Should the system be marketed as a commercial product such features would be essential.

Power Regulation

The internal system components run off a regulated 5v bus which can be supplied by any unregulated 9v supply as outlined above. The system contains

two regulators so the ultrasonic sensor has its own regulated power supply to eliminate the 'heartbeat' effect on the voltage of the bus as the ultrasonic sensor draws very high current spikes. A large capacitor (1 milliFarad) was added across the ultrasonic power lines to limit this surge current

The second power rail supplies power to the microcontroller, the servos, the infrared and temperature sensors,as well as to the local user controller interface. This is shown in 4.1

All components are also bypassed with capacitors to avoid such effects, although the ultrasonic sensor called for additional measures.

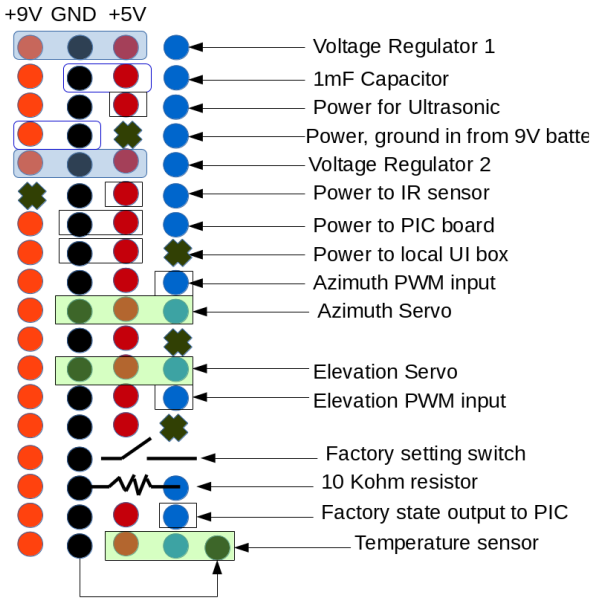

Figure 4.1: Power circuit schematic on veroboard/stripboard

Safety Features

Currently the prototype system does not make use of any type of internal fusing or circuit breaker functionality. Should the system be marketed as a commercial product such features would be essential.

4.2.2 Computer Design

Description of computer hardware, including all interface circuitry to sensors, actuators, and I/O hardware.

Sensor Hardware

The system consists of three sensors.

Temperature (LM35)

The LM35 is a centigrade temperature sensor calibrated directly in Degrees Celsius. Its also has a linear + 10 mV/C scale factor. Only a basic configuration of the temperature sensor was used as we only need to measure between to +2C to +150C as shown in the figure below. We found that 0mv started at +2C so we had to adjust the final calculations by -20mV offset to get accurate temperature readings.

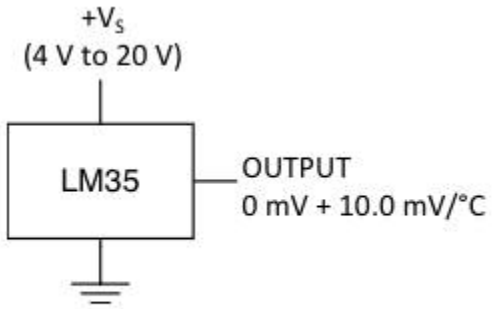

Figure 1. Basic Centigrade Temperature Sensor (+2°C to +150°C)

Figure 4.2: Temperature sensor linear response and wiring diagram

IR [Distance Measuring] (GP2Y0A02YK0F)

The IR sensor used in this system is ideal for distance measuring between 20cm to 150cm. It has a non-linear analogue output that reflects the figure below. This show that if using a 10bit ADC at 5V Reference we would expect a decimal output value of around 512 for 20cm. We could use the below formula to calculate expected values for all points to compare when testing.

Decimal Value = (1024 / 5Vref) * (Output Voltage at given distance)

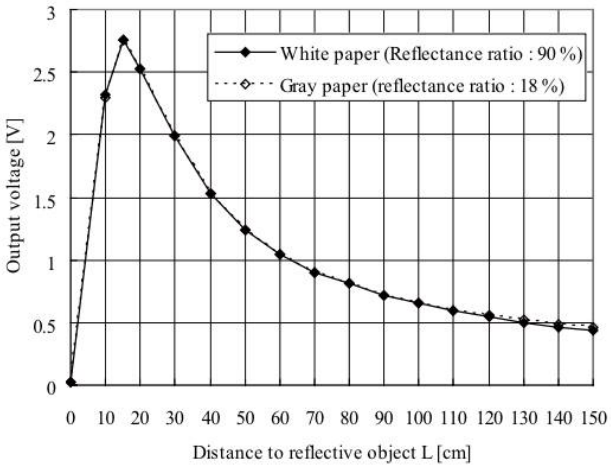

Figure 4.3: Infrared sensor response

Ultrasonic (6500)

This sonar ranging sensor is able to measure distances from 0.15m to 10.67m with a typical 1We used single-echo mode in our system as we were only tracking one object and therefore only need one echo output at a time. Basically the distance between to a target is measured by the time between INIT going high and ECHO going high as seen in the figure below. The signal travels at approximately 0.274 milliseconds per meter.

Actuator Hardware

Operator Input Hardware

The input hardware for the local operator consists of a controller designed to fit most users' hands. Controls consist of 2 buttons, one for entering menu levels, and one for returning to previous menu levels. To select between menu items on the same heriarchy level, a otentiometer control was implemented, to resemble simple controls found in car audio systems to maximize intuitive control of the system. The controller was designed with one wiring harness consisting of 16 wires exiting in a insulated pipe to the main sensor hardwae enclosure, so that the wires would be protected from excessive physical stresses. The 16 wires consisted of 11 wires for the LCD (8 data wires, 3 control wires), 3 input capture wires for the 2 buttons and potentiometer respectively, and 2 power wires (5V, GND) to provide power to all components in the enclosure.

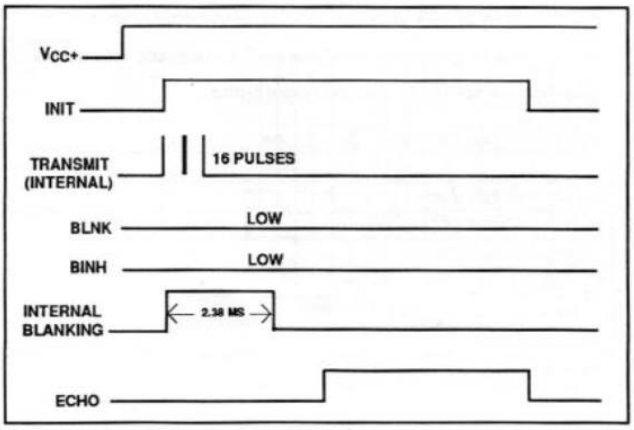

FIGURE 1: EXAMPLE OF A SINGLE-ECHO-MODE CYCLE WITHOUT BLANKING INPUT

Figure 4.4: Ultrasonic sensor response and timing diagram

Operator Output Hardware

The output hardware for the operator to see consists of an LCD screen mounted on the local user controller. Pins on the LCD are as follows (also available in the LCD conceptual design section):

The output hardware also consists of the serial port of both the PIC and the host computer, through the use of a RS-232 standard serial connection.

Hardware Quality assurance

Describe any measures that were taken to control (improve) hardware quality and reliability - Heartbeats, brownout conditioning/resets, reset conditions, testing and validation, etc.

4.2.3 Hardware Validation

It was important to test all hardware separately first before integration as this showed all components working correctly and gave a baseline for expected results so component issues could be isolated and debugged once integrated.
In order to confirm that the hardware was working as intended we used the datasheet information as detailed in the sections above to compare values we were receiving from each hardware with the expected results outlined in the datasheets. For example, the infrared sensor was validated by measuring 50cm with an accurate measuring tape and checking that the sensor reading was correct. If the sensor reported inaccurate readings, then a calibration offset was added to make future readings accurate.
More information including the results of working components can be found in the testing documentation within the appendix.

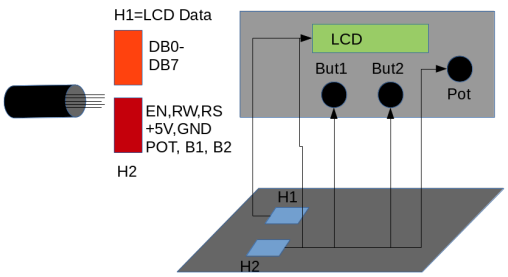

Figure 4.5: Local Ui data+power flow, implemented on veroboard

4.2.4 Hardware Validation

Details of any systematic testing to ensure that the hardware actually functions as intended.

4.2.5 Hardware Calibration Procedures

Procedures for calibration required in the factory, or in the field

4.2.6 Hardware Maintenance and Adjustment

Routine adjustment and maintenance procedures

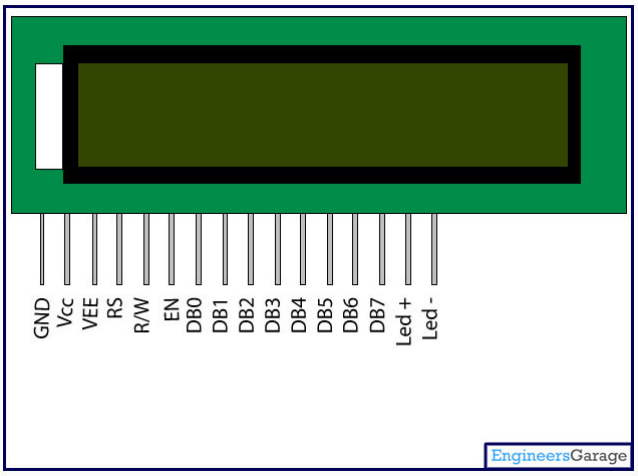

Figure 4.6: The two LCD pins labelled LCD- an LCD+ are backlight pins, and were not implemented.

Chapter 5

Software Design

The software requirements and overview have been dealt with elsewhere in this section addresses the design and implementation of the software that forms the ¡X system¿.

5.1 Software Design Process

The software was designed in a top down manner, around a basic state machine shown in Fig. 9.1. A full detailed description of the system states is included in the state descriptions documents. Once the states and transitions were decided on a basic framework was written that stored the state as an enumeration and a switch case within an infinite loop that continuously calls functions based on the state.
while the system was designed in a top down manner many of the functions were designed bottom up, primarily the hardware interface functions which were designed by starting with the datasheet, and working upwards. This was however restricted to the individual functions and met the top down design at the function design level.
The entire system was designed to be simple, and fit together nicely. As such we had very few interface problems, and almost all of these were hardware related, due to things like common power supplies etc.

5.1.1 Software Development Environment

The software was developed in the MPLAB X IDE v2.15 using the v3.47 C18 compiler. Much of the software was written and tested using the simulator included in the MPLAB environment, which allowed functionality to be tested without the need for actual hardware, which allows more flexibility and better debugging resources. The hardware interface however needed to be tested on either the minimal board or the PICDEM. Where possible, all code was designed for, written and tested on the minimal board so there would be no issues in

porting it. Due to the parallel nature in which the code was written however some modules, the LCD in particular were written on the PICDEM which caused some issues when it came to integration.
For ease we defined the minimal board in the code, which whether defined or undefined would switch between the hardware. This was mainly the included headers and the clock frequencies. This was never actually tested as the main code was only ever run on the minimal board, and there may be some differences in library functions etc.

5.1.2 Software Implementation Stages and Test Plans

It should be noted that the following stages was often an iterative process, especially with the module stages, where each of the modules went through the described stages independently as they were finished.

State Design

The first stage of the software implementation was to design a state machine for the system. This was done by considering the problem at hand, and creating some preliminary idea of how we were going to solve it with the resources at hand.
The preliminary design is described in great detail in the State Descriptions document. This design did however change as we were implementing the software, particularly the tracking component, so the final system is as shown in Fig. 9.1.

State Implementation

Once all the states were decided on, a basic state machine framework was written which consisted only of an infinite loop in the main function going through a switch case and calling a state function depending on the current state which was stored as an enumeration. The state variable was also implemented as a struct containing the current and previous states so the system would know if it was entering a state for the first time, and could perform different functionality. In the final design this was not necessary.
At this stage, all the state functions were implemented merely as stub functions that could be filled with actual functionality later.

Module Design

Once the initial framework was in place, the functionality of the system was broken into modules that could be coded and tested in complete isolation. Some modules, such as the tracking module make use of other modules, as shown in Fig. 2.1, but otherwise the modules are completely separate, with no shared or global variables and were often coded in parallel.
A full description of the module breakdown is given in the Module Descriptions document, which details how the functionality is split into different modules,

the public interfaces between modules and how they would communicate with the rest of the program.
Once the modules had been designed, skeleton code for the vast majority of modules (some additional modules were added, or changed) was written, outlining the basic framework of the module, and the public interface, as detailed in the module descriptions document. This skeleton code was supposed to reduce the daunting nature of trying to write an entire module for members with weaker programming backgrounds, as well as speed up the process for more experienced members, but primarily to ensure that everyone stuck to the decided upon design so that everything would work when we integrated it.

Module Implementation

This step, as expected took up the bulk of the time. Group members were allocated, or picked modules to work on, and there was much collaborating between group members to get modules working. The initial code was not difficult to write, because the system design had split everything into such workable segments the complete picture of each module and function was easy to visualise. Most of the difficulty was trying to get the hardware, and processor resources on the PIC to function correctly. This took the form of writing code primarily using the in-built library functions, finding it not working, and then trying to debug it and try to find why it was not working. There were also some other challenges associated with the compiler, such as the C18 compiler not supporting integer promotion, which means when variables are used in an operation with large intermediate values they often simply overflow and you end up with very strange undesirable results such as 17*30=8 without even a compiler warning.

Module Testing

There was a testing document drawn up at the beginning of the project which contained every function to be written, its current status, if it had been tested, verified, when and by whom. It also contained the working code so there would be no chance of making some changes, breaking it and not knowing what happened. However as the project took off, it was hard to police the testing, especially toward the end, with everyone just writing their own code and saying that it works without providing any documentation or evidence. Despite this much of the system (bar perhaps some of the final additions) were tested, and the testing document facilitated the detailing of a testing procedure by the author of the function, even if he did not perform the actual testing.
On this project we did not implement any kind of automated testing such as would be desirable in an industrial environment, but rigorous testing procedures were outlined, documented and implemented whenever possible.

Module Integration

As the modules were finished they were able to be placed into the state functions created in the state Implementation stage. Some of the state functions were also

completely replaced by some of the module functions as there was little point having an entire function which just called another function. The interfaces to the modules had already been decided upon well in advance, and details on how to use each module existed, which made this stage surprisingly simple.

System Testing

At the end of the project there was very little time for rigorous system testing, however due to the way the system was designed to facilitate the integration of the modules there were very little issues. Our final system testing primarily consisted of simply playing with the system and making sure there were no issues.

Dependencies

Personally I found there to be few dependencies throughout the project; while it was beneficial to have the serial operational when we were working on the range finding (to display the output), much of the debugging was spent stepping though code and the output was easily seen that way, really almost all the modules and things could be tested merely with dummy inputs to simulate what the rest of the program would output, and the entire functionality of a module could be tested in isolation of the other modules. The only real dependency was the tracking algorithm, which required both the range and the Pan Tilt modules. Even this could probably be tested without the other modules functioning with some complex wrapper function to supply inputs in order to illicit and test a particular response, but this would be unnecessary and much more effort than simply changing the order of the functions. However, it remains that the vast majority of the functionality could be written, tested and debugged in complete isolation of everything else.

Pseudocode (PDL)

It is my opinion that pseudocode should contain essentially the function declarations and the comment blocks of the functions, describing in an algorithmic manner the way that the function should operate without going into language specifics. For this reason we thought the skeleton code, and comment blocks that were written with the skeleton code, in addition to the module descriptions document adequate in lieu of dedicated pseudocode. If the solution was more complex algorithmically pseudocode would definitely have been warranted, but as it was, most of the code was simply interfacing with hardware, and the only modules that could really warrant pseudocode at all would be the tracking and menusystem modules. We had very few algorithmic related issues, but again, were the solution more complex we would have made use of pseudocode.

5.2 Software Quality Assurance

Describe any measures that were taken to control (improve) the software quality - code or documentation standards, code walkthroughs, testing and validation, etc.

The code was designed and written to be as modular, and easy to implement and integrate as possible. This necessarily improves the quality as any errors, or unhanded exceptions become much clearer as the solution becomes simpler. In other words the very way we went about writing and integrating the software improved and helped ensure the software was of a high standard.

In addition we also drew up documents, such as the testing document into which everyone contributes. This document was designed as a running document which details which functions and functionality is working, what has been tested, by what vigour and methodology (in case someone finds a hole in the testing methodology), and when and by whom it was tested and validated. This document should at a glance communicate what has been completed and tested, what may be working and what is yet to be done. The core concept was that nothing went into the testing document unless it had been rigorously and assuredly been tested and verified, even if it were a trivial macro or something in one line. This ensured that everything in the testing document was absolutely working, and never had to be examined again, ensuring that once a function has been completed it is verified and never has to be touched again. This also allows a mechanism for the author of a function to write out a testing method for their function, even if they don't do the actual testing.

Another method we used to ensure quality was to begin by creating a working skeleton code framework of the entire code before starting to write any of the functionality. The entire time we were testing and writing code, even for individual systems, the rest of the code was simply not called. This ensured that the code as as whole built to start with, and was written incrementally, so it never stopped building. Thus our integration was simply putting all the function calls in, and working through run time errors. We had no real compiling or building errors due to integration at the end.

There were also other documents such as the state descriptions and module descriptions which gives a detailed holistic view of the entire system and how it all fits and works together.

5.3 Software Design Description

5.3.1 Architecture

System Overview

The system at its highest level is as basic state machine. Fig. 9.1 shows the states and transitions in the final version. The attached State descriptions document describes in great detail all the states and transitions as they were designed at the beginning of the project. Some of these states have changed

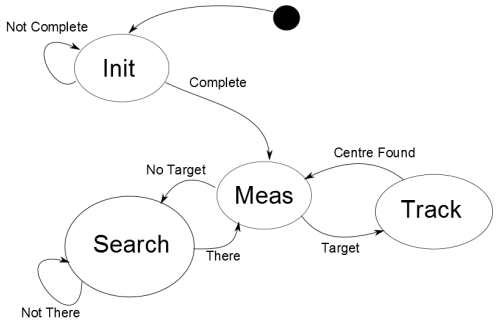

Figure 5.1: System overview state diagram

since the document was written, but much of the basic concept is the same.

Much of the code is functionality around hardware interfaces, most of which is self explanatory, or accessible through the datasheet. Here we will discuss some of the higher level software modules such as the search, tracking and menus.

Tracking

The tracking algorithm is really the heart of the specification; to design and build a death star tracker. Despite this there is alot of functionality that must be in place before the module is operational. Here only the high level tracking algorithm is discussed.
Fig. 5.2 shows the original concept for tracking the moving targets. The basic concept was that the system would start somewhere on the target (after having detected it) and then move outward in each direction (up, down, left and right) until it found the edge, and would return to the centre, and begin again. The problem we found with the original design was that it was not detecting the edges well, or if it hit the maximum azimuth before it got to the edge then it would simply keep going. We also found that if the target was moving in the opposite direction to the edge it was trying to track it would easily loose the target.
Working from this design I implemented a new concept where discrete, fixed points are sampled and the results are used to create a weighted average of the positions, giving the next target location. This is shown in Fig. 5.3.
These algorithms are also incremental in nature; they perform a single weighted average and return execution back to the main program, telling it which state to go to next. This may seem less efficient than staying in the tracking algorithm while ever it has a track, but when we are waiting for such periods for the servos

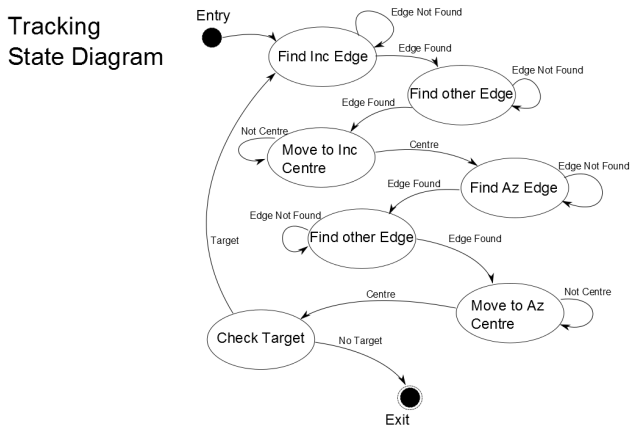

Figure 5.2: State Transition Diagram depicting the Original Idea for the tracking algorithm

to move, and the ultrasonic to return there is no comparative loss of efficiency. It also means that the program is able to do other things such as service the menus while it is searching and tracking, and it lessens the chance of getting stuck in sub-functions. This design also facilitates the implementation of a watch dog timer.

Searching

Fig. 5.4 depicts the operation of the searching algorithm. The concept is very simple: the system scans the azimuth until it reaches the maximum and minimum azimuth (user defined), and then increments the elevation. The increments for the azimuth and elevation are stored as variables, so their magnitude can be changed at any time, and it means that they are simply negated when the system reaches a maximum value, again as shown in Fig. 5.4. This algorithm is incremental, so it does a single pan tilt movement in the correct direction, checks the target and returns the next system state. If the system detects a target this next system state will be the tracking state, otherwise it will remain in the searching state. In this way the searching function acts as the 'state functions' described in the system design process.
Among the benefits of this system is that execution is continuously returned to the primary loop, which lowers the chance of nested infinite loops and getting stuck in sub-functions, but it also means that we are able to implement a watch dog timer simply by clearing it every loop iteration. Thus if for whatever reason the execution does not complete a loop in a few seconds or so, the entire system resets.

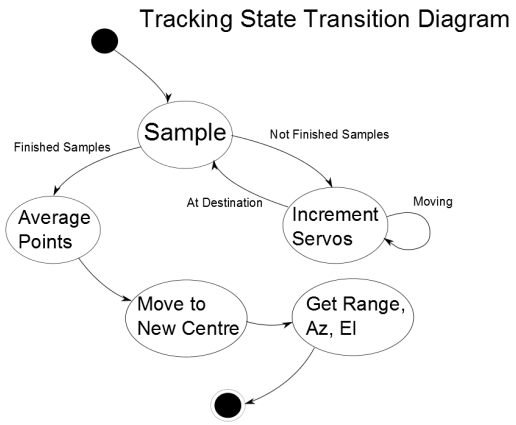

Figure 5.3: State Transition Diagram depicting the final tracking algorithm

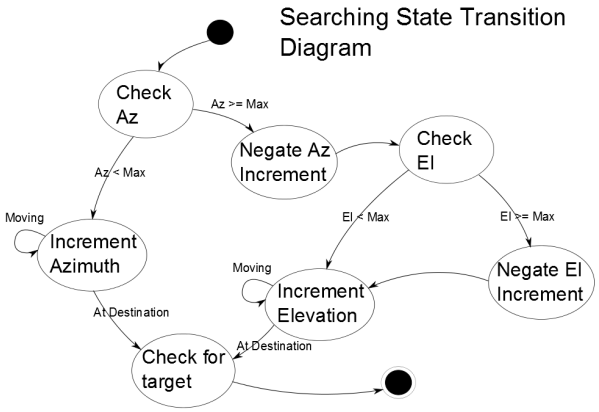

Figure 5.4: State Transition Diagram of the searching algorithm

5.3.2 Software Interface

The software interfaces for each module are described in great detail in the module descriptions document.

5.3.3 Software Components

This is a detailed view of the internal workings of each of the software modules

5.4 Preconditions for Software

5.4.1 Preconditions for System Startup

Describe any preconditions that must be satisfied before the system can be started.

Before the system can be started it must be connected to a power source with a sufficient power rating. Then it must be fully assembled. Apart from that there are no preconditions for the system to begin startup. The code is designed to configure all the modules and hardware that it needs automatically on startup. There is however some background bootloader which loads all the constants in from program memory and initialises the variables defined in the idata sections etc. Again this happens automatically at startup and requires no other preconditions. The Pan Tilt module is then moved to the starting point of (0,0).

5.4.2 Preconditions for System Shutdown

There are no preconditions for a system shutdown, and the system can be shutdown at any point with no warning. There is however an option to store the user changes for the configuration settings in the eeprom, and should the user wish to retain their settings this should be completed before shutting down the system. Also if the user were to shut down the system during the eeprom write the action may corrupt their stored settings. Other than this, the system can safely be shut down at any point with no consequences.

Chapter 6

System Performance

6.1 Performance Testing

Give the results of testing conducted to determine the characteristics and performance of the system - memory usage, loop time, system accuracy, repeatabability, ease of use, etc.

The testing document goes through the testing procedures and results of each function as they were completed.

6.2 State of the System as Delivered

Over the period of development, a majority of the system modules met their required functional and non-functional requirements. The PanTilt and Range modules performed as designed once calibrated, which allowed the tracking module to accurately track the model Death Star from ranges between 30 centimetres to 2.5 metres. The tracking algorithm performed at a rate exceeding the requirement of the target travelling at 10 centimetres per second.

The system hardware and user interface was constructed and put together with a custom faceplate to improve the aesthetic design of the device, which made the device appear much more presentable and professional. The user interface over both serial and the LCD were refined and easy to understand, and was praised for it's effectiveness.

At the end of development, a watchdog timer was implemented but not adequately tested, which caused a major serial communications issue to occur during the product demonstration from a last-minute change in values. The watchdog timer successfully reset the program when it crashed, however it would also cause the program to crash whenever a user had the ability to change system settings. This was a major issue, as the code to change system variables was correct and working, the demonstrator was not able to do so due to the program halting. This was the largest issue with the presentation of the system, as multiple requirements could not be demonstrated. These changes to the

watchdog timer should not have been included with the demonstration version of the program as they were not thoroughly tested, and a more stable build would be presented in the future.

Although the hardware for the local interface and software for the interface was most likely correct, an adequate amount of integration testing had not been completed, so these systems were disconnected upon system demonstration as to not affect each other. From this, we were not able to demonstrate switching between remote and local user modes, but could show the ability to switch to factory mode using the key switch, which worked as expected.

6.3 Future Improvements

The system could be improved in many ways if the project had a longer deadline or if future work is allowed. The following is a list of prioritised work that could be completed based on the current design of the system.

- **Improve and fix the watchdog timer:** The watchdog timer change to the code could be fixed so that it does not interfere with the serial communications as it did in the demonstration. This is the most important fix which would need to be done, as most of the serial user interface did not function due to the watchdog timer, and having the ability to provide to both the remote user and factory user is of the utmost importance.
- **Integrate Local Interface:** Some further testing would need to be performed so that the local interface software could be integrated correctly. This is also a large requirement of the system, and is prioritised as such.
- **Add additional factory menus:** Some factory settings were developed as part of the appropriate module but did not have a user menu dedicated to them, such as the ability to calibrate the servo motors. This would satisfy more of the requirements quickly and easily fits into the current menu system.
- **Fix serial communications issues:** On occasion, the serial interface would display an incorrect character or shorten messages. The user input would also occasionally be received or parsed incorrectly. These issues were investigated over the course of development, but more time could be invested so that the user feels as if the product has a high level of quality.
- **Integrate the EEPROM module:** An EEPROM module was written and tested, but was not integrated into the system due to time constraints. Adding this into the system would allow the user to save system variables regardless of if the device was powered down. This is a quality of life improvement for the user, and is hence not a top priority.

Chapter 7

Safety Implications

This section outlines identified hazards associated with operating this system, their likelihood, consequences and overall severity. It then details measures taken to reduce the severity of all hazards to an acceptably low level.

7.1 Hazard Risk Table

Table 7.1 shows the risk associated with different hazards based on the likelihood and severity of the injury. In general practise a low risk is considered acceptable, whereas anything higher should be addressed with safety measures to reduce both the likelihood and severity of the possible injury.

Likelihood	Consequence				
	Severe	Major	Moderate	Minor	Insignificant
Almost Certain	Very High	Very High	High	Medium	Medium
Likely	Very High	High	High	Medium	Medium
Possible	Very High	High	High	Medium	Low
Unlikely	High	Medium	Medium	Low	Low
Rare	Medium	Medium	Medium	Low	Low

7.2 Identified Hazards

Below is a list of hazards identified for the product, their associated risk, and measures taken to reduce the risk.

7.2.1 Cut Hazard

Description

There is a hazard that someone could cut themselves on one of the edges of the product, or poke their eye on one of the corners etc.

Risk

This hazard has a Possible to likely likelihood, and a minor consequence. This gives the hazard a risk factor of Medium.

Hazard Reduction Techniques

To reduce the likelihood of this hazard we recommend that it is kept away from small children, and used professionally, which could reduce the likelihood of this type of injury to Possible. We also smooth all the sharp edges and corners to reduce the consequence to insignificant. This reduces the severity of the hazard to an acceptably low standard.

7.2.2 Drop hazard

Description

There is a hazard associated with people dropping the product, if it lands on them, or someone else, or breaks and produces dangerous shards etc.

Risk

This hazard has a Possible to unlikely likelihood, and a minor consequence (as the system is not particularly heavy). Thus it carries a Medium to low severity.

Hazard Reduction Techniques

To reduce the likelihood of this hazard we have ensured that the product does not contain any materials that may shatter or splinter during a typical drop scenario. We can also recommend that the product is installed (fixed down) or handled carefully to reduce the likelihood to unlikely. We have also ensured that the product is reasonably light such as to not injure someone by dropping it on someone, so it carries a minor to insignificant consequence. Thus the hazard risk is reduced to an acceptably low level.

7.2.3 Shock hazard

Description

This hazard is separated from the electrocution hazard as it is a much more likely, but more insignificant effect. Such a shock hazard could include touching the ultrasonic sensor during operation, and getting a shock, while the electricity does really pass through you.

Risk

This hazard carries a likely to almost certain risk if being used by children or curious casual users. This hazard carries no real risk of any kind of injury, but

typically more of an unpleasant shock unless amplified by some kind of conductor. Thus the hazard carries an insignificant consequence, giving a severity of medium.

Hazard Reduction

To reduce the likelihood of this hazard we can place warning labels on the ultrasonic sensor, and box, again recommend it be kept away from small children, and that aeronautical engineers should operate it under the supervision of an adult. This could reduce the likelihood of the hazard to possible to unlikely, and reducing the severity to low.

7.2.4 Electrocution hazard

Description

There is always an electrocution risk associated with any electrical product, especially one drawing high amperage. This product however is typically battery powered and thus there is no active terminal that will flow to ground.

Risk

This hazard has a likelihood of rare, and a consequence of moderate to major. This gives a severity of medium.

Hazard Reduction

To reduce the hazard we have completely concealed the electrical components and wires in an insulating container that is not designed to be opened by a typical user. We have also used insulating wires and electrical tape to conceal connections. The product is typically battery operated, which significantly reduces the risk, and while it does not have any inbuilt fuses, circuit breakers or grounds, should it be powered from an external source (by an experienced user) it is assumed that such a source have its own safety features. Thus the consequence can be reduced to minor (possibly moderate if using an external power supply) which reduces the severity to low (possibly low to moderate with an external power supply).

7.2.5 Collision Hazard

Description

As the system has moving parts there is a risk that it will collide with something or someone during operation.

Risk

This hazard carries a possible to likely likelihood, as the system has no sensors or safeguards to prevent such collisions. As the system is using low torque (0.1N.m) servo's, and the system itself is quite light (e.g. would probably just turn the box) this has an insignificant consequence. This gives a low to medium severity.

Hazard Reduction

To reduce hazard we can put moving parts warning labels on the pan tilt, and recommend that users maintain some kind of distance from it. This could reduce the likelihood to possible to unlikely, giving an overall severity of low.

Chapter 8

Conclusions

In conclusion, the Yavin IV Defence System has reached it's purpose of being able to track a model of the Galactic Empire's Death Star using ultrasonic and infra-red range-finders mounter using servo motors and running software developed for a PIC18F4520 microcontroller. Whilst much of the functionality was achieved, time remaining became a large concern, the system could be improved with more time to improve the product, specifically in regards to integrating the user menu with other components.

Chapter 9

Appendix

9.1 Code

9.1.1 CircularBuffers.h

```
/*! ***************************************************************************
 * File:    CircularBuffers.h
 * Author: Grant
 *
 * Description:
 * Contains Circular Buffer functionality and structure definitions. This allows
 * Circular Buffers to be used in any file by simply including this header.
 * Furthermore, circular buffer functionality can be altered on a program level.
 *
 * Contains:
 *      -Definition of CircularBuffer type
 *      -Peek, pop, push and init functionality
 *      -Empty, full, queries
 *
 * Created on 8 October 2014, 10:30 AM
 ****************************************************************************/

//Ensure there is only 1 inclusion of this file in the preprocessor execution
#ifndef BUFFERS_H

///Buffer related macro functionality
#define incMod(ptr) if(ptr==BUFFERLENGTH-1) ptr = 0; else ptr++;
#define empty(buf) (buf.tail == buf.head)
#define full(buf) (buf.head == (buf.tail + 1) % BUFFERLENGTH)
#define peek(buf) buf.data[buf.head]

#define push(byte, buf) buf.data[buf.tail] = byte; if(full(buf)) incMod(buf.head); incMod(buf.
#define pop(buf) buf.data[buf.head]; if (!empty(buf)) incMod(buf.head)
#define init(buf) buf.tail = 0; buf.head = 0

//NOTE: The BUFFERLENGTH can be redefined at the top of any module if a different length is de
#define BUFFERLENGTH 80

```

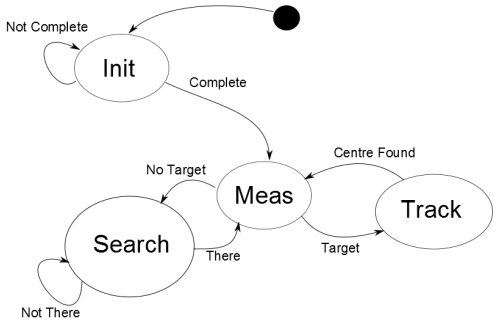

Figure 9.1: System overview state diagram

```
/*! ***************************************************************************
 * typedef of circularBuffer struct
 *
 * @brief Stores bytes of data in a circular buffer
 *
 * Length: BUFFERLENGTH
 *
 * Description:
 * Defines a circular buffer in which bytes of data can be placed, or removed at
 * any time in the correct order that they were pushed. The accompanying functionality
 * facilitates all necessary circular buffer related operations.
 * This implementation reduces the buffer length by 1 to differentiate between
 * full and empty states.
 * BUFFERLENGTH can be redefined in individual files to get different buffer lengths
 ****************************************************************************/
typedef struct
{
    unsigned char head;
    unsigned char tail;
    unsigned char data[BUFFERLENGTH];
} circularBuffer;

#define BUFFERS_H 0
#endif
```

../Code/CircularBuffers.h

9.1.2 Common.h

```
/*! ***************************************************************************
 * File:   Common.h
 * Author: Grant
 *
 * Description:
 * Contains all program scope definitions, declarations and inclusions. This header
```

```
 * should be included in all source files by default.
 *
 * Contains:
 *      -PIC18F family library headers
 *      -Direction struct typedef
 *      -TrackingData struct typedef
 *      -systemState struct typedef
 *      -system state macro functionality
 *      -TargetState enumeration
 *      -Interrupt flag macros
 *      -Division macros
 *      -Clock frequency definitions
 *      -SWAP macro functionality
 *
 * Created on 11 September 2014, 12:24 PM
 ******************************************************************************/

//Ensure that there is only 1 inclusion of this file in the preprocessor execution
#ifndef COMMON_H

// Use the MNML Board. Comment this out for the PICDEM
#define MNML

//Include files
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#ifdef MNML
#include <p18f4520.h>
#else
#include <p18f452.h>
#endif

//Peripherial headers
#include <timers.h>
#include <adc.h>
#include <capture.h>
#include <usart.h>
#include <capture.h>
#include <compare.h>

#ifndef MNML
//System configurations
#pragma config WDR = OFF
#pragma config OSC = HS
#pragma config LVP = OFF
#pragma config DEBUG = ON
#endif

/*! ***************************************************************************
 * typedef of Direction struct
 *
 * @brief Stores an inclination and azimuth
 *
 * Description:
```

```
 * A fully defined direction in which to point the pan tilt actuator
 * or any other such purpose
 *
 * Elements:
 *      -Azimuth: Contains the azimuth component of the direction (generally degrees)
 *      -Inclination: Contains the inclination component of the direction (generally degrees)
 ******************************************************************************/
typedef struct
{
    int azimuth;
    int elevation;
} Direction;

//Tracking Data definition: Stores the current target information
/*! ***************************************************************************
 * typedef of TrackingData struct
 *
 * @brief Stores the current target information
 *
 * Description:
 * Stores distance, azimuth and inclination tracking data to the target.
 * This struct is used by the tracking function and others to communicate
 * the position of the target
 ******************************************************************************/
typedef struct
{
    unsigned int range;      //Distance to the target
    int azimuth;               //Azimuth to the target
    int elevation;            //inclination to the target
} TrackingData;

///The different types of tracking possible depending on which sensors observe the target
//          !(IR||US)  (US&&!IR) (US&&!IR), (US&&IR)    (IR&&!US)
typedef enum{NO_TARGET, OUT_OF_IR, BAD_DIR, GOOD_TRACK, CLOSE_RANGE} TargetState;

///The different PanTilt settings that can be changed
typedef enum{MAX_AZ, MIN_AZ, MAX_EL, MIN_EL} PanTiltSettings;

///The different Range settings that can be changed
typedef enum{MAX_RANGE, MIN_RANGE, IR_RATE, US_RATE, IR_SAMPLES, US_SAMPLES} RangeSettings;

///Available ANSI escape Sequences
typedef enum{CLR_SCN, ERS_LN} escSqnce;

/// Macros to change the state of the system
#define NEXT_STATE(s, state) state.previous = state.current; state.current = s
#define NEXT_STATE_PTR(s, state) state->previous = state->current; state->current = s

///Define the possible states the system can be in
typedef enum{UNDEF, INIT, SRCH, TRCK, MENU, RAW_RANGE_STATE} possible_states;

/*! ***************************************************************************
 * typedef of systemState struct
 *
 * @brief Stores the current state of the system
 *
 * Description:
```

```
 * Stores which state the system is currently in, as defined by the
 * possible_states enumeration. Also stores the previous system state
 * so the system knows if this is the first iteration of the state
 ******************************************************************************/
typedef struct {
    possible_states current;
    possible_states previous;
} systemState;

///Efficient Division macros based on bit shifting, Cannot be used on negative numbers
#define DIV_2(v) ((v) >> 1)       //Divide by 2
#define DIV_4(v) ((v) >> 2)       //Divide by 4
#define DIV_8(v) ((v) >> 3)       //Divide by 8
#define DIV_16(v) ((v) >> 4)      //Divide by 16
#define DIV_32(v) ((v) >> 5)      //Divide by 32
#define DIV_64(v) ((v) >> 6)      //Divide by 64
#define DIV_128(v) ((v) >> 7)     //Divide by 128
#define DIV_256(v) ((v) >> 8)     //Divide by 256
#define DIV_512(v) ((v) >> 9)     //Divide by 512
#define DIV_1024(v) ((v) >> 10)   //Divide by 1024
#define DIV_4096(v) ((v) >> 12)   //Divide by 4096
#define DIV_65536(v) ((v) >> 16)    //Divide by 65536

///Swaps the values in two variables
#define SWAP(x, y) (y = (y ^ (x = (x ^ (y = (x ^ y))))))

///ADC Channel Select macros
#define ADC_IR_READ 0x01            //Sets ADC to read the IR
#define ADC_TEMP_READ 0x05          //Sets the ADC to read the temp
#define ADC_DIAL_READ 0x09          //Sets the ADC to read the DIAL

///Interrupt macros
#define TX_INT (PIR1bits.TXIF && PIE1bits.TXIE)       //serial transmit interrupt fired
#define RC_INT (PIR1bits.RCIF && PIE1bits.RCIE)       //serial receive interrupt fired
#define CCP1_INT (PIR1bits.CCP1IF && PIE1bits.CCP1IE) //Whether CCP1 fired the interrupt
#define CCP2_INT (PIR2bits.CCP2IF && PIE2bits.CCP2IE) //Whether CCP2 fired the interrupt
#define INT0_INT (INTCONbits.INT0IF && INTCONbits.INT0IE) //Whether the external interrupt fire
#define INT1_INT (INTCON3bits.INT1IF && INTCON3bits.INT1IE) //Whether the external interrupt f
#define INT2_INT (INTCON3bits.INT2IF && INTCON3bits.INT2IE) //Whether the external interrupt f
#define RB_INT   (INTCONbits.RBIF && INTCONbits.RBIE)  //Whether the RB port change interrupt
#define TMR2_INT (PIR1bits.TMR2IF && PIE1bits.TMR2IE)   //TMR2 matching PR2 fired the interrup

#define TMR0_INT (INTCONbits.TMR0IF && INTCONbits.TMR0IE) //Whether the timer0 overflow fired
#define TMR1_INT (PIR1bits.TMR1IF && PIE1bits.TMR1IE)   //TMR1 overflow fired interrupt
#define TMR3_INT (PIR2bits.TMR3IF && PIE2bits.TMR3IE)   //TMR3 overflow fired interrupt
#define ADC_INT (PIR1bits.ADIF && PIE1bits.ADIE)        //Whether the ADC fired the interrupt
#define SSP_INT (PIR1bits.SSPIF && PIE1bits.SSPIE)      //Whether the SSP module fired the int
#define BCL_INT (PIR2bits.BCLIF && PIE2bits.BCLIE)      //Bus collision interrupt fired
#define LVD_INT (PIR2bits.LVDIF && PIE2bits.LVDIE)      //Low voltage detect interrupt fired

#define CCP1_CLEAR (PIR1bits.CCP1IF = 0)
#define CCP2_CLEAR (PIR2bits.CCP2IF = 0)

///Dfine the clock rate and FOSC_4
#ifdef MNML
#define CLOCK   10000000    //10MHz clock source
#define FOSC_4  2500000     //2.5MHz Fosc_4
```

```
#else
#define CLOCK 4000000    //2.5MHz clock source
#define FOSC_4 1000000    //1MHz Fosc_4
#endif

#define INT_SETUP() INTCONbits.GIEH = 1; \
INTCONbits.GIEL = 1; \
PIR1 = 0; \
PIR2 = 0; \
RCONbits.IPEN = 1; \
IPR1 = 0; \
IPR2 = 0; \
IPR1bits.TXIP = 1; \
IPR2bits.CCP2IP = 1;\
IPR1bits.CCP1IP = 1;

/*
 * Pins
 */

#define COMMON_H
#endif
```

../Code/Common.h

9.1.3 EEPROM.c

```
//
//unsigned char ReadEEPROM(unsigned char address){
//     EECON1=0;                    //ensure CFGS=0 and EEPGD=0
//     EEADR = address;
//     EECON1bits.RD = 1;
//     return(EEDATA);
//}
//
//void WriteEEPROM(unsigned char address,unsigned char data){
//char SaveInt;
//     SaveInt=INTCON;              //save interrupt status
//     EECON1=0;                    //ensure CFGS=0 and EEPGD=0
//     EECON1bits.WREN = 1;         //enable write to EEPROM
//     EEADR = address;             //setup Address
//     EEDATA = data;               //and data
//     INTCONbits.GIE=0;            //No interrupts
//     EECON2 = 0x55;               //required sequence #1
//     EECON2 = 0xaa;               //#2
//     EECON1bits.WR = 1;           //#3 = actual write
//     INTCON=SaveInt;              //restore interrupts
//     while(!PIR2bits.EEIF);       //wait until finished
//     EECON1bits.WREN = 0;         //disable write to EEPROM
//}


//microchip code, not ready yet

//* The Processor starts with the External Crystal (8 Mhz).
//*****************************************************************/
```

```
//#define USE_OR_MASKS
//#include <p18cxxx.h>
//#include "EEP.h"
//
//unsigned char  EEPWrite[15] = "MICROCHIP_TECH", EEPRead[15],Error=0 ;
//
//void main(void)
//{
//    unsigned char q=0;
//  unsigned int  address;
//
//    address = 0x0200;
//
//      /* Write single byte to Internal EEP*/
//    for(q=0;q<16;q++)
//    {
//      Write_b_eep (address, EEPWrite[q]); // write into to EEPROM
//    address++;                //increment the address of EEPROM to next location
//      /* Checks & waits the status of ER bit in EECON1 register */
//      Busy_eep ();
//    }
//
//    address = 0x0200;   // initialize the starting address
//    Error = 0;       //clear the error flag
//      /* Read single byte from Internal EEP*/
//    for(q=0;q<16;q++)
//    {
//      EEPRead[q] = Read_b_eep (address++);    //read the EEPROM data written previously from
//    if ( EEPRead[q] != EEPWrite[q] )      //check if the data read abck is same as that was
//    {
//    Error=1;                //if the data read/ write match does not occur, then flag the er
//    while(1);               //error occured
//    }
//    }
//
//    while(1);           //End of program
// }
//
//
//
```

../Code/EEPROM.c

9.1.4 Interrupts.c

```
/*!*****************************************************************
 * File:   Interrupts.c
 * Author: Grant
 *
 * Description:
 * Contains the interrupt service routines for the system.
 *
 * Duties:
 *      -Handle all interrupts called by the system
 *      -Call relevant module service routines
 *
 * Created on 8 October 2014, 11:27 AM
```

```
 ********************************************************************/

#include "Common.h"

/** ************************************************************
 Any module that uses interrupts will need to be included here:
 ************************************************************/
#include "Tracking.h"
#include "Range.h"
#include "User_Interface.h"
#include "Serial.h"
#include "PanTilt.h"
#include "Temp.h"
#include "Menusystem.h"
/** ************************************************************
 * Add your module header file here...
 ************************************************************/

//ISR prototypes
void lowISR(void);
void highISR(void);

/*! ***************************************************************************
 * Function: highVector(void)
 *
 * Include: Local to Interrupts.c
 *
 * Description: Sends program control to the high priority ISR
 *
 * Arguments: None
 *
 * Returns: None
 *
 * Remarks: This is an interrupt vector, placing a goto in the
 *          high priority interrupt table to call the high priority ISR
 ****************************************************************************/
#pragma code highPriorityInterruptAddress=0x0008
void highVector(void)
{
    _asm GOTO highISR _endasm
}

/*! ***************************************************************************
 * Function: lowVector(void)
 *
 * Include: Local to Interrupts.c
 *
 * Description: Sends program control to the low priority ISR
 *
 * Arguments: None
 *
 * Returns: None
 *
 * Remarks: This is an interrupt vector, placing a goto in the
 *          low priority interrupt table to call the low priority ISR
 ****************************************************************************/
#pragma code lowPriorityInterruptAddress=0x0018
```

```
70 void lowVector(void)
   {
72     _asm GOTO lowISR _endasm
   }
74
   /*! ***************************************************************************
76  * Function: lowISR(void)
    *
78  * Include: Local to Interrupts.c
    *
80  * Description: Interrupt Service Routine to check what condition initiated
    *              a low priority interrupt call, and perform the nessicary action
82  *
    * Arguments: None
84  *
    * Returns: None
86  ****************************************************************************/
   #pragma interruptlow lowISR
88 void lowISR(void)
   {
90     if (SERIAL_INT)
       {
92         serialISR();
       }
94     if (PAN_TILT_ISR)
       {
96         panTiltISR();
       }
98     if (RANGE_INT)
       {
100        rangeISR();
       }
102    if (USER_INT)
       {
104        userISR();
       }
106    //Add additional checks here
   }
108
   /*! ***************************************************************************
110 * Function: highISR(void)
    *
112 * Include: Local to Interrupts.c
    *
114 * Description: Interrupt Service Routine to check what condition initiated
    *              a high priority interrupt call, and perform the nessicary action
116 *
    * Arguments: None
118 *
    * Returns: None
120 ****************************************************************************/
   #pragma interrupt highISR
122 void highISR(void)
   {
124    if (PAN_TILT_ISR)
       {
126        panTiltISR();
```

```
    }
128     if (SERIAL_INT)
    {
130         serialISR();
    }
132     if (RANGE_INT)
    {
134         rangeISR();
    }
136     if (USER_INT)
    {
138         userISR();
    }
140     //Add additional checks here
}
```

../Code/Interrupts.c

9.1.5 jEEP.h

```
1 /*
  * File:   jEEP.h
3 * Author: Jacob
  *
5 * Created on 27 October 2014, 3:24 PM
  */
7
#define EEPADDRESS 0x00
9
extern void sendEep(int number, int address);
11 extern int readEep(int address);
```

../Code/jEEP.h

9.1.6 LCD.h

```
/*! ***************************************************************************
2 * File:   LCD.h
  * Author: Grant
4 *
  * Description:
6 * Contains the public interface for the LCD module.
  *
8 * Created on 17 September 2014, 9:01 PM
  *****************************************************************************/
10
//Ensure that there is only 1 inclusion of this file in the preprocessor execution
12 #ifndef LCD_H

14 /*=============================================================================*/
  //External declarations of public access functions
16 /*=============================================================================*/

18 /*! ***************************************************************************
  * Function: lcdWriteString(char *string, unsigned char line)
20 *
  * \brief
```

```
22  *
    * Include: LCD.h
24  *
    * Description: Writes a string to the LCD at the given line
26  *
    * Arguments: string - The string to write
28  *            line - the line to write to
    *
30  * Returns: None
    ************************************************************************/
32 extern void lcdWriteString(char *string, char line);   //!Feed character string, and line (1 o

34 /*! ***********************************************************************
    * Function: lcdWriteChar(unsigned char byte, unsigned char line, unsigned char column)
36  *
    * \brief
38  *
    * Include: LCD.h
40  *
    * Description: Writes a character to the LCD at a given location
42  *
    * Arguments: byte - byte to write
44  *            line - line to wrtie to
    *            column - column to write in
46  *
    * Returns: None
48  ************************************************************************/
   extern void lcdWriteChar(char byte, char line, char column);    //!Feed character, line (1 or
50
   /*! ***********************************************************************
52  * Function: confiigLCD(void)
    *
54  * \brief Initialises the LCD so that it can be used
    *
56  * Include: LCD.h
    *
58  * Description: Initialises the LCD hardware so that data can be displayed
    *              on the LCD in local interface mode
60  *
    * Arguments: None
62  *
    * Returns: None
64  ************************************************************************/
   extern void configLCD(void);                //initialize
66
   #define LCD_H
68 #endif
```

../Code/LCD.h

9.1.7 LCD_defs.h

```
   /*
2   * File:   LCD_defs.h
    * Author: Jacob
4   *
    * Created on 22 October 2014, 12:01 AM
```

```
 */

#define LCD_D0 PORTDbits.RD0        //!LCD data bits
#define LCD_D1 PORTDbits.RD1
#define LCD_D2 PORTDbits.RD2
#define LCD_D3 PORTDbits.RD3
#define LCD_D4 PORTDbits.RD4
#define LCD_D5 PORTDbits.RD5
#define LCD_D6 PORTDbits.RD6
#define LCD_D7 PORTDbits.RD7

#define LCD_D0_DIR TRISDbits.RD0    //!LCD direction for data bits
#define LCD_D1_DIR TRISDbits.RD1
#define LCD_D2_DIR TRISDbits.RD2
#define LCD_D3_DIR TRISDbits.RD3
#define LCD_D4_DIR TRISDbits.RD4
#define LCD_D5_DIR TRISDbits.RD5
#define LCD_D6_DIR TRISDbits.RD6
#define LCD_D7_DIR TRISDbits.RD7

#define LCD_DATA_LINE       PORTD   //!All data bits
#define LCD_DATA_LINE_DIR   TRISD   //!Direction for data bits

//#define LCD_PWR PORTCbits.RC0       //!LCD Power
#define LCD_RS   PORTCbits.RC4      //!LCD register select line
#define LCD_RW   PORTCbits.RC5      //!LCD read/write line
#define LCD_E    PORTAbits.RA4      //!LCD E clock

//#define LCD_PWR_DIR TRISCbits.RC0
#define LCD_RS_DIR   TRISCbits.RC4
#define LCD_RW_DIR   TRISCbits.RC5
#define LCD_E_DIR    TRISAbits.RA4

#define LCD_INS      0              //! instruction
#define LCD_DATA     1              //! data
#define LCD_WRITE    0              //! write
#define LCD_READ     1              //! read
#define LCD_CLKLOW   0              //! clock low
#define LCD_CLKHGH   1              //! clock high
#define LCD_OFF      0              //! power off
#define LCD_ON       1              //! power on

#define LCD_INPUT        0xFF       //! macro for complete register input
#define LCD_OUTPUT       0x00       //! macro for complete register output
#define LCD_PIN_INPUT    1          //! single pin input
#define LCD_PIN_OUTPUT   0          //! single pin output

#define DISPCLR          0x01
#define RTNHOME          0x02
#define SETENTRYMODE     0x04
#define ID_CURSL         0x00
#define ID_CURSR         0x02
#define SH_DISPNSHIFT    0x00
#define SH_DISPSHIFT     0x01
#define DISPOPT          0x08
#define D_DISPOFF        0x00
#define D_DISPON         0x04
```

```
#define C_CURSOFF         0x00
#define C_CURSON          0x02
#define B_BLINKOFF        0x00
#define B_BLINKON         0x01
#define DISPSHIFTCURS     0x10
#define SC_CURSMOVE       0x00
#define SC_DISPMOVE       0x08
#define RL_LFTSHFT        0x00
#define RL_RGTSHFT        0x04
#define SETDISPFXN        0x20
#define DL_4BIT           0x00
#define DL_8BIT           0x10
#define N_1LINE           0x00
#define N_2LINE           0x08
#define F_5X7DOT          0x00
#define F_5X10DOT         0x04
#define SETCGRAMADD       0x40
#define SETDDRAMADD       0x80

#define BF_READY 0x00
//#define BF_BUSY 0x80
#define BF_BUSY 1

#define LINE1 0x00
#define LINE2 0x40

#define LINESTART 0x00
#define LINEEND 0x0F
```

../Code/LCD_defs.h

9.1.8 LCD.c

```
/*! ****************************************************************
 * File:    User_Interface.c
 * Author:
 *
 * Description:
 * Contains all the interface to the LCD hardware
 *
 * Duties:
 *      -Interfaces with and controls the LCD
 *      -Displays data on the LCD
 *
 * @todo Complete, test, debug and document this code!!!
 *
 * Created on 15 September 2014, 1:21 PM
 ******************************************************************/

#include "Common.h"
#include "LCD_defs.h"
#include <delays.h>

void delay(unsigned int t);
char lcdBusy(void);                 //check busy flag
void lcdWrite(unsigned char byte, unsigned char mode);

```

```
/*! ********************************************************************
 * Function: configLCD(void)
 *
 * \brief Initialises the LCD so that it can be used
 *
 * Include: LCD.h
 *
 * Description: Initialises the LCD hardware so that data can be displayed
 *              on the LCD in local interface mode
 *
 * Arguments: None
 *
 * Returns: None
 ********************************************************************/
void configLCD(void){

    Delay10KTCYx(6);

    LCD_DATA_LINE_DIR = LCD_OUTPUT; //! Set data lines to output

    //LCD_PWR_DIR = LCD_PIN_OUTPUT;     //!Set control lines to output
    LCD_RS_DIR  = LCD_PIN_OUTPUT;
    LCD_RW_DIR  = LCD_PIN_OUTPUT;
    LCD_E_DIR   = LCD_PIN_OUTPUT;

    //LCD_PWR = LCD_ON;                 //!Enable power
    LCD_E   = LCD_CLKLOW;             //!Set control lines low
    LCD_RS = LCD_INS;
    LCD_RW = LCD_WRITE;

    lcdWrite((SETDISPFXN | DL_8BIT | N_2LINE | F_5X7DOT), LCD_INS);   //! 8 bits, 2 lines, 5x7
    delay(2);
    lcdWrite((DISPOPT | D_DISPON | C_CURSOFF | B_BLINKOFF), LCD_INS);  //! Display on, cursor
    lcdWrite((SETENTRYMODE | ID_CURSL | SH_DISPNSHIFT), LCD_INS);     //! Cursor moves right?,
    lcdWrite((DISPCLR), LCD_INS);                                     //! Clear display
    delay(5);
}

/*! ********************************************************************
 * Function: lcdBusy(void)
 *
 * \brief
 *
 * Include:
 *
 * Description:
 *
 * Arguments: None
 *
 * Returns: None
 ********************************************************************/
char lcdBusy(void){

    //LCD_PWR_DIR = LCD_PIN_OUTPUT;       //! Set control lines to output
    LCD_RS_DIR  = LCD_PIN_OUTPUT;
    LCD_RW_DIR  = LCD_PIN_OUTPUT;
    LCD_E_DIR   = LCD_PIN_OUTPUT;
```

```
82
        LCD_DATA_LINE_DIR = LCD_INPUT;        //! Set data lines to input
84
        LCD_E    = LCD_CLKLOW;                //!Set the enable low, ensure clock signal is low
86      LCD_RW   = LCD_READ;                  //!Set RW to read read from LCD
        LCD_RS   = LCD_INS;                   //!Set the register select to low for instruction
88
        LCD_E    = LCD_CLKHGH;                //!Set the enable high, ensure clock signal is high to
90      delay(2);

92      if(LCD_D7 == BF_BUSY){
            LCD_E_DIR    = LCD_PIN_OUTPUT;    //!Set control lines to output
94          LCD_RW_DIR   = LCD_PIN_OUTPUT;
            LCD_RS_DIR   = LCD_PIN_OUTPUT;
96
            LCD_DATA_LINE_DIR = LCD_OUTPUT; //! Set data lines to output
98
            LCD_E = LCD_CLKLOW;               //!Set enable low to allow data to flow through
100         return 1;
        }
102
        else {
104         LCD_E_DIR    = LCD_PIN_OUTPUT;    //!Set control lines to output
            LCD_RW_DIR   = LCD_PIN_OUTPUT;
106         LCD_RS_DIR   = LCD_PIN_OUTPUT;

108         LCD_DATA_LINE_DIR = LCD_OUTPUT; //! Set data lines to output

110         LCD_E = LCD_CLKLOW;               //!Set enable low to allow data to flow through
            return 0;
112     }
    }
114
    /*! ********************************************************************
116  * Function: delay(void)
     *
118  * \brief
     *
120  * Include:
     *
122  * Description:
     *
124  * Arguments: t -
     *
126  * Returns: None
     **********************************************************************/
128 void delay(unsigned int t)  //delay 1 ms
    {
130     unsigned char x, y, z;
        for(z=4; z>0; z--){
132         for (x=t; t>0; t--){
                for(y=100; y>0; y--){}
134         }
        }
136
    }
138
```

```
/*! ***************************************************************************
 * Function: lcdWrite(unsigned char byte, unsigned char mode)
 *
 * \brief
 *
 * Include: LCD.h
 *
 * Description:
 *
 * Arguments: byte -
 *            mode -
 *
 * Returns: None
 ****************************************************************************/
void lcdWrite(unsigned char byte, unsigned char mode){
    while(lcdBusy()){}
    LCD_DATA_LINE_DIR = LCD_OUTPUT; //! Set data lines to output

    LCD_RW = LCD_WRITE;
    LCD_RS = mode;

    LCD_DATA_LINE = byte;      //! Write data to data lines
    delay(2);

    LCD_E = LCD_CLKHGH;   //! Set clock high
    delay(2);
    LCD_E = LCD_CLKLOW;   //! Set clock low
}

/*! ***************************************************************************
 * Function: lcdWriteString(char *string, unsigned char line)
 *
 * \brief
 *
 * Include: LCD.h
 *
 * Description: Writes a string to the LCD at the given line
 *
 * Arguments: string - The string to write
 *            line - the line to write to
 *
 * Returns: None
 ****************************************************************************/
//!Feed character string, and line (1 or 2)
void lcdWriteString(char *string, char line){
    char column, a;         //! Also include information about which row
    if(line==1){column=LINE1;}
    else if(line==2){column=LINE2;}
        for(a=0; a<16; a++){
            if (*string == 0) *string = ' ';
            lcdWrite((SETDDRAMADD | column | a), LCD_INS);
            lcdWrite(*string, LCD_DATA);
            string++;
            delay(5);
        }
    lcdWrite(RTNHOME, LCD_INS);
}
```

```
196
//!Feed character, line (1 or 2), and column(1-16)
198 /*! ************************************************************************
 * Function: lcdWriteChar(unsigned char byte, unsigned char line, unsigned char column)
200 *
 * \brief
202 *
 * Include: LCD.h
204 *
 * Description: Writes a character to the LCD at a given location
206 *
 * Arguments: byte - byte to write
208 *            line - line to wrtie to
 *            column - column to write in
210 *
 * Returns: None
212 ************************************************************************/
//!Feed character, line (1 or 2), and column(1-16)
214 void lcdWriteChar(char byte, char line, char column){
    char row;
216     if(line==1){row=LINE1;}
    else if(line==2){row=LINE2;}
218     lcdWrite((SETDDRAMADD | row | (column-1)), LCD_INS);
    lcdWrite(byte, LCD_DATA);
220     delay(5);
}
```

../Code/LCD.c

9.1.9 MenuDefs.h

```
1 /*
 * File:   MenuDefs.h
3 * Author: Grant
 *
5 * Created on 25 October 2014, 11:49 PM
 */
7
#ifndef MENUDEFS_H
9 #define MENUDEFS_H

11 // Set up the port tp be used for the Factory key
#define FACTORY_SWITCH PORTBbits.RB2
13
#define MENU_ISR 0
15 #define width 80
#define height 24
17
#define MAX_ANGLE_SUPREMUM 45
19 #define MAX_ANGLE_INFIMUM 20
#define MIN_ANGLE_SUPREMUM -20
21 #define MIN_ANGLE_INFIMUM -45

23 #define MAX_RANGE_SUPREMUM 3000      //3m Max range
#define MAX_RANGE_INFIMUM 1000       //1m max range
25 #define MIN_RANGE_SUPREMUM 600       //60cm min range
#define MIN_RANGE_INFIMUM 300        //30cm min range
```

```
27
#define SAMPLE_PER_AVG_MIN 1
29 #define SAMPLE_PER_AVG_MAX 5

31 typedef enum {TOP_LEVEL,
                        TRACKING,
33                      AZ_MENU,
                            AZ_GOTO,
35                          AZ_MAX,
                            AZ_MIN,
37                          AZ_CALIBRATE,
                        EL_MENU,
39                          EL_GOTO,
                            EL_MAX,
41                          EL_MIN,
                            EL_CALIBRATE,
43                      RANGE_MENU,
                            RANGE_MAX,
45                          RANGE_MIN,
                            US_SAMPLE_RATE,
47                          US_SAMPLE_AVG,
                            IR_SAMPLE_RATE,
49                          IR_SAMPLE_AVG,
                            RANGE_RAW,
51                          CALIBRATE_RANGE,
                        SHOW_TEMP,
53                      CALIBRATE_TEMP
                } menuState;
55 typedef enum {LOCAL, REMOTE, FACTORY} userState;

57 #define CLEAR_SCREEN_STRING "\033[2J\033[0;0H"
#define CLEAR_LINE_STRING "\033[2K"
59 #define SERIAL_CURSOR_OFF "\033[?25l"
#define SERIAL_CURSOR_ON "\033[?25h"
61
#define ERR_NUM_OUT_OF_RANGE -1000
63 #define ERR_NOT_NUMERIC -2000
#define ERR_NO_NUMBER -3000
65 #define ESC_PRESSED -4000
#define MINUS_CHAR 0x2D
67
//typedef enum setMenu {AZ_GOTO, AZ_MAX, AZ_MIN, EL_GOTO, EL_MIN, EL_MAX, RNG_MAX, RNG_MIN, IR_
69 typedef void (*numericInputFunction)(int);
typedef void (*voidFunction) (void);
71
/*! ************************************************************************
73  * typedef of menuStruct struct
 *
75  * @brief Defines a sub menu
 *
77  * Description:
 * Stores all data required to define a submenu
79  *
 * Elements:
81  *      - menuNum:
 *      - SerialMessage: The message to be displayed on entering the state
83  *      - LcdTitleMessage: LCD Message to be displayed on entering the state
```

```
 *      - MinVal: The minimum value accepted by the state
 *      - MaxVal: The maximum value accepted by the state
 *      - increment:
 *      - serialDisplayFunction: Function to display result over serial
 *      - confirmFunction: Function to confirm entry
 *      - lcdDisplayFunction: Function to display result on LCD
 *      - returnToPrevious: Function to return to previous menu state
 *****************************************************************************/
struct menuStruct
{
    menuState menuID;
    rom char* serialMessage;    //Serial Message to be displayed on entering the state
    rom char* lcdTitleMessage;      //LCD Message to be displayed on entering the state

    int minVal;
    int maxVal;
    char increment;
    voidFunction serialDisplayFunction;
    numericInputFunction confirmFunction;
    numericInputFunction lcdDisplayFunction;
    voidFunction returnToPrevious;
} menuStruct;

//Global
const rom char newLine[] = "\r\n";
//const rom char CHOOSE[] = "\n\n\tPlease select your option (eg. 2): \r\n";
const rom char CHOOSE[] = "\tPlease select option (eg. 2)";
const rom char CHOOSE2[] = "\tPress Enter to confirm, or Esc to return";
const rom char menuPrefix3[] = "3:";
const rom char menuPrefix4[] = "4:";
const rom char menuPrefix5[] = "5:";
const rom char menuPrefix8[] = "8:";
const rom char goUp[] = "Return to previous menu";
const rom char goUp2[] = "Press Esc to return to the previous menu.";
const rom char errStr[] = "Error: " ;
const rom char inputNumRangeStr[] = "Please enter a number between ";
const rom char currentValueStr[] = "The current value is: ";
const rom char and[] = " and ";
const rom char AzStr[] = "Azimuth";
const rom char ElStr[] = "Elevation";
const rom char RngStr[] = "Range";
const rom char mmStr[] = "Range (mm)";
const rom char sampleRate[] = "Sampling Frequency (Hz)";
const rom char numPerSample[] = "Number of samples per estimate";
const rom char hz[] = "Hz";
const rom char tab[] = "\t";
//

//1: Top
const rom char title[] = "Welcome to the Yavin IV Defence System!";
const rom char welcomeLcd[] = "Home Menu";
//const rom char topOption1[] = "\t1:\tAutomatic Tracking\r\n";
const rom char topOption1[] = "\t1:\tAutomatic Tracking";
//const rom char topOption2[] = "\t2:\tAzimuth Options\r\n";
const rom char topOption2[] = "\t2:\tAzimuth Options";
const rom char topOption3[] = "\t3:\tElevation Options";
const rom char topOption4[] = "\t4:\tRange Options";
```

```
141 const rom char topOption5[] = "\t5:\tDisplay Current Temperature";
    const rom char topOptionCalTemp[] = "\t6:\tCalibrate the Temperature sensor";
143 const rom char topOptionLocal[] = "\t6:\tSwitch to Local mode";
    const rom char topOptionForFactory[] = "\t7:\tSwitch to Factory mode";
145 const rom char topOptionRemote[] = "\t6:\tSwitch to Remote Serial mode";
    const rom char topOptionRemoteLCD[] = "Switch to Remote";
147 //

149 //1.1: Automatic tracking
    const rom char autoSerialMessage[] = "Automatic tracking mode initiated.";
151 const rom char autoLcdTitle[] = "Auto-tracking";
    const rom char autoSearching[] = "Searching...";
153 //

155 const rom char set[] = "Set";
    const rom char range_str[] = "Range";
157 const rom char goto_str[] = "Go to";
    const rom char Options[] = "Options";
159
    //1.2: Azimuth Menu
161 const rom char azMenu[] = "\r\nAzimuth Configuration\r\n";
    const rom char azMenuLcd[] = "Azimuth Options";
163 const rom char azOption1[] = "\t1:\tGo to a specified azimuth angle";
    const rom char azOption2[] = "\t2:\tSet the min azimuth angle";
165 const rom char azOption3[] = "\t3:\tSet the max azimuth angle";
    const rom char azOption4[] = "\t4:\tCalibrate the azimuth motor";
167 //

169 //1.3: Elevation Menu
    const rom char elMenu[] = "Elevation Configuration";
171 const rom char elMenuLcd[] = "Elevation Options";
    const rom char elOption1[] = "\t1:\tGo to a specified elevation angle";
173 const rom char elOption2[] = "\t2:\tSet min elevation angle";
    const rom char elOption3[] = "\t3:\tSet max elevation angle";
175 const rom char elOption4[] = "\t4:\tCalibrate elevation servo";
    //
177
    //1.4: Range Menu
179 const rom char rngMenu[] = "Range Configuration";
    const rom char rngMenuLcd[] = "Range Options";
181 const rom char rngOption1[] = "\t1:\tSet min system range";
    const rom char rngOption2[] = "\t2:\tSet max system range";
183 const rom char rngOption3[] = "\t3:\tView raw ultrasound and IR sensor data";
    const rom char rngOption4[] = "\t4:\tSet the ultrasound sample rate";
185 const rom char rngOption5[] = "\t5:\tSet number of ultrasound samples per estimate";
    const rom char rngOption7[] = "\t6:\tSet number of IR samples per estimate";
187 const rom char rngOption8[] = "\t7:\tCalibrate to a set range";
    //
189
    //1.5: Show Temp
191 const rom char showTempLCDTitle[] = "Display Temp";
    const rom char showTempLCD[] = "Temp = ";
193 const rom char showTemp1[] = "Temp: ";
    const rom char showTemp2[] = " deg Celcius. \r\n\n\tPress Esc to return";
195 //

197 //1.6: Calibrate Temperature
```

```
//
199
//1.2.1: Go to Azimuth Angle
201 const rom char gotoAzAngle[] = "Enter azimuth angle in degrees";
const rom char gotoAzAngleLCD[] = "Go to Azimuth";
203 const rom char gotoElAngle[] = "Enter elevation angle in degrees";
const rom char gotoELAngleLCD[] = "Go to Elevation";
205 const rom char gotoAngle2[] = " Current min & max angles: ";
const rom char angleStr[] = "Angle";
207 //

209 //1.2.2: Set Max Azimuth Angle
const rom char maxAzStr[] = "Max Azimuth";
211 const rom char maxAzSetStr[] = "Set Max Azimuth";
const rom char maxAz1[] = "Enter a new max azimuth: ";
213 const rom char maxAz3[] = "Max azimuth set to ";
//
215
//1.2.2: Set Min Azimuth Angle
217 const rom char currentMinAngleStr[] = "Current min ";
const rom char minAzStr[] = "Min Azimuth";
219 const rom char minAzSetStr[] = "Set Min Azimuth";
const rom char minAz1[] = "Enter a new min azimuth: ";
221 const rom char minAz3[] = "Min azimuth set to ";
//
223
//1.2.3: Calibrate Azimuth / Elevation
225 const rom char calibrateAngle1[] = "Enter Calibration angles: ";
//
227
//1.3.2: Set Max Elevation Angle
229 const rom char maxElStr[] = "Max Elevation";
const rom char maxElSetStr[] = "Set Max Elevation";
231 //const rom char maxEl1[] = "\r\n The current maximum elevation angle is ";
const rom char maxEl1[] = "Enter a new max elevation: ";
233 const rom char maxEl3[] = "Max elevation set to ";
//
235
//1.3.2: Set Min Elevation Angle
237 const rom char minElStr[] = "Min Elevation";
const rom char minElSetStr[] = "Set Min Elevation";
239 //const rom char minEl1[] = "\r\n The current minimum elevation angle is ";
const rom char minEl1[] = "Enter a new min elevation: ";
241 const rom char minEl3[] = "Min elevation set to ";
//
243
//1.4.1 Range Min
245 const rom char minRngStr[] = "Min Range";
const rom char minRngSetStr[] = "Set Min Range";
247 const rom char minRngSerialStr[] = "Enter a new min range: (mm)";
//
249
// 1.4.2 Range Max
251 const rom char maxRngStr[] = "Max Range";
const rom char maxRngSetStr[] = "Set Max Range";
253 const rom char maxRngSerialStr[] = "Enter a new max range (mm):";
//
```

```
255
   // RAW RANGE
257 const rom char rawRangeStr[] = "Displaying raw range in mm:";
   const rom char rawRangeTitle[] = "Raw Range";
259
   // CALIBRATE RANGE
261 const rom char calRangeStr[] = "Place an object 500mm away from the sensor and press enter.";
   const rom char calRangeTitle[] = "Calibrate Range";
263 const rom char calRangeConfirm[] = "Range calibrated";

265 // US SAMPLE RATE
   const rom char usSampleRateStr[] = "Please enter a sample";
267 const rom char usSampleRateTitle[] = "Ultrasound Sample Rate";

269 // US NUMBER OF SAMPLE
   const rom char numSamplesStr[] = "Enter a number of samples to take.";
271 const rom char numSampleTitle[] = "Num. of Samples";

273 #endif  /* MENUDEFS_H */
```

../Code/MenuDefs.h

9.1.10 Menusystem.h

```
   /*
2   * File:    Menusystem.h
    * Author: Jacob
4   *
    * Created on 7 October 2014, 9:56 PM
6   */

8  //Global
   extern const rom char newLine[];
10 extern const rom char CHOOSE[];
   extern const rom char menuPrefix3[];
12 extern const rom char menuPrefix4[];
   extern const rom char menuPrefix5[];
14 extern const rom char menuPrefix8[];
   extern const rom char goUp[];
16 extern const rom char goUp2[];
   extern const rom char inputNumRangeStr[];
18 extern const rom char and[];
   extern const rom char tab[];
20 extern const rom char hz[];
   //
22
   //1: Top
24 extern const rom char title[];
   extern const rom char welcomeLcd[];
26 extern const rom char topOption1[];
   extern const rom char topOption2[];
28 extern const rom char topOption3[];
   extern const rom char topOption4[];
30 extern const rom char topOption5[];
   extern const rom char topOptionCalTemp[];
32 extern const rom char topOptionLocal[];
   extern const rom char topOptionForFactory[];
```

```
34 extern const rom char topOptionRemote[];
   extern const rom char topOptionRemoteLCD[];
36 //

38 //1.1: Automatic tracking
   //extern const rom char automessage[];
40 //extern const rom char autoinit[];
   //extern const rom char autohelp1[];
42 ////extern const rom char autohelp2[];
   ////extern const rom char autohelp3[];
44 //extern const rom char autogo[];
   //extern const rom char autorange[];
46 //extern const rom char autoeleva[];
   //extern const rom char autoazimu[];
48 //extern const rom char autostatu[];
   //extern const rom char autoend[];
50 //

52 //1.2: Azimuth Menu
   extern const rom char azMenu[];
54 extern const rom char azMenuLcd[];
   extern const rom char azOption1[];
56 extern const rom char azOption2[];
   extern const rom char azOption3[];
58 extern const rom char azOption4[];
   //
60
   //1.3: Elevation Menu
62 extern const rom char elMenu[];
   extern const rom char elMenuLcd[];
64 extern const rom char elOption1[];
   extern const rom char elOption2[];
66 extern const rom char elOption3[];
   extern const rom char elOption4[];
68 //

70 //1.4: Range Menu
   extern const rom char rngMenu[];
72 extern const rom char rngMenuLcd[];
   extern const rom char rngOption1[];
74 extern const rom char rngOption2[];
   extern const rom char rngOption3[];
76 extern const rom char rngOption4[];
   extern const rom char rngOption5[];
78 extern const rom char rngOption6[];
   extern const rom char rngOption7[];
80 //

82 //1.5: Show Temp
   extern const rom char showTempLCDTitle[];
84 extern const rom char showTempLCD[];
   extern const rom char showTemp1[];
86 extern const rom char showTemp2[];
   //
88
   //1.6: Calibrate Temperature
90 //
```

```
92 //1.2.1: Go to Azimuth Angle
   extern const rom char gotoAzAngle[];
94 extern const rom char gotoAzAngleLCD[];
   extern const rom char gotoElAngle[];
96 extern const rom char gotoELAngleLCD[];
   extern const rom char gotoAngle2[];
98 extern const rom char angleStr[];
   //
100
   //1.2.2: Set Max Azimuth Angle
102 extern const rom char maxAzStr[];
   extern const rom char maxAzSetStr[];
104 extern const rom char maxAz1[];
   extern const rom char maxAz3[];
106 //

108 //1.2.2: Set Min Azimuth Angle
   extern const rom char currentMinAngleStr[];
110 extern const rom char minAzStr[];
   extern const rom char minAzSetStr[];
112 extern const rom char minAz1[];
   extern const rom char minAz3[];
114 //

116 //1.2.3: Calibrate Azimuth / Elevation
   extern const rom char calibrateAngle1[];
118 //

120 //1.3.2: Set Max Elevation Angle
   extern const rom char maxElStr[];
122 extern const rom char maxElSetStr[];
   //extern const rom char maxEl1[];
124 extern const rom char maxEl1[];
   extern const rom char maxEl3[];
126 //

128 //1.3.2: Set Min Elevation Angle
   extern const rom char minElStr[];
130 extern const rom char minElSetStr[];
   //extern const rom char minEl1[];
132 extern const rom char minEl1[];
   extern const rom char minEl3[];
134 //

136 //1.4.1 Range Min
   extern const rom char minRngStr[];
138 extern const rom char minRngSetStr[];
   extern const rom char minRngSerialStr[];
140 //

142 // 1.4.2 Range Max
   extern const rom char maxRngStr[];
144 extern const rom char maxRngSetStr[];
   extern const rom char maxRngSerialStr[];
146 //
```

```
148 /*! ****************************************************************
    * Function: initialiseMenu(systemState *state)
150 *
    * \brief Initialises the menu system
152 *
    * Include: Menusystem.h
154 *
    * Description: initialises the menu system so that it is fully operational
156 *
    * Arguments: state - The current system state
158 *
    * Returns: None
160 ******************************************************************/
    extern void initialiseMenu(systemState *state);
162
    /*! ****************************************************************
164 * Function: checkForSerialInput(void)
    *
166 * Include: Menusystem
    *
168 * Description: Checks the serial/local buffers for inputs
    *
170 * Arguments: None
    *
172 * Returns: 1 if input has been received, 0 otherwise
    ******************************************************************/
174 extern char checkForSerialInput(void);

176 /*! ****************************************************************
    * Function: serviceMenu(void)
178 *
    * \brief services any user interface with the menu
180 *
    * Include:
182 *
    * Description: Checks if the user has made any inputs to the system. If not
184 *              the function simply returns. If they have then it services
    *              the inputs, displays the correct outputs and performs the
186 *              specified actions
    *
188 * Arguments: None
    *
190 * Returns: None
    ******************************************************************/
192 extern void serviceMenu(void);


194
    extern void dispTrack(TrackingData target);
196 extern void dispSearching(void);
    extern void dispRawRange(void);
```

../Code/Menusystem.h

9.1.11 Menusystem.c

```
    /*!*****************************************************************************
2 * File:    Menusystem.c
```

```
 * Author:
 *
 * Description:
 *
 * Duties:
 *
 * Functions:
 *
 * Created on 16 September 2014, 6:47 PM
 ******************************************************************************/

#include "Common.h"
#include "Serial.h"
#include "User_Interface.h"
#include "Menusystem.h"
#include "LCD.h"
#include "Range.h"
#include "PanTilt.h"
#include "Temp.h"

#include "MenuDefs.h"


// Serial Display
//void sendROM(const static rom char *romchar);
static void sendROM(const rom char *romchar);
static void clearScreen(void);
static void sendNewLine(char length);
static void filler(char length);
static void errOutOfRange(int lowerBound, int upperBound);
static char* intToAscii(int num);
static void autodisp(void);

int parseNumeric(char *input);
static int getSerialNumericInput(void);
static void configureTimer0(void);
static int getLocalInputMenu(int maxStates, int (*function)(int));
//static int getLocalPotResult(int min, int max, int interval);

// Confirm Methods
static void setValue(int input);
static void setMenu(struct menuStruct menu);
static void navigateTopMenu(int inputResult);
static void noFunctionNumeric(int input);
static void setCalibrateRange(int num);

// Return Methods
static void noFunction(void);
static void returnToTopMenu(void);
static void exitFromTracking(void);
static void returnToAzMenu(void);
static void returnToElMenu(void);
static void returnToRngMenu(void);

// Navigation Methods
static void navigateTopMenu(int inputResult);
```

```
60 static void navigateAzimuthMenu(int inputResult);
   static void navigateElevationMenu(int inputResult);
62 static void navigateRangeMenu(int inputResult);

64 // Display methods
   static void dispTopOptions(void);
66 static void displayMenuSerial();
   static void dispTopOptions(void);
68 static void dispAzOptions(void);
   static void dispElOptions(void);
70 static void dispRngOptions(void);
   static void dispTempSerialMessage(void);
72 static void dispSerialMessage(void);
   static void dispSetValueMessage(void);
74
   /// LCD Display Functions
76 static void dispLCDTopMenu(int option);
   static void dispLCDAzMenu(int option);
78 static void dispLCDElMenu(int option);
   static void dispLCDRngMenu(int option);
80 static void dispLCDNum(int option);

82 //// Menus
   struct menuStruct topMenu = {TOP_LEVEL, title, welcomeLcd,1, 6, 1, dispTopOptions, navigateTopM
84 struct menuStruct AzMenu = {AZ_MENU, azMenu, azMenuLcd,1, 6, 1, dispAzOptions, navigateAzimuthM
   struct menuStruct ElMenu = {EL_MENU, elMenu, elMenuLcd,1, 6, 1, dispElOptions, navigateElevatio
86 struct menuStruct RangeMenu = {RANGE_MENU, rngMenu, rngMenuLcd,1, 6, 1, dispRngOptions, navigat

88 /// Remote/Local Functions
   struct menuStruct AzGoto = {AZ_GOTO, gotoAzAngle, gotoAzAngleLCD,MIN_ANGLE_INFIMUM, MAX_ANGLE_S
90 struct menuStruct ElGoto = {EL_GOTO, gotoElAngle, gotoELAngleLCD,MIN_ANGLE_INFIMUM, MAX_ANGLE_S
   struct menuStruct AzMin = {AZ_MIN, minAz1, minAzSetStr,MIN_ANGLE_INFIMUM, MIN_ANGLE_SUPREMUM,
92 struct menuStruct AzMax = {AZ_MAX, maxAz1, maxAzSetStr,MAX_ANGLE_INFIMUM, MAX_ANGLE_SUPREMUM,
   struct menuStruct ElMin = {EL_MIN, minEl1, minElSetStr,MIN_ANGLE_INFIMUM, MIN_ANGLE_SUPREMUM,
94 struct menuStruct ElMax = {EL_MAX, maxEl1, maxElSetStr,MAX_ANGLE_INFIMUM, MAX_ANGLE_SUPREMUM,
   struct menuStruct RngMin = {RANGE_MIN, minRngSerialStr, minRngSetStr, MIN_RANGE_INFIMUM, MIN_RA
96 struct menuStruct RngMax = {RANGE_MAX, maxRngSerialStr, maxRngSetStr, MAX_RANGE_INFIMUM, MAX_R
   struct menuStruct ShowTemp = {SHOW_TEMP, showTempLCDTitle, showTempLCDTitle, 0, 0, 0, dispTemp
98 struct menuStruct Tracking = {TRACKING, autoSerialMessage, autoLcdTitle, 0, 0, 0, dispSerialMe
   struct menuStruct RawRange;// = {RANGE_RAW, rawRangeStr, rawRangeTitle, 0, 0, 0, dispSerialMess
100 struct menuStruct NumSamples;
   struct menuStruct UsSampleRate;
102 struct menuStruct calibrateRangeMenu;
   /*! Global variable with the current menu position */
104 struct menuStruct m_currentMenu;

106 /*! Global variable with the current user mode: Local, remote or factory */
   static userState m_userMode;
108
   static systemState *m_trackingState;
110
   // ****************************************************************************
112 // ********************** MENU STRUCTURE FUNCTIONS ****************************
   // ****************************************************************************
114
   /*! ***************************************************************
116  * Function: sendROM(void)
```

```
 *
 * @brief
 *
 * Include: Local to Menusystem.c
 *
 * @description: Transmits the given string from ROM over serial
 *
 * @input The string to transmit
 *
 * Returns: None
 ***************************************************************************/
static void sendROM(const rom char *romchar) {
    char temp[80] = {0};
    int j;

    // Convert the string from ROM to RAM
    strcpypgm2ram(temp, romchar);
    transmit(temp);
    for (j = 0; j < 8000; j++); //Some Arbitrary Delay
}

/*! ************************************************************************
 * Function: sendNewLine(char length)
 *
 * @brief
 *
 * Include: Local to Menusystem.c
 *
 * @description: Prints a number of new line (\r\n) charaters.
 *
 * @input The number of new lines to print
 *
 * Returns: None
 ***************************************************************************/
static void sendNewLine(char length)
{
    char index;
    char temp[height + 2] = {0};        // This is two greater for the \r and end message char
    int j;

    // Fills the buffer with \n characters
    for (index = 0; index<length; index++) {
        temp[index] = '\n';
    }
    // End with a line return
    temp[index] = '\r';
    transmit(temp);
    for (j = 0; j < 2000; j++);
}

/*!
 * Clears the Serial display
 */
/*! ************************************************************************
 * Function: clearScreen(void)
 *
 * @brief Clears the Serial Display
```

```
174  *
     * Include: Local to Menusystem.c
176  *
     * @description: Transmits the given string from ROM over serial
178  *
     * @input The string to transmit
180  *
     * Returns: None
182  ********************************************************************/
   static void clearScreen()
184 {
        int j;
186     char clearScreen[] = CLEAR_SCREEN_STRING;     // Clears the screen and writes from the top
        transmit(clearScreen);
188
        for (j = 0; j < 1000; j++);
190 }

192 static void filler(char length) {
        char index = 0;
194     char temp[81] = {0};
        int j;
196     for (; length > 0; length--) {
            temp[index] = '+';
198         index++;
        }
200     transmit(temp);
        for (j = 0; j < 3000; j++);
202 }

204 /*! ********************************************************************
     * Function: initialiseMenu(void)
206  *
     * \brief Initialises the menu system
208  *
     * Include: Menusystem.h
210  *
     * Description: initialises the menu system so that it is fully operational
212  *
     * Arguments: A pointer to the global tracking state
214  *
     * Returns: None
216  ********************************************************************/
   void initialiseMenu(systemState *state) {
218     m_userMode = REMOTE;
        configureSerial();
220     //@TODO WHILE UNPLUGGED configLCD();
        configUSER();
222     setMenu(topMenu);
        m_trackingState = state;
224
        RawRange;// = {RANGE_RAW, rawRangeStr, rawRangeTitle, 0, 0, 0, dispSerialMessage, noFunctio
226     RawRange.confirmFunction = noFunctionNumeric;
                RawRange.menuID = RANGE_RAW;
228     RawRange.serialMessage = rawRangeStr;
        RawRange.lcdTitleMessage = rawRangeTitle;
230     RawRange.minVal = 0;
```

```
    RawRange.maxVal = 0;
    RawRange.increment = 0;
    RawRange.serialDisplayFunction = dispSerialMessage;
    RawRange.returnToPrevious = returnToRngMenu;
    RawRange.lcdDisplayFunction = noFunctionNumeric;

    calibrateRangeMenu.confirmFunction = setCalibrateRange;
    calibrateRangeMenu.menuID = CALIBRATE_RANGE;
    calibrateRangeMenu.serialMessage = calRangeStr;
    calibrateRangeMenu.lcdTitleMessage = calRangeTitle;
    calibrateRangeMenu.minVal = 0;
    calibrateRangeMenu.maxVal = 0;
    calibrateRangeMenu.increment = 0;
    calibrateRangeMenu.serialDisplayFunction = dispSerialMessage;
    calibrateRangeMenu.returnToPrevious = returnToRngMenu;
    calibrateRangeMenu.lcdDisplayFunction = noFunctionNumeric;

    NumSamples.confirmFunction = setValue;
    NumSamples.menuID = US_SAMPLE_AVG;
    NumSamples.serialMessage = numSamplesStr;
    NumSamples.lcdTitleMessage = numSampleTitle;
    NumSamples.minVal = 1;
    NumSamples.maxVal = 3;
    NumSamples.increment = 1;
    NumSamples.serialDisplayFunction = dispSetValueMessage;
    NumSamples.returnToPrevious = returnToRngMenu;
    NumSamples.lcdDisplayFunction = noFunctionNumeric;

    UsSampleRate.confirmFunction = setValue;
    UsSampleRate.menuID = US_SAMPLE_RATE;
    UsSampleRate.serialMessage = usSampleRateStr;
    UsSampleRate.lcdTitleMessage = usSampleRateTitle;
    UsSampleRate.minVal = 1;
    UsSampleRate.maxVal = 40;
    UsSampleRate.increment = 1;
    UsSampleRate.serialDisplayFunction = dispSetValueMessage;
    UsSampleRate.returnToPrevious = returnToRngMenu;
    UsSampleRate.lcdDisplayFunction = noFunctionNumeric;

}

/*! ********************************************************************
 * Function: checkForSerialInput(void)
 *
 * Include: Menusystem
 *
 * Description: Checks the serial/local buffers for inputs
 *
 * Arguments: None
 *
 * Returns: 1 if input has been received, 0 otherwise
 ********************************************************************/
char checkForSerialInput(void)
{
    return receiveCR() | receiveEsc(); //!Wait until the receive buffer is no longer empty
                                       //!Indicating that a command has been passed
}
```

```
288
/*! ********************************************************************
290  * Function: parseNumeric(char *number)
     *
292  * \brief parses user input string into a number
     *
294  * Include:
     *
296  * Description: Converts ASCII input to a number, and records an error for
     *              non-numeric input, or if the number is larger than 4 digits.
298  *              No number used by the user in this program will be larger
     *              than 4 digits.
300  *
     * Arguments: The ASCII string to decode
302  *
     * Returns: The integer result of the string
304  *          ERR_NOT_NUMERIC for any non-numeric input
     *          ERR_NUM_OUT_OF_RANGE for 0 digits or 5+ digits
306  ********************************************************************/
int parseNumeric(char *number)
308 {
    char digits = 0;
310 char index;
    char tempDigits[4] = {0};
312 signed char multiplier = 1;

314 // Array to hold powers of 10 for multiplication
    int decimalPowers[] = {1, 10, 100, 1000};
316 int result = 0;

318 // Check if any number was given.
    if (*number == 0x0D) return ERR_NO_NUMBER;
320
    if (*number == MINUS_CHAR)
322 {
        // Negative number, so multiply by -1 and move to the next digit
324     multiplier = -1;
        number++;
326 }
    // Only need first 4 digits, as the biggest input is 3
328 for (; *number != 0x0D; digits++)    //Count number of digits
    {
330     // Return error if more than 4 digits were given
        if (digits >= 6) return ERR_NUM_OUT_OF_RANGE;
332     if (*number < 0x30 && *number > 0x39) return ERR_NOT_NUMERIC;    // Not a numeric inpu

334     // Store the number in a temporary buffer
        tempDigits[digits] = *number - 0x30;
336     number++;
    }
338
    // Multiply the digits to get the result
340 for(index = 0; digits >0; digits--)
    {
342     result += ((int) tempDigits[index] * decimalPowers[digits-1]);
        index++;
344 }
```

```
346     // Return the positive or negative number
        return multiplier * result;
348 }

350 /*!
     * Description: Displays a number out of range error
352  */
    static void errOutOfRange(int lowerBound, int upperBound)
354 {
    //    sendROM(errStr);
356     sendROM(inputNumRangeStr);
        transmit(intToAscii(lowerBound));
358     sendROM(and);
        transmit(intToAscii(upperBound));
360     sendNewLine(1);
    }
362
    /*! ********************************************************************
364  * Function: waitForNumericInput
     *
366  * \brief Waits for input and parses number
     *
368  * Include:
     *
370  * Description: Waits in a loop for a user command ended with a new line character.
     *              Once a command is received, it is converted into the appropriate
372  *              numeric value that the user has given.
     *
374  * Arguments: None
     *
376  * Returns: The integer result of the string input
     *          ERR_NOT_NUMERIC for any non-numeric input
378  *          ERR_NUM_OUT_OF_RANGE for 0 digits or 5+ digits
     *          ESC_PRESSED if escape was pressed
380  ********************************************************************/
    static int getSerialNumericInput()
382 {
        char userget[80] = {0};
384     if (receiveEsc())
        {
386         popEsc();
            return ESC_PRESSED;
388     }
        readString(userget);    //!Get the input string and store it in @userget
390     return parseNumeric(userget);
    }
392
    /*!
394  * Description: Returns the value of the potentiometer on the user
     *              interface, given a maximum and minimum value, and
396  *              the interval between values (eg 10-100 in multiples
     *              of 10).
398  */
    //static int getLocalPotResult(int min, int max, int interval)
400 //{
    //    int adcResult = readDial((max - min)/interval);
```

```
//     adcResult = min + interval*adcResult;
//     return adcResult;
//}

/*!
 * Description: Converts a number to a string
 * Can only print numbers under 8 digits
 */
static char* intToAscii(int num)
{
    char string[10] = {0};
    sprintf(string, "%d", num);
    return string;
}

/*! ********************************************************************
 * Function: serviceMenu(void)
 *
 * \brief services any user interface with the menu
 *
 * Include:
 *
 * Description: Checks if the user has made any inputs to the system. If not
 *              the function simply returns. If they have then it services
 *              the inputs, displays the correct outputs and performs the
 *              specified actions
 *
 * Arguments: None
 *
 * Returns: None
 ********************************************************************/
void serviceMenu(void)
{
    char buttonInput;
    char string[50];
    int numericInpt;

    // Go to factory mode
    if (FACTORY_SWITCH == 1 && m_userMode == REMOTE)
    {
        m_userMode = FACTORY;
        setMenu(m_currentMenu);
    }
    // Rest back to serial mode
    else if(FACTORY_SWITCH == 0 && m_userMode == FACTORY)
    {
        m_userMode = REMOTE;
        setMenu(topMenu);
    }

    if (m_userMode == REMOTE || m_userMode == FACTORY)
    {
        // Check for serial input
        if (checkForSerialInput())
        {
            // Handle serial input
            numericInpt = getSerialNumericInput();
```

```
                sendNewLine(1);
460             sendROM("DEBUG: Value entered = ");
                transmit(intToAscii(numericInpt));
462             sendNewLine(1);
                clearReceive();
            }
            else
466         {
                return;
468         }
        }
470 //      else
    //      {
472 //          if (userEmpty())
    //          {
474 //              // Display the current value on the LCD
    //              // Get the Potentiometer result for the current menu
476 //              numericInpt = getLocalPotResult(m_currentMenu.minVal, m_currentMenu.maxVal, m_cu
    //              // Display string on LCD
478 //              m_currentMenu.lcdDisplayFunction(numericInpt);
    //              return;
480 //          }
    //          else
482 //          {
    //              // Confirm the user's input
484 //              buttonInput = userPop();
    //              if (buttonInput == CONFIRM_CHAR)
486 //              {
    //                  numericInpt = getLocalPotResult(m_currentMenu.minVal, m_currentMenu.maxVal,
488 //              }
    //              else
490 //              {
    //                  // Back was pressed
492 //                  numericInpt =  ESC_PRESSED;
    //              }
494 //          }
    //      }
496
        /// If Esc or Back button pressed, return
498     if (numericInpt == ESC_PRESSED)
        {
500         m_currentMenu.returnToPrevious();
            return;
502     }
        else
504     {
            /// Otherwise Confirm the selection
506         m_currentMenu.confirmFunction(numericInpt);
            return;
508     }
    }
510
    static void setCalibrateRange(int num)
512 {
        calibrateRange(500);
514     sendROM(tab);
        sendROM(calRangeConfirm);
```

```
516     sendNewLine(1);
    }
518
    /*!
520  * Description: General funtion for menus which set values
     *              (Such as Set Max Range). This calls the
522  *              appropriate function, and transmits user messages.
     */
524 static void setValue(int input)
    {
526     char string[20] = " set to \0";
        char stringLcd[20] = {0};
528     Direction dir;

530     // Sends a new line
        sendNewLine(1);
532
        // Handle inpropper input values
534     if (input < m_currentMenu.minVal || input > m_currentMenu.maxVal)
        {
536         errOutOfRange(m_currentMenu.minVal, m_currentMenu.maxVal);
            sendNewLine(1);
538         return;
        }
540
        // Handle Different cases
542     // @TODO handle current values
        // @TODO Integrate
544     switch (m_currentMenu.menuID)
        {
546         case AZ_GOTO:
                dir.azimuth = input;
548             dir.elevation = getDir().elevation;
                move(dir);
550             sendROM(angleStr);
                break;
552         case EL_GOTO:
                dir.azimuth = getDir().azimuth;
554             dir.elevation = input;
                move(dir);
556             sendROM(angleStr);
                break;
558         case AZ_MIN:
                setMinAzimuthAngle((char) input);
560             AzGoto.minVal = input;
                sendROM(angleStr);
562             break;
            case AZ_MAX:
564             setMaxAzimuthAngle((char) input);
                AzGoto.maxVal = input;
566             sendROM(angleStr);
                break;
568         case EL_MIN:
                setMinElevationAngle((char) input);
570             ElGoto.minVal = input;
                sendROM(angleStr);
572             break;
```

```
            case EL_MAX:
574             setMaxElevationAngle((char) input);
                ElGoto.maxVal = input;
576             sendROM(angleStr);
                break;
578         case RANGE_MIN:
                setMinRange(input);
580             sendROM(mmStr);
                break;
582         case RANGE_MAX:
                setMaxRange(input);
584             sendROM(mmStr);
                break;
586         case US_SAMPLE_RATE:
                setUsSampleRate(input);
588             sendROM(sampleRate);
                break;
590         case US_SAMPLE_AVG:
                setNumSamples(input);
592             sendROM(numPerSample);
                break;
594     }

596     // Transmit the final part of the sentence
        transmit(string);
598     transmit(intToAscii(input));
        sendNewLine(1);
600
        //lcdWriteString(strcpypgm2ram(stringLcd, "OK!"), 2);
602 }

604 static void setMenu(struct menuStruct menu)
    {
606     char stringLcd[20] = {0};
        m_currentMenu = menu;
608     if (m_userMode == REMOTE || m_userMode == FACTORY) displayMenuSerial();
        //lcdWriteString(strcpypgm2ram(stringLcd, m_currentMenu.lcdTitleMessage), 1);
610 }

612 static void noFunctionNumeric(int input)
    {
614     return;
    }
616
    static void noFunction(void)
618 {
        return;
620 }

622 static void returnToTopMenu(void)
    {
624     setMenu(topMenu);
    }
626
    static void exitFromTracking(void)
628 {
        m_trackingState->previous = m_trackingState->current;
```

```
630     m_trackingState->current = MENU;
        setMenu(topMenu);
632 }

634 static void returnToAzMenu(void)
    {
636     setMenu(AzMenu);
    }
638 static void returnToElMenu(void)
    {
640     setMenu(ElMenu);
    }
642 static void returnToRngMenu(void)
    {
644     setMenu(RangeMenu);
    }
646
    // ********************************************************************************
648 // ************************* NAVIGATE MENUS ********************************
    // ********************************************************************************
650
    static void navigateTopMenu(int inputResult) {
652     switch (inputResult) {
            case 1:
654             //auto();
                setMenu(Tracking);
656             if (m_trackingState->current = MENU)
                {
658                 m_trackingState->previous = m_trackingState->current;
                    m_trackingState->current = SRCH;
660             }
                break;
662         case 2:
                // Go to the Azimuth menu
664             setMenu(AzMenu);
                break;
666         case 3:
                // Go to the Elevation Menu
668             setMenu(ElMenu);
                break;
670         case 4:
                // Go to Range Menu
672             setMenu(RangeMenu);;
                break;
674             setMenu(RangeMenu);
                break;
676         case 5:
                // Show Temperature
678             setMenu(ShowTemp);
                break;
680         case 6:
                // If Factory mode is on, calibrate Temperature
682             if (m_userMode == FACTORY) {
                    //setMenu(Calibrate);
684                 break;
                }
686             else
```

```
            {
688                 // Switch user mode!
//                  if (m_userMode == REMOTE) {
690 //                      m_userMode = LOCAL;
//                  } else if (m_userMode == LOCAL) {
692 //                      m_userMode = REMOTE;
//                  }
694 //                  setMenu(topMenu);
                    break;
696             }
        default:
698             errOutOfRange(1, 5);
                break;
700     }
}
702
static void navigateAzimuthMenu(int inputResult) {
704     switch (inputResult) {
            case 1:
706             // Go to elevation angle
                setMenu(AzGoto);
708             break;
            case 2:
710             // Set elevation minimum
                setMenu(AzMin);
712             break;
            case 3:
714             // Set elevation maximum
                setMenu(AzMax);
716             break;
            case 4:
718             if (m_userMode == FACTORY) {
                    // Calibrate the elevation servo
720             } else {
                    // Go back one level
722                 m_currentMenu.returnToPrevious();
                }
724             break;
            case 5:
726             if (m_userMode == FACTORY) {
                    // Go back!
728                 m_currentMenu.returnToPrevious();
                    break;
730             }
                // If not in factory mode, this is an error;
732         default:
                errOutOfRange(1, 4);
734             break;
    }
736 }

738 static void navigateRangeMenu(int inputResult)
{
740     switch (inputResult) {
            case 1:
742             // Set the minimum range
                setMenu(RngMin);
```

```
744             break;
            case 2:
746             // Set the maximum range
                setMenu(RngMax);
748             break;
            case 3:
750             if (m_userMode == FACTORY) {
                    // View the raw range data
752                 setMenu(RawRange);
                    m_trackingState->previous = m_trackingState->current;
754                 m_trackingState->current = RAW_RANGE_STATE;
                } else {
756                 // Go back one level
                    m_currentMenu.returnToPrevious();
758             }
                break;
760         case 4:
                if (m_userMode == FACTORY) {
762                 // Set the ultrasound sample rate
                    setMenu(UsSampleRate);
764             } else {
                    errOutOfRange(1, 3); // Display an error
766             }
                break;
768         case 5:
                if (m_userMode == FACTORY) {
770                 // Set the number of US estimations per sample
                    setMenu(NumSamples);
772             } else {
                    errOutOfRange(1, 3); // Display an error
774             }
                break;
776         case 6:
                if (m_userMode == FACTORY) {
778                 // Set the IR sample rate
                } else {
780                 errOutOfRange(1, 3); // Display an error
                }
782             break;
            case 7:
784             if (m_userMode == FACTORY) {
                    // Calibrate Range
786                 setMenu(calibrateRangeMenu);
                } else {
788                 errOutOfRange(1, 3); // Display an error
                }
790             break;
            case 8:
792             if (m_userMode == FACTORY) {
                    m_currentMenu.returnToPrevious();
794             } else {
                    errOutOfRange(1, 3); // Display an error
796             }
                break;
798         default:
                if (m_userMode == FACTORY) {
800                 errOutOfRange(1, 7);
```

```
            } else {
802             errOutOfRange(1, 3); // Display an error
            }
804         break;
    }
806 }

808 static void navigateElevationMenu(int inputResult)
    {
810     switch (inputResult) {
        case 1:
812         // Go to elevation angle
            setMenu(ElGoto);
814         break;
        case 2:
816         // Set elevation minimum
            setMenu(ElMin);
818         break;
        case 3:
820         // Set elevation maximum
            setMenu(ElMax);
822         break;
        case 4:
824         if (m_userMode == FACTORY) {
                // Calibrate the elevation servo
826         } else {
                m_currentMenu.returnToPrevious();
828             // Go back one level
            }
830         break;
        case 5:
832         if (m_userMode == FACTORY) {
                // Calibrate the elevation servo
834             break;
            }
836         // If not in factory mode, this is an error;
        default:
838         if (m_userMode == FACTORY) errOutOfRange(1, 5);
            else errOutOfRange(1, 4);
840         break;
    }
842 }

844 // ********************************************************************************
    // ************************* SERIAL DISPLAY ***************************************
846 // ********************************************************************************

848 /*!
     * Display the current menu Title and other information over serial
850  */
    static void displayMenuSerial()
852 {
        int j;
854     sendNewLine(1);
        clearScreen();
856     filler(width);
        sendNewLine(1);
```

```
858     transChar('\t');
        sendROM(m_currentMenu.lcdTitleMessage);
860     sendNewLine(1);
        filler(width);
862     sendNewLine(2);
        m_currentMenu.serialDisplayFunction();
864     sendNewLine(2);
        filler(width);
866     for (j=0;j<1000;j++);
        sendNewLine(1);
868 }

870 /*!
     * Display the user options for the top level home menu
872  */
    static void dispTopOptions(void) {
874     sendROM("\t");
        sendROM(m_currentMenu.serialMessage);
876     sendNewLine(2);
        sendROM(topOption1);
878     sendNewLine(1);
        sendROM(topOption2);
880     sendNewLine(1);
        sendROM(topOption3);
882     sendNewLine(1);
        sendROM(topOption4);
884     sendNewLine(1);
        sendROM(topOption5);
886     sendNewLine(1);

888     if (m_userMode == FACTORY)
        {
890         sendROM(topOptionCalTemp);
            sendNewLine(1);
892         sendROM(topOptionForFactory);
        }
894     else
        {
896         sendROM(topOptionLocal);
        }
898
        sendNewLine(2);
900     sendROM(CHOOSE);
        sendNewLine(1);
902     sendROM(CHOOSE2);
    }
904
    /*!
906  * Display the user options for the Azimuth menu
     */
908 static void dispAzOptions()
    {
910     sendROM(azOption1);
        sendNewLine(1);
912     sendROM(azOption2);
        sendNewLine(1);
914     sendROM(azOption3);
```

```
        sendNewLine(1);
916     if (m_userMode == FACTORY)
        {
918         sendROM(azOption4);
            sendNewLine(1);
920         transChar('\t');
            sendROM(menuPrefix5);
922         transChar('\t');
            sendROM(goUp);
924     }
        else
926     {
            transChar('\t');
928         sendROM(menuPrefix4);
            transChar('\t');
930         sendROM(goUp);
        }
932     sendNewLine(2);
        sendROM(CHOOSE);
934     sendNewLine(1);
        sendROM(CHOOSE2);
936 }

938 /*!
     * Display the user options for the Azimuth menu
940  */
    static void dispElOptions()
942 {
        sendROM(elOption1);
944     sendNewLine(1);
        sendROM(elOption2);
946     sendNewLine(1);
        sendROM(elOption3);
948     sendNewLine(1);
        if (m_userMode == FACTORY)
950     {
            sendROM(elOption4);
952         sendNewLine(1);
            transChar('\t');
954         sendROM(menuPrefix5);
            transChar('\t');
956         sendROM(goUp);
        }
958     else
        {
960         transChar('\t');
            sendROM(menuPrefix4);
962         transChar('\t');
            sendROM(goUp);
964     }
        sendNewLine(2);
966     sendROM(CHOOSE);
        sendNewLine(1);
968     sendROM(CHOOSE2);
    }
970
    /*!
```

```
 * Display the user options for the Azimuth menu
 */
static void dispRngOptions()
{
    //sendROM(rngOption1);
    sendROM(rngOption1);
    sendNewLine(1);
    //sendROM(rngOption2);
    sendROM(rngOption2);
    sendNewLine(1);
    if (m_userMode == FACTORY)
    {
        sendROM(rngOption3);
        sendNewLine(1);
        sendROM(rngOption4);
        sendNewLine(1);
        sendROM(rngOption5);
        sendNewLine(1);
        sendROM(rngOption7);
        sendNewLine(1);
        sendROM(rngOption8);
        sendNewLine(1);
        transChar('\t');
        sendROM(menuPrefix8);
        transChar('\t');
        sendROM(goUp);
    }
    else
    {
        transChar('\t');
        sendROM(menuPrefix3);
        transChar('\t');
        sendROM(goUp);
    }
    sendNewLine(2);
    sendROM(CHOOSE);
    sendNewLine(1);
    sendROM(CHOOSE2);
}

/*!
 * Display the Show Temperature serial message
 */
static void dispTempSerialMessage(void)
{
    int temp = readTemp();
    sendROM("\t");
    sendROM(showTemp1);
    transmit(intToAscii(temp));
    sendROM(showTemp2);
}

/*!
 * Display the menu serial message
 */
static void dispSerialMessage(void)
{
```

```
    sendROM("\t");
    sendROM(m_currentMenu.serialMessage);
}

/*!
 * Display the menu serial message with the maximum, minimum and current values
 */
static void dispSetValueMessage(void)
{
    int num = 0;
    Direction dir;

    sendROM("\t");
    sendROM(m_currentMenu.serialMessage);
    sendNewLine(1);

    // Enter a value between x and y
    sendROM("\t");
    sendROM(inputNumRangeStr);
    transmit(intToAscii(m_currentMenu.minVal));
    sendROM(and);
    transmit(intToAscii(m_currentMenu.maxVal));
    sendNewLine(1);

    // The current value is:
    sendROM("\t");
    sendROM(currentValueStr);
    switch (m_currentMenu.menuID)
    {
        case AZ_GOTO:
            dir = getDir();
            num = (int) dir.azimuth;
            break;
        case EL_GOTO:
            dir = getDir();
            num = (int) dir.elevation;
            break;
        case AZ_MIN:
            num = getMinAzimuthAngle();
            break;
        case AZ_MAX:
            num = getMaxAzimuthAngle();
            break;
        case EL_MIN:
            num = getMaxAzimuthAngle();
            break;
        case EL_MAX:
            num = getMaxAzimuthAngle();
            break;
        case RANGE_MIN:
            num = getMinRange();
            break;
        case RANGE_MAX:
            num = getMaxRange();
            break;
        case US_SAMPLE_RATE:
            num = getUsSampleRate();
```

```
1086                break;
            case US_SAMPLE_AVG:
1088                num = getNumSamples();
                    break;
1090        }
        transmit(intToAscii(num));
1092        sendNewLine(1);
    }
1094
    /*!
1096 * Displays the Tracking data over serial
     */
1098 void dispTrack(TrackingData target)
    {
1100        unsigned int j;

1102        const char rng_string[] = "Range: ";
        const char inc_string[] = "Elevation: ";
1104        const char az_string[] = "Azimuth: ";
        const char separator[] = "\t";
1106        char unitsR[] = "mm";
        char unitsD[] = "deg";
1108        char num[5];

1110        if (target.azimuth > 100 || target.azimuth < -100) return;

1112        // If in tracking mode, Display the Azimuth, Elevation and Range
        if (m_currentMenu.menuID == TRACKING && transmitComplete())
1114        {
            sendROM(CLEAR_LINE_STRING);
1116            transChar('\r');
            transmit(rng_string);
1118            sprintf(num, "%d", target.range);
            transmit(num);
1120            transmit(unitsR);
            transmit(separator);
1122
            transmit(az_string);
1124
            sprintf(num, "%d", target.azimuth);
1126            transmit(num);
            transmit(unitsD);
1128            transmit(separator);

1130            transmit(inc_string);
            sprintf(num, "%d", target.elevation);
1132            transmit(num);
            transmit(unitsD);
1134            transmit(separator);
        }
1136 }

1138 /*!
     * Displays the Tracking data over serial
1140  */
    void dispRawRange()
1142 {
```

```
    unsigned int j;

    const char us_rng_string[] = "US Range: ";
    const char ir_string[] = "IR Range: ";
    const char separator[] = "\t";
    char unitsR[] = "mm";
    char num[5];

    unsigned int irRange = rawRangeIR();
    unsigned int usRange = rawRangeUS();

    // If in tracking mode, Display the Azimuth, Elevation and Range
    if (m_currentMenu.menuID == RANGE_RAW && transmitComplete())
    {
        sendROM(CLEAR_LINE_STRING);
        transChar('\r');
        transmit(us_rng_string);
        sprintf(num, "%u", usRange);
        transmit(num);
        transmit(unitsR);
        transmit(separator);

        transmit(ir_string);
        sprintf(num, "%u", irRange);
        transmit(num);
        transmit(unitsR);
    }
}

void dispSearching()
{
    // Clear display and search!
    if (m_currentMenu.menuID == TRACKING && transmitComplete())
    {
        // Transmit the buffer to clear the extra chars
        transChar('\r');
        transmit(CLEAR_LINE_STRING);
        transChar('\r');
        sendROM(autoSearching);

    }
}

// ********************************************************************************
// ************************** LCD FUNCTIONS ***************************************
// ********************************************************************************
//
///*!
// * Displays the current potentiometer reading on the LCD based on the
// * menu options contextualised by the Home menu.
// */
//static void dispLCDTopMenu(int option)
//{
//    char string[20] = {0};
//    switch(option)
//    {
//        case 1:
```

```
1200 //             //Automatic Tracking;
     //             strcpypgm2ram(string, Tracking.lcdTitleMessage);
1202 //             break;
     //         case 2:
1204 //             // Azimuth Menu
     //             strcpypgm2ram(string, AzMenu.lcdTitleMessage);
1206 //             break;
     //         case 3:
1208 //             // Elevation Menu
     //             strcpypgm2ram(string, ElMenu.lcdTitleMessage);
1210 //             break;
     //         case 4:
1212 //             // Range Menu
     //             strcpypgm2ram(string, RangeMenu.lcdTitleMessage);
1214 //             break;
     //         case 5:
1216 //             // Show Temp
     //             strcpypgm2ram(string, ShowTemp.lcdTitleMessage);
1218 //             break;
     //         case 6:
1220 //             // Switch to Remote
     //             strcpypgm2ram(string, topOptionRemoteLCD);
1222 //             break;
     //         default:
1224 //             strcpypgm2ram(string, "ERROR");
     //     }
1226 //     //lcdWriteString(string, 2);
     //}
1228 ///*!
     // * Displays the current potentiometer reading on the LCD based on the
1230 // * menu options contextualised by the Azimuth menu.
     // */
1232 //static void dispLCDAzMenu(int option)
     //{
1234 //     char string[20] = {0};
     //     switch(option)
1236 //     {
     //         case 1:
1238 //             // Go to Azimuth
     //             strcpypgm2ram(string, AzGoto.lcdTitleMessage);
1240 //             break;
     //         case 2:
1242 //             // Set Min Azimuth
     //             strcpypgm2ram(string, AzMin.lcdTitleMessage);
1244 //             break;
     //         case 3:
1246 //             // Set Max Azimuth
     //             strcpypgm2ram(string, AzMax.lcdTitleMessage);
1248 //             break;
     //         default:
1250 //             strcpypgm2ram(string, "ERROR");
     //     }
1252 //     //lcdWriteString(string, 2);
     //}
1254 ///*!
     // * Displays the current potentiometer reading on the LCD based on the
1256 // * menu options contextualised by the Elevation menu.
```

```
// */
//static void dispLCDElMenu(int option)
//{
//    char string[20] = {0};
//    switch(option)
//    {
//        case 1:
//            // Go to Elevation
//            strcpypgm2ram(string,  ElGoto.lcdTitleMessage);
//            break;
//        case 2:
//            // Set Min Elevation
//            strcpypgm2ram(string, ElMin.lcdTitleMessage);
//            break;
//        case 3:
//            // Set Max Elevation
//            strcpypgm2ram(string, ElMax.lcdTitleMessage);
//            break;
//        default:
//            strcpypgm2ram(string, "ERROR");
//    }
//    //lcdWriteString(string, 2);
//}
//
///*!
// * Displays the current potentiometer reading on the LCD based on the
// * menu options contextualised by the Range menu.
// */
//static void dispLCDRngMenu(int option)
//{
//    char string[20] = {0};
//    switch(option)
//    {
//        case 1:
//            // Set Min Range
//            strcpypgm2ram(string, RngMin.lcdTitleMessage);
//            break;
//        case 3:
//            // Set Max Range
//            strcpypgm2ram(string, RngMax.lcdTitleMessage);
//            break;
//        default:
//            strcpypgm2ram(string, "ERROR");
//    }
//    //lcdWriteString(string, 2);
//}
//
///*!
// * Description: Displays the current converted value from the potentiometer
// *              onto the LCD display
// *
// * Arguments: The integer converted from the ADC
// */
//static void dispLCDNum(int option)
//{
//    char *string = intToAscii(option);
//    //lcdWriteString(string, 2);
```

```
1314 //}
```

../Code/Menusystem.c

9.1.12 newmain.c

```
1  /*! ************************************************************************
   * File:   newmain.c
3  *
   * Author: Grant, Bas, Ayush, Zhenning, Jacob, Alvaro
5  *
   * Description:
7  * Controls the main system state of the product based on a state transition
   * type template.
9  *
   * Created on 7 September 2014, 4:12 PM
11 ************************************************************************/

13 //#pragma config WDT = OFF  //Turns watchdog Timer off
   //#pragma config OSC = HS   //The crystal oscillator set to "High Speed"
15 //#pragma config LVP = OFF  //
   //#pragma config DEBUG = ON
17
   #include "Common.h"
19
   #include "Tracking.h"
21 #include "Range.h"
   #include "User_Interface.h"
23 #include "Serial.h"
   #include "PanTilt.h"
25 #include "Menusystem.h"
   #include "ConfigRegs18f4520.h"
27
   //Local Function Prototypes:
29 static void initialization(systemState *state);

31 /*! ********************************************************************
   * Function: main(void)
33 *
   * \brief Program entry point
35 *
   * Include: Local to newmain.c
37 *
   * Description: stores the current system state and manages all transitions
39 *
   * Arguments: None
41 *
   * Returns: None
43 ************************************************************************/
   void main() {
45     systemState state = {INIT, UNDEF};
       TrackingData target;
47
       OSCCON = 0x64;              //Set the oscillator
49     WDTCONbits.SWDTEN = 1;       //Enable the Watch Dog time

51     initialiseMenu(&state);
```

```
53      for (;;)
        {
55          //Clear Watch Dog Timer
            _asm CLRWDT _endasm

57
            serviceMenu();

59
            switch (state.current)
61          {
                case INIT:
63                  initialization(&state);
                    break;
65              case SRCH:
                    search(&state);
67                  dispSearching();
                    break;
69              case TRCK:
                    target = track(&state);
71                  dispTrack(target);
                    break;
73              case MENU:
                    break;
75              case RAW_RANGE_STATE:
                    range();
77                  dispRawRange();
                    Delay10KTCYx(255);
79                  break;
                default:     //Any other undefined state
81                  NEXT_STATE(INIT, state);       //Set the next state to be Initialize
                    break;
83          }
        }
85
    }
87
    /*! ***********************************************************************
89   * Function: initialization(systemState *state)
     *
91   * Include: Local to newmain.c
     *
93   * Description: Initializes the system, turns on the sensors and checks if
     * they are ready to begin working
95   *
     * Arguments: state - The current state of the system
97   *
     * Returns: The next system state - At the moment always just transitions to CHECK
99   ************************************************************************/
    static void initialization(systemState *state)
101 {
        Direction dir = {0,-45};
103     unsigned int i = 0;

105     configureSerial();
        configureTracking();
107     configUSER();
```

```
109     move(dir);

111     for (i = 0; i < 10000; i++);

113     NEXT_STATE_PTR(MENU, state);    //Go to the searching state
}
```

../Code/newmain.c

9.1.13 PanTilt.h

```
/*! ***************************************************************************
 * File:   PanTilt.h
 * Author: Grant
 *
 * Description:
 * This file contains the public interface for the PanTile module
 *
 * Created on 16 September 2014, 6:33 PM
 *****************************************************************************/

//Ensure that there is only 1 inclusion of this file in the preprocessor execution
#ifndef PANTILT_H

//True if Pan ISR concerns pan tilt module
#define PAN_TILT_ISR     CCP2_INT

/*=============================================================================*/
//External declarations of public access functions
/*=============================================================================*/

/*! ***************************************************************************
 * Function: configureBase(void)
 *
 * Include: PanTilt.h
 *
 * Description: Configures the Pan Tilt mechanism for operation
 *
 * Arguments: None
 *
 * Returns: None
 *****************************************************************************/
extern void configureBase(void);

/*! ***************************************************************************
 * Function: move(Direction destination)
 *
 * Include: PanTilt.h
 *
 * Description: Moves the pan tilt actuator to the specified destination
 *
 * Arguments: destionation - A struct containing the desired azimuth and inclination
 *
 * Returns: None
 *****************************************************************************/
extern void move(Direction destination);

```

```
/*! ***************************************************************
 * Function: increment(Direction difference)
 *
 * Include: PanTilt.h
 *
 * Description: Moves the pan tilt actuator to the specified destination
 *
 * Arguments: destionation - A struct containing the desired azimuth and inclination
 *
 * Returns: None
 ****************************************************************/
extern void increment(Direction difference);

/*! ***************************************************************
 * Function: incrementFine(Direction difference)
 *
 * Include: PanTilt.h
 *
 * Description: Moves the pan tilt actuator to the specified (Relative) destination
 *
 * Arguments: destionation - A struct containing the desired azimuth and inclination
 *
 * Returns: None
 ****************************************************************/
extern void incrementFine(Direction difference);

/*! ***************************************************************
 * Function: getDir(void)
 *
 * Include: PanTilt.h
 *
 * Description: returns the current position of the pan tilt mechanism
 *
 * Arguments: None
 *
 * Returns: A struct containing the azimuth and inclination
 ****************************************************************/
extern Direction getDir(void);

/*! ***************************************************************
 * Function: calibratePanTilt(Direction reference)
 *
 * Include: PanTilt.h
 *
 * Description: Calibrates the pan tile mechanism offset so that any future reference
 *              to move to the reference value specified in the function call
 *              will move the pan tilt back to the current position.
 *
 * Arguments: reference - A struct containing the azimuth and inclinaion you
 *                        wish to define as this position.
 *
 * Returns: None
 ****************************************************************/
extern void calibratePanTilt(Direction reference);

/*! ***************************************************************
 * Function: calibratePanTiltRange(Direction reference)
```

```
104  *
     * Include: PanTilt.h
106  *
     * Description: Calibrates the pan tile mechanism's range so that any future reference
108  *              to move to the reference value specified in the function call
     *              will move the pan tilt back to the current position.
110  *
     * Arguments: reference - A struct containing the azimuth and inclinaion you
112  *                        wish to define as this position.
     *
114  * Returns: None
     ******************************************************************************/
116 extern void calibratePanTiltRange(Direction reference);

118 /*! ***************************************************************************
     * Function: rawDir(void)
120  *
     * Include: PanTilt.h
122  *
     * Description: returns the current PanTile position without calibrating
124  *
     * Arguments: None
126  *
     * Returns: The position of the pan tilt without any calibration
128  ******************************************************************************/
    extern Direction rawDir(void);
130
    /*! ***************************************************************************
132  * Function: updated(void)
     *
134  * Include: PanTilt.h
     *
136  * Description: returns true if the last move or increment or incrementFine
     *              function has taken effect. The new direction is only loaded
138  *              in at the end of the PDM, so it could take up to 0.02 seconds
     *              for the change to take effect.
140  *
     * Arguments: delay - a pointer to the delay variable
142  *
     * Returns: None
144  ******************************************************************************/
    extern char updated(void);
146
    /*! ***************************************************************************
148  * Function: panTiltISR(void)
     *
150  * Include: PanTilt.h
     *
152  * Description: Acts as the ISR for the PanTilt module
     *
154  * Arguments: None
     *
156  * Returns: None
     ******************************************************************************/
158 extern void panTiltISR(void);

160 /*! ***************************************************************************
```

```
 * Function: getMaxAzimuthAngle(void)
 *
 * Include: PanTilt.h
 *
 * Description: returns the maximum angle of the azimuth servo
 *
 * Arguments: None
 *
 * Returns: A char with the maximum azimuth angle.
 ********************************************************************/
extern char getMaxAzimuthAngle(void);

/*! ********************************************************************
 * Function: getMinAzimuthAngle(void)
 *
 * Include: PanTilt.h
 *
 * Description: returns the minimum angle of the azimuth servo
 *
 * Arguments: None
 *
 * Returns: A char with the minimum azimuth angle.
 ********************************************************************/
extern char getMinAzimuthAngle(void);

/*! ********************************************************************
 * Function: getMaxElevationAngle(void)
 *
 * Include: PanTilt.h
 *
 * Description: returns the maximum angle of the elevation servo
 *
 * Arguments: None
 *
 * Returns: A char with the maximum elevation angle.
 ********************************************************************/
extern char getMaxElevationAngle(void);

/*! ********************************************************************
 * Function: getMinElevationAngle(void)
 *
 * Include: PanTilt.h
 *
 * Description: returns the minimum angle of the elevation servo
 *
 * Arguments: None
 *
 * Returns: A char with the minimum elevation angle.
 ********************************************************************/
extern char getMinElevationAngle(void);


/*! ********************************************************************
 * Function: setMaxAzimuthAngle(void)
 *
 * Include: PanTilt.h
 *
```

```
218  * Description: sets the maximum angle of the azimuth servo
     *
220  * Arguments: The maximum angle (as char) to set for the azimuth servo
     *
222  * Returns: None.
     ********************************************************************/
224 extern void setMaxAzimuthAngle(char p_angle);

226 /*! ********************************************************************
     * Function: setMinAzimuthAngle(void)
228  *
     * Include: PanTilt.h
230  *
     * Description: sets the minimum angle of the azimuth servo
232  *
     * Arguments: The minimum angle (as char) to set for the azimuth servo
234  *
     * Returns: None.
236  ********************************************************************/
    extern void setMinAzimuthAngle(char p_angle);
238
    /*! ********************************************************************
240  * Function: setMaxElevationAngle(void)
     *
242  * Include: PanTilt.h
     *
244  * Description: sets the maximum angle of the elevation servo
     *
246  * Arguments: The maximum angle (as char) to set for the elevation servo
     *
248  * Returns: None.
     ********************************************************************/
250 extern void setMaxElevationAngle(char p_angle);

252 /*! ********************************************************************
     * Function: setMinElevationAngle(void)
254  *
     * Include: PanTilt.h
256  *
     * Description: sets the minimum angle of the elevation servo
258  *
     * Arguments: The minimum angle (as char) to set for the elevation servo
260  *
     * Returns: None.
262  ********************************************************************/
    extern void setMinElevationAngle(char p_angle);
264
    #define PANTILT_H
266 #endif
```

../Code/PanTilt.h

9.1.14 PanTilt.c

```
    /*! ********************************************************************************
2    * File:   ADC.c
     * Author: Grant
```

```
4  *
   * Description:
6  * Contains all the functionality for the Pan Tilt module. All variables and settings
   * concerning the pan tilt module including the current direction, PDM delay info, min
8  * and max settings are private to this module. The interface functions allow all valid
   * access to the module.
10 *
   * Duties:
12 *          -Software interface to the Pan Tilt Module
   *          -Moves Pan Tilt
14 *          -Reads current Pan Tilt position (based on PDM's)
   *          -Generate PDM signals
16 *
   * Functions:
18 *      Local:
   *           void validate(unsigned int *delay);
20 *           Direction delay2Direction(Delay dly);
   *           Delay direction2Delay(Direction dir);
22 *
   *      Public Interface:
24 *          void configureBase(void);
   *          void move(Direction destination);
26 *          void increment(Direction difference);
   *          void incrementFine(Direction difference);
28 *          Direction getDir(void);
   *          void calibratePanTilt(Direction reference);
30 *          Direction rawDir(void);
   *          char updated(void);
32 *          void panTiltISR(void);
   *          char getMaxAzimuthAngle(void);
34 *          char getMinAzimuthAngle(void);
   *          char getMaxElevationAngle(void);
36 *          char getMinElevationAngle(void);
   *          void setMaxAzimuthAngle(char p_angle);
38 *          void setMinAzimuthAngle(char p_angle);
   *          void setMaxElevationAngle(char p_angle);
40 *          void setMinElevationAngle(char p_angle);
   *
42 * Created on 16 September 2014, 6:47 PM
   ******************************************************************************/
44
#include "Common.h"
46 #include <pwm.h>
#include <timers.h>
48
typedef struct
50 {
    signed int AzimuthDelay;
52  signed int InclinationDelay;
} Delay;
54
//Define the PWM output pins
56 #define AZ_PWM_PIN PORTCbits.RC0
#define IN_PWM_PIN PORTCbits.RC1
58
#ifdef MNML     //Using minimal board, or picdem - Clock speeds differ by x4
60 #define DUTY_CYCLE_TIME 2500
```

```
#define PWM_PERIOD 50000 //The period for 50Hz at 2.5MHz
#define PWM_HALF_PERIOD 25000 //Half the period for 50Hz at 1MHz
#else
#define DUTY_CYCLE_TIME 1000
#define PWM_PERIOD 20000 //The period for 50Hz at 2.5MHz
#define PWM_HALF_PERIOD 10000 //Half the period for 50Hz at 1MHz
#endif

//Interrupt Latency
#define LATENCY 340

#define SERVO_INIT() TRISCbits.RC0 = 0; TRISCbits.RC1 = 0; PORTCbits.RC0 = 0; PORTCbits.RC1 =

//Local Function Prototypes
static void validate(unsigned int *delay);
static Direction delay2Direction(Delay dly);
static Delay direction2Delay(Direction dir);

//Static calibration offset
static Direction calibration_offset = { 0, 3 };
static Direction arcRange = { 94, 103 };
static Delay global_delay;

//Static Max/Min
static signed char azimuth_angle_max;
static signed char azimuth_angle_min;
static signed char elevation_angle_max;
static signed char elevation_angle_min;

//Static Current direction
static Direction current_direction;
static volatile char changed = 0;

/*! ************************************************************************
 * Function: configureBase(void)
 *
 * Include: PanTilt.h
 *
 * Description: Configures the Pan Tilt mechanism for operation
 *
 * Arguments: None
 *
 * Returns: None
 ************************************************************************/
void configureBase(void)
{
    unsigned char config;

    //Set the initial servo PWM's to zeros
    Direction zero = { 0, 0 };
    global_delay = direction2Delay(zero);

    INT_SETUP();

    SERVO_INIT();

    config = T3_16BIT_RW & T3_PS_1_1& T3_SYNC_EXT_OFF & T3_SOURCE_INT;
```

```
118
        OpenTimer3(config);
120
        //Timer1 source for CCP1, and timer3 source for CCP2
122     SetTmrCCPSrc(T1_CCP1_T3_CCP2);

124     config = COM_INT_ON & COM_UNCHG_MATCH;

126     OpenCompare2(config, PWM_PERIOD);

128     azimuth_angle_max = 45;
        azimuth_angle_min = -45;
130     elevation_angle_max = 45;
        elevation_angle_min = -45;
132 }

134 /*! ***********************************************************************
 * Function: move(Direction destination)
136 *
 * Include: PanTilt.h
138 *
 * Description: Moves the pan tilt actuator to the specified destination
140 *
 * Arguments: destionation - A struct containing the desired azimuth and inclination
142 *
 * Returns: None
144 ************************************************************************/
    void move(Direction destination)
146 {
        int i = 0;
148     global_delay = direction2Delay(destination);

150     //Update the current_direction
        current_direction = delay2Direction(global_delay);
152
        for (i = 0; i < 2000; i++);
154 }

156 /*! ***********************************************************************
 * Function: increment(Direction difference)
158 *
 * Include: PanTilt.h
160 *
 * Description: Moves the pan tilt actuator to the specified destination
162 *
 * Arguments: destionation - A struct containing the desired azimuth and inclination
164 *
 * Returns: None
166 ************************************************************************/
    void increment(Direction difference)
168 {
        current_direction.azimuth += difference.azimuth;
170     current_direction.elevation += difference.elevation;

172     move(current_direction);
    }
174
```

```
/*! ***************************************************************
 * Function: incrementFine(Direction difference)
 *
 * Include: PanTilt.h
 *
 * Description: Moves the pan tilt actuator to the specified (Relative) destination
 *
 * Arguments: destionation - A struct containing the desired azimuth and inclination
 *
 * Returns: None
 ****************************************************************/
void incrementFine(Direction difference)
{
    unsigned int az, inc;
    az = global_delay.AzimuthDelay + difference.azimuth - PWM_HALF_PERIOD;
    inc = global_delay.InclinationDelay + difference.elevation + LATENCY;

    //Ensure that the delays are still within the max and min duty cycles
    validate(&az);
    validate(&inc);

    global_delay.AzimuthDelay = az + PWM_HALF_PERIOD;
    global_delay.InclinationDelay = inc - LATENCY;

    //Update the current_direction
    current_direction = delay2Direction(global_delay);
    // Delay
    for(az = 0;az < 20000;az++);
}

/*! ***************************************************************
 * Function: getDir(void)
 *
 * Include: PanTilt.h
 *
 * Description: returns the current position of the pan tilt mechanism
 *
 * Arguments: None
 *
 * Returns: A struct containing the azimuth and inclination
 ****************************************************************/
Direction getDir(void)
{
    return current_direction;
}

/*! ***************************************************************
 * Function: getMaxAzimuthAngle(void)
 *
 * Include: PanTilt.h
 *
 * Description: returns the maximum angle of the azimuth servo
 *
 * Arguments: None
 *
 * Returns: A char with the maximum azimuth angle.
 ****************************************************************/
```

```
232 char getMaxAzimuthAngle(void)
    {
234     return azimuth_angle_max;
    }
236
    /*! ***************************************************************
238  * Function: getMinAzimuthAngle(void)
     *
240  * Include: PanTilt.h
     *
242  * Description: returns the minimum angle of the azimuth servo
     *
244  * Arguments: None
     *
246  * Returns: A char with the minimum azimuth angle.
     ****************************************************************/
248 char getMinAzimuthAngle(void)
    {
250     return azimuth_angle_min;
    }
252
    /*! ***************************************************************
254  * Function: getMaxElevationAngle(void)
     *
256  * Include: PanTilt.h
     *
258  * Description: returns the maximum angle of the elevation servo
     *
260  * Arguments: None
     *
262  * Returns: A char with the maximum elevation angle.
     ****************************************************************/
264 char getMaxElevationAngle(void)
    {
266     return elevation_angle_max;
    }
268
    /*! ***************************************************************
270  * Function: getMinElevationAngle(void)
     *
272  * Include: PanTilt.h
     *
274  * Description: returns the minimum angle of the elevation servo
     *
276  * Arguments: None
     *
278  * Returns: A char with the minimum elevation angle.
     ****************************************************************/
280 char getMinElevationAngle(void)
    {
282     return elevation_angle_min;
    }
284
    /*! ***************************************************************
286  * Function: setMaxAzimuthAngle(void)
     *
288  * Include: PanTilt.h
```

```
 *
 * Description: sets the maximum angle of the azimuth servo
 *
 * Arguments: The maximum angle (as char) to set for the azimuth servo
 *
 * Returns: None.
 ********************************************************************/
void setMaxAzimuthAngle(char p_angle)
{
    azimuth_angle_max = p_angle;
}

/*! ********************************************************************
 * Function: setMinAzimuthAngle(void)
 *
 * Include: PanTilt.h
 *
 * Description: sets the minimum angle of the azimuth servo
 *
 * Arguments: The minimum angle (as char) to set for the azimuth servo
 *
 * Returns: None.
 ********************************************************************/
void setMinAzimuthAngle(char p_angle)
{
    azimuth_angle_min = p_angle;
}
/*! ********************************************************************
 * Function: setMaxElevationAngle(void)
 *
 * Include: PanTilt.h
 *
 * Description: sets the maximum angle of the elevation servo
 *
 * Arguments: The maximum angle (as char) to set for the elevation servo
 *
 * Returns: None.
 ********************************************************************/
void setMaxElevationAngle(char p_angle)
{
    elevation_angle_max = p_angle;
}

/*! ********************************************************************
 * Function: setMinElevationAngle(void)
 *
 * Include: PanTilt.h
 *
 * Description: sets the minimum angle of the elevation servo
 *
 * Arguments: The minimum angle (as char) to set for the elevation servo
 *
 * Returns: None.
 ********************************************************************/
void setMinElevationAngle(char p_angle)
{
    elevation_angle_min = p_angle;
```

```
346 }
348 /*! ***********************************************************************
    * Function: calibratePanTilt(Direction reference)
350 *
    * Include: PanTilt.h
352 *
    * Description: Calibrates the pan tile mechanism offset so that any future reference
354 *              to move to the reference value specified in the function call
    *              will move the pan tilt back to the current position.
356 *
    * Arguments: reference - A struct containing the azimuth and inclinaion you
358 *                        wish to define as this position.
    *
360 * Returns: None
    ************************************************************************/
362 void calibratePanTilt(Direction reference)
    {
364     calibration_offset.azimuth += current_direction.azimuth - reference.azimuth;
        calibration_offset.elevation += current_direction.elevation - reference.elevation;
366     current_direction = reference;
    }
368
    /*! ***********************************************************************
370 * Function: calibratePanTiltRange(Direction reference)
    *
372 * Include: PanTilt.h
    *
374 * Description: Calibrates the pan tile mechanism's range so that any future reference
    *              to move to the reference value specified in the function call
376 *              will move the pan tilt back to the current position.
    *
378 * Arguments: reference - A struct containing the azimuth and inclinaion you
    *                        wish to define as this position.
380 *
    * Returns: None
382 ************************************************************************/
    void calibratePanTiltRange(Direction reference)
384 {
        arcRange.azimuth = arcRange.azimuth * reference.azimuth / current_direction.azimuth;
386     arcRange.elevation = arcRange.elevation * reference.elevation / current_direction.elevatio
        current_direction = reference;
388 }

390 /*! ***********************************************************************
    * Function: rawDir(void)
392 *
    * Include: PanTilt.h
394 *
    * Description: returns the current PanTile position without calibrating
396 *
    * Arguments: None
398 *
    * Returns: The position of the pan tilt without any calibration
400 ************************************************************************/
    Direction rawDir(void)
402 {
```

```
404 }
406 /*! ************************************************************************
    * Function: panTiltISR(void)
408 *
    * Include: PanTilt.h
410 *
    * Description: Acts as the ISR for the PanTilt module
412 *
    * Arguments: None
414 *
    * Returns: None
416 *************************************************************************/
    void panTiltISR(void)
418 {
        unsigned int timer_value;
420     static Delay current_delay;
        unsigned char i = 0;
422
        if (CCP2_INT)
424     {
            timer_value = ReadTimer3();
426
            if (timer_value > PWM_PERIOD)
428         {
                IN_PWM_PIN = 1;
430             WriteTimer3(0);      //Clear timer2
                current_delay = global_delay;   //update the static delay
432             changed = 1;         //Indicate the change has been loaded
                OpenCompare2(COM_INT_ON & COM_UNCHG_MATCH, current_delay.InclinationDelay);
434         }
            else if (timer_value > current_delay.AzimuthDelay)
436         {
                AZ_PWM_PIN = 0;
438             OpenCompare2(COM_INT_ON & COM_UNCHG_MATCH, PWM_PERIOD);
            }
440         else if (timer_value > PWM_HALF_PERIOD)
            {
442             AZ_PWM_PIN = 1;
                OpenCompare2(COM_INT_ON & COM_UNCHG_MATCH, current_delay.AzimuthDelay);
444         }
            else if (timer_value > current_delay.InclinationDelay)
446         {
                IN_PWM_PIN = 0;
448             OpenCompare2(COM_INT_ON & COM_UNCHG_MATCH, PWM_HALF_PERIOD);
            }
450
            PIR2bits.CCP2IF = 0;
452     }
    }
454
    /*! ************************************************************************
456  * Function: direction2Delay(DirectionState dir)
     *
458  * Include: Local to PanTilt.c
     *
```

```
460  * Description: Converts an azimuth and inclination direction into a pwm period
     *
462  * Arguments: dir - Locally defined struct containing the desired Azimuth
     *                  and Inclination
464  *
     * Returns: Delay - The required PDM delay to move the servos to the given direction
466  *
     * Remark: This function relies on the ARC_RANGE macro being set correctly. This
468  *         should hold the value of the maximum angle that the servos can be commanded
     ********************************************************************/
470 static Direction delay2Direction(Delay dly)
    {
472     Direction ret;

474     ret.azimuth = ((dly.AzimuthDelay - DUTY_CYCLE_TIME - PWM_HALF_PERIOD) * (long int)arcRange
        ret.elevation = ((dly.InclinationDelay - DUTY_CYCLE_TIME + LATENCY) * (long int)arcRange.e
476
        ret.azimuth = ret.azimuth - DIV_2(arcRange.azimuth) - calibration_offset.azimuth;
478     ret.elevation = -(ret.elevation - DIV_2(arcRange.elevation) - calibration_offset.elevation

480     return ret;
    }
482
    /*! ********************************************************************
484  * Function: direction2Delay(DirectionState dir)
     *
486  * Include: Local to PanTilt.c
     *
488  * Description: Converts an azimuth and inclination direction into a pwm period
     *
490  * Arguments: dir - Locally defined struct containing the desired Azimuth
     *                  and Inclination
492  *
     * Returns: Delay - The required PDM delay to move the servos to the given direction
494  *
     * Remark: This function relies on the ARC_RANGE macro being set correctly. This
496  *         should hold the value of the maximum angle that the servos can be commanded
     ********************************************************************/
498 static Delay direction2Delay(Direction dir)
    {
500     Delay result;
        unsigned int az, inc;
502
        az = DUTY_CYCLE_TIME + (dir.azimuth + DIV_2(arcRange.azimuth) + calibration_offset.azimuth
504     inc = DUTY_CYCLE_TIME + (-dir.elevation + DIV_2(arcRange.elevation) + calibration_offset.e

506     validate(&az);
        validate(&inc);
508
        result.AzimuthDelay = az + PWM_HALF_PERIOD;
510     result.InclinationDelay = inc - LATENCY;

512     return result;
    }
514

516 /*! ********************************************************************
```

```
 * Function: validate(unsigned int *delay)
 *
 * Include: Local to PanTilt.c
 *
 * Description: Limits the duration of the PDM to between 1000us and 2000us
 *
 * Arguments: delay - a pointer to the delay variable
 *
 * Returns: None
 ******************************************************************************/
static void validate(unsigned int *delay)
{
    if (*delay < DUTY_CYCLE_TIME)
    {
        *delay = DUTY_CYCLE_TIME;
    }
    if (*delay > 2*DUTY_CYCLE_TIME)
    {
        *delay = 2 * DUTY_CYCLE_TIME;
    }
}

/*! ****************************************************************************
 * Function: updated(void)
 *
 * Include: PanTilt.h
 *
 * Description: returns true if the last move or increment or incrementFine
 *              function has taken effect. The new direction is only loaded
 *              in at the end of the PDM, so it could take up to 0.02 seconds
 *              for the change to take effect.
 *
 * Arguments: delay - a pointer to the delay variable
 *
 * Returns: None
 ******************************************************************************/
char updated(void)
{
    return changed;
}
```

../Code/PanTilt.c

9.1.15 Range.h

```
/*! ****************************************************************************
 * File:   Range.h
 * Author: Grant
 *
 * Description:
 * Contains the public interface for the ultrasonic module. This file contains
 * all the external declarations, macros and global variables for using and
 * Interfacing with the Range module.
 *
 * Created on 15 September 2014, 11:24 AM
 ******************************************************************************/

```

```
//Ensure that there is only 1 inclusion of this file in the preprocessor execution
#ifndef RANGE_H

//Ultrasonic module interrupt macro
#define RANGE_INT CCP1_INT | TMR1_INT

/*=============================================================================*/
//External declarations of the public access functions
/*=============================================================================*/

/*! ***************************************************************************
 * Function: configureRange(void)
 *
 * Include: Range.h
 *
 * Description: Configures the Range module
 *
 * Arguments: None
 *
 * Returns: None
 *****************************************************************************/
extern void configureRange(void);

/*! ***************************************************************************
 * Function: configureAD(void)
 *
 * Include: Range.h
 *
 * Description: Configures the ADC,
 * In ADCON1, we set right-justified mode, and select AN0 as the input channel.
 * In ADCON0, we set a sample rate of Fosc/8, select AN0, and enable the ADC.
 * Arguments: None
 *
 * Returns: None
 *****************************************************************************/
extern void configureAD(void);

/*! ***************************************************************************
 * Function: range(void)
 *
 * Include: Range.h
 *
 * @brief Samples the range
 *
 * Description: Takes a number of samples of the ultrasonic sensor at a specified
 *              rate. Continues to sample the IR sensor at a different rate while
 *              sampling the ultrasonic. Then combines the ranges and sets the
 *              target state
 *
 * Arguments: None
 *
 * Returns: the range to the target in mm
 *****************************************************************************/
extern unsigned int range(void);

/*! ***************************************************************************
 * Function: getLastRange(void)
```

```
70  *
    * Include: Range.h
72  *
    * @brief Samples the range
74  *
    * Description: Returns the last range sample, so the data can be used without
76  *              re-sampling the range finding sensors
    *
78  * Arguments: None
    *
80  * Returns: the last Range sample data
    ***************************************************************************/
82 extern unsigned int lastRange(void);

84 /*! ***************************************************************************
    * Function: rangeISR(void)
86  *
    * Include: range.h
88  *
    * Description: Called when an range related interrupt is fired, acts
90  *              as the service routine for the rangefinding module.
    *
92  * Arguments: None
    *
94  * Returns: None
    ***************************************************************************/
96 extern void rangeISR(void);

98 /*! ***************************************************************************
    * Function: calibrateRange(unsigned int reference)
100 *
    * Include:
102 *
    * Description: Calibrates the range of both the IR and ultrasonic sensors
104 *              based on a given range.
    *
106 * Arguments: reference - The distance in mm to calibrate the current
    *                        measurements from
108 *
    * Returns: None
110 ***************************************************************************/
   extern void calibrateRange(signed int distance);
112

   /*! ***************************************************************************
114 * Function: rawRangeIR(void)
    *
116 * Include: Range.h
    *
118 * Description: Returns the last IR range reading
    *
120 * Arguments: None
    *
122 * Returns: the range in mm
    ***************************************************************************/
124 extern unsigned int rawRangeIR(void);

126 /*! ***************************************************************************
```

```
 * Function: rawRangeUS(void)
 *
 * Include: Range.h
 *
 * Description: Returns the last US range reading
 *
 * Arguments: None
 *
 * Returns: the range in mm
 ****************************************************************************/
extern unsigned int rawRangeUS(void);


/*! ***************************************************************************
 * Function: getTargetState(void)
 *
 * Include: Range.h
 *
 * Description: Returns the target state from the last range reading. E.g.
 *              Good track, or direction not quite correct as US returned,
 *              but IR didn't and was within IR range etc.
 *
 * Arguments: None
 *
 * Returns: the target state
 ****************************************************************************/
extern TargetState getTargetState(void);

/*! ***************************************************************************
 * Function: readTargetState(void)
 *
 * Include: Range.h
 *
 * Description: Does the same thing as getTargetState, but actually performs
 *              a range() read
 *
 * Arguments: None
 *
 * Returns: the target state
 ****************************************************************************/
extern TargetState readTargetState(void);

/*! ***************************************************************************
 * Function: setMaxRange()
 *
 * Include: Range.h
 *
 * Description: Sets the maximum range of the device
 *
 * Arguments: range - The max range to set in mm
 *
 * Returns: N/A
 ****************************************************************************/
extern void setMaxRange(int range);

/*! ***************************************************************************
 * Function: setMinRange()
```

```
184  *
     * Include: Range.h
186  *
     * Description: Sets the minimum range of the device
188  *
     * Arguments: range - The max range to set in mm
190  *
     * Returns: N/A
192  ********************************************************************/
   extern void setMinRange(int range);
194
   /*! ********************************************************************
196  * Function: getMaxRange()
     *
198  * Include: Range.h
     *
200  * Description: Gets the maximum range of the device
     *
202  * Arguments:
     *
204  * Returns: The max range in mm
     ********************************************************************/
206 extern int getMaxRange(void);

208 /*! ********************************************************************
     * Function: getMinRange()
210  *
     * Include: Range.h
212  *
     * Description: Gets the minimum range of the device
214  *
     * Arguments:
216  *
     * Returns: The min range in mm
218  ********************************************************************/
   extern int getMinRange(void);
220
   /*! ********************************************************************
222  * Function: getUsSampleRate()
     *
224  * Include: Range.h
     *
226  * Description: Gets the sampling rate of the ultrasonic sensor
     *
228  * Arguments:
     *
230  * Returns: The sampling rate (In Hz)
     ********************************************************************/
232 extern char getUsSampleRate();

234 /*! ********************************************************************
     * Function: setUsSampleRate(char sampleRate)
236  *
     * Include: Range.h
238  *
     * Description: Sets the ultrasonic sampling rate
240  *
```

```
 * Arguments: the sampling rate in Hz
 *
 * Returns: None
 ***************************************************************************/
extern void setUsSampleRate(char samples);

/*! ************************************************************************
 * Function: getNumSamples()
 *
 * Include: Range.h
 *
 * Description: Gets the number of samples per range estimate
 *
 * Arguments: None
 *
 * Returns: The number of samples
 ***************************************************************************/
extern char getNumSamples();

/*! ************************************************************************
 * Function: setNumSamples()
 *
 * Include: Range.h
 *
 * Description: sets the number of samples per range estimage
 *
 * Arguments: the number of samples
 *
 * Returns: None
 ***************************************************************************/
extern void setNumSamples(char sampleRate);

#define RANGE_H
#endif
```

../Code/Range.h

9.1.16 Range.c

```
/*! ************************************************************************
 * File:    Range.c
 * Author: Grant
 *
 * Description:
 * Contains all the functionality for the range module. All variables and settings
 * concerning the range module including the the max distance, timeouts etc are
 * private to this module. The interface functions allow all valid access to the module.
 *
 * Duties:
 *      -Interface to the IR and Ultrasonic sensors
 *      -Read the range from the IR and ultrasonic sensors
 *      -Fuse distances from the IR and ultrasonic sensors
 *      -Calibrate the IR and ultrasonic sensors
 *
 * Functions:
 *
 * Created on 15 September 2014, 11:27 AM
```

```
 ********************************************************************************/

#include "Common.h"
#include "Temp.h"
#include <delays.h>

//ONLY FOR RANGE TESTING PURPOSES
#include "Serial.h"
static void transRange(unsigned int us, unsigned int ir);
//

///Converts AD reading into IR range
#define IR_CONV(ad) ((unsigned long)135174 / (ad) - 28)

///Converts a time delay (in clock cycles) and a temperature into an ultrasonic distance
///Uses a linear appoximation of the speed of sound to temperature relation, which changes the
#ifdef MNML
#define ULTRA_CONV(tme, T) DIV_65536(tme * (unsigned long)(DIV_65536((unsigned long)519078 * T
#else
#define ULTRA_CONV(tme, T) DIV_65536(tme * (unsigned long)(DIV_65536((unsigned long)1297695 * 
#endif

//Hardware Related macros
#define INIT_PIN PORTCbits.RC3
#define INIT_TRIS TRISCbits.RC3
#define CCP1_INPT TRISCbits.RC2

//Flag for performing an ultrasonic measurment
volatile static char measuringUS = 0;

static int m_minRange = 300;
static int m_maxRange = 1000;

//Static variable to store the range
static unsigned int lastRange = 0;
static unsigned int lastUSRange = 0;    //These are used by the calibration function
static unsigned int lastIRRange = 0;

//Calibration offsets
static signed int calibration_offset_IR = 0;
static signed int calibration_offset_US = 0;

//Configuration variables
static char numSamples = 1;          //Number of ultrasonic samples
static unsigned char rateUS = 15;    //Ultrasonic sampling rate in Hz
static unsigned int rateIR = 200;    //IR sampling rate in Hz

static TargetState current_target_state;

//Private function prototypes:
static void beginUS(void);
static unsigned int rangeUS(unsigned char temp);
static unsigned int fuseRange(unsigned int us, unsigned int ir);


static volatile unsigned int ccp_value;
```

```
76 void configureRange(void);

78 unsigned int range(void);

80 /*! ************************************************************************
    * Function: configureAD(void)
82  *
    * Include: Range.h
84  *
    * Description: Configures the ADC,
86  * In ADCON1, we set right-justified mode, and select AN0 as the input channel.
    * In ADCON0, we set a sample rate of Fosc/8, select AN0, and enable the ADC.
88  * Arguments: None
    *
90  * Returns: None
    ************************************************************************/
92 void configureAD(void)
   {
94     int i = 0;
       TRISA = 0xFF;
96
       //Write the configuration values into the configuration registers
98     ADCON2bits.ADFM = 1; //Right justified
       ADCON1 = 0x8C;       // AN0, AN1,  AN2 analogue inputs
100    ADCON0 = 0x41;

102    //Arbitrary wait period to allow the ADC to initialise
       for (i = 0; i < 1000; i++);
104 }

106 /*! ************************************************************************
    * Function: configureRange(void)
108 *
    * Include: Range.h
110 *
    * Description: Configures the Range module
112 *
    * Arguments: None
114 *
    * Returns: None
116 ************************************************************************/
   void configureRange(void)
118 {
       unsigned char config;
120
       INIT_PIN = 0;
122
       //Enable global interrupts and interrupt priority
124    INT_SETUP()

126    readTemp();

128    //Make sure the AD is configured
       configureAD();
130
       CCP1_INPT = 1;
132    INIT_TRIS = 0;  //Make the INIT
```

```
134     //Open  Timer
        config = T1_16BIT_RW & T1_SOURCE_INT & T1_OSC1EN_OFF & T1_PS_1_1 & T1_SYNC_EXT_OFF &TIMER_
136     OpenTimer1(config);

138     config = CAPTURE_INT_ON & CAP_EVERY_RISE_EDGE;

140     //CloseCapture1, which will clear any interrupt flags etc
        CloseCapture1();
142
        //Open the input capture on compare1
144     OpenCapture1(config);
    }
146
    /*! ************************************************************************
148  * Function: beginUS(void)
     *
150  * Include: ultrasonic.h
     *
152  * Description: starts a scan on the ultrasonic sensor
     *
154  * Arguments: None
     *
156  * Returns: None
     ************************************************************************/
158 static void beginUS(void)
    {
160     CCPR1H = 0;
        CCPR1L = 0;
162     //Set the INIT_PIN high to begin ultrasonic 'read'
        INIT_PIN = 1;
164
        //Clear the timer, so the CCP input value is the delay
166     TMR1H = 0;
        TMR1L = 0;
168
        PIE1bits.CCP1IE = 1;
170
        //Set the measuring flag
172     measuringUS = 1;
    }
174
    /*! ************************************************************************
176  * Function: rangeUS(unsigned char temp)
     *
178  * Include: ultrasonic.h
     *
180  * Description: Returns the result of the ultrasonic read (zero if no target
     *              found). Will poll until measurement is complete.
182  *
     * Arguments: tempx2 - 2x the temperature in deg Celsius
184  *
     * Returns: Distance in mm (unsigned int)
186  ************************************************************************/
    static unsigned int rangeUS(unsigned char temp)
188 {
        unsigned int range;
```

```
190     unsigned long int t;
        unsigned long i = 0;
192     //Continue to poll while measurement is still in progress
        // (measuringUS);
194     for (i = 0; i < 50000 && measuringUS; i++);
        measuringUS = 0;
196
        #ifdef MNML
198     //if (CCPR1 < 0x1770) return 0;
        //if (CCPR1 < 0x5DC) return 0;
200     if (ccp_value < 0x100 || i >=50000) return 0;

202     //Perform calculation (ReadCapture in us, speed of sound in m/s->um)
        // um/1024 = ~mm
204
        t = DIV_4096((unsigned long int) ccp_value * (unsigned long int) 285) - 18;
206
        #else
208     if (CCPR1 < 5DC) return 0;

210     //Perform calculation (ReadCapture in us, speed of sound in m/s->um)
        // um/1024 = ~mm
212
        t = DIV_4096((unsigned long int) CCPR1 * (unsigned long int) 712) - 18;
214     #endif

216     range = (unsigned int) t;
    //    if (range > m_maxRange) range = 0;
218     return range;
    }
220
    /*! ***************************************************************
222  * Function: rangeISR(void)
     *
224  * Include: range.h
     *
226  * Description: Called when an range related interrupt is fired, acts
     *              as the service routine for the rangefinding module.
228  *
     * Arguments: None
230  *
     * Returns: None
232  ****************************************************************/
    void rangeISR(void)
234 {
        if (CCP1_INT && CCPR1 > 0x5DC)
236     //if (CCP1_INT)
        //if (CCP1_INT && CCPR1 > 0x300)
238     {   //Checks if the CCP2 module fired the interrupt
            measuringUS = 0;
240         INIT_PIN = 0;
            PIE1bits.CCP1IE = 0;
242         ccp_value = CCPR1;
        }
244     if (TMR1_INT)
        {
246         measuringUS = 0;
```

```
        CCPR1 = 0;
248         ccp_value = 0;
            INIT_PIN = 0;
250         PIR1bits.TMR1IF = 0;
            PIE1bits.TMR1IE = 0;
252     }
        CCP1_CLEAR;
254     measuringUS = 0;
}
256
/*! ************************************************************************
258  * Function: rangeUltrasonic(void)
     *
260  * Include:
     *
262  * Description: performs an ultrasonic range reading.
     * Pins:
264  *
     * Arguments: None
266  *
     * Returns: the average of the samples
268  *
     * todo remove this function?
270  ************************************************************************/
unsigned int rangeUltrasonic(void)
272 {
        unsigned int rng;
274
            configureRange();
276
            beginUS();
278
            rng = rangeUS(25);
280
            INIT_PIN = 0;
282         CloseCapture1();
            CloseTimer1();
284
            return rng;
286
}
288
/*! ************************************************************************
290  * Function: calibrateRange(unsigned int reference)
     *
292  * Include:
     *
294  * Description: Calibrates the range of both the IR and ultrasonic sensors
     *              based on a given range.
296  *
     * Arguments: reference - The distance in mm to calibrate the current
298  *                        measurements from
     *
300  * Returns: None
     ************************************************************************/
302 void calibrateRange(unsigned int reference)
    {
```

```
304     range();     //Sample the range

306     //Check if valid data was returned by the sensors before they perform a calibration
        if (current_target_state != NO_TARGET && current_target_state != CLOSE_RANGE)
308     {
            calibration_offset_US = reference - lastUSRange;
310     }
        if (current_target_state == GOOD_TRACK || current_target_state == CLOSE_RANGE)
312     {
            calibration_offset_IR = reference - lastIRRange;
314     }
    }
316
    /*! ***********************************************************************
318  * Function: range(void)
     *
320  * Include: Range.h
     *
322  * @brief Samples the range
     *
324  * Description: Takes a number of samples of the ultrasonic sensor at a specified
     *              rate. Continues to sample the IR sensor at a different rate while
326  *              sampling the ultrasonic. Then combines the ranges and sets the
     *              target state
328  *
     * Arguments: None
330  *
     * Returns: the range
332  ************************************************************************/
    unsigned int range(void)
334 {
    #define range_IR sumIR      //Come conenient name changes
336 #define range_US sumUS
        char i;
338     unsigned int k;
        unsigned int temp;
340     unsigned long int sumUS = 0;
        unsigned long int sumIR = 0;
342     int IR_samples = 0;
        unsigned char delayUS = 100 / rateUS;       //100Hz will give 1 delay increment of 10ms
344     unsigned char delayIR = 10000 / rateIR;     //10KHz will give 1 delay increment of 0.1ms

346     //Multiplex onto the IR sensor
    //    SetChanADC(ADC_IR_READ);
348     ADCON0 = ADC_IR_READ;

350     for (i = 0; i < numSamples; i++)
        {
352         configureRange();   //Still have to reconfigure each time???
            beginUS();
354
            //Continue sampling the IR while waiting for the ultrasonic
356         while (measuringUS)
            {
358             ADCON0bits.GO = 1;
                while (ADCON0bits.GO_NOT_DONE);
360             temp = ADRES;
```

```
            if (temp > 100) sumIR += IR_CONV(temp);
            else IR_samples--;
            //sumIR += IR_CONV(ADRES >> 6);
            IR_samples++;

            Delay100TCYx(delayIR);  //Delays in inrements of 100Tcy, which is 100 x 1us for 4M
        }
        //get range of ultrasonic reading
        sumUS += rangeUS(25);   ///Standard room temperature for now @todo Read in temperature
        Delay10KTCYx(delayUS);  //Delays in increments of 10KTcy, which is 10,000 * 1us for a
    }

    if (numSamples) sumUS = sumUS / numSamples;     //Calculate the average Ultrasonic range
    else sumUS = 0;

    //Average all IR samples taken, and convert to distance
    if (IR_samples) range_IR = sumIR / IR_samples;
    else sumIR = 0;

    // Save the range value for printing raw rang
    lastIRRange = range_IR;
    lastUSRange = range_US;

    // Eliminate out of range values
    if (range_IR > m_maxRange) range_IR = 0;
    if (range_US > m_maxRange) range_US = 0;
    if (range_IR < m_minRange) range_IR = 0;
    if (range_US < m_minRange) range_US = 0;

    return lastRange = fuseRange(range_US, range_IR);
#undef range_IR
#undef range_US
}

/*! ***************************************************************************
 * Function: rawRangeIR(void)
 *
 * Include: Range.h
 *
 * Description: Returns the last IR range reading
 *
 * Arguments: None
 *
 * Returns: the range in mm
 ****************************************************************************/
unsigned int rawRangeIR(void)
{
    return lastIRRange;
}

/*! ***************************************************************************
 * Function: rawRangeUS(void)
 *
 * Include: Range.h
 *
 * Description: Returns the last US range reading
 *
```

```
418  * Arguments: None
     *
420  * Returns: the range in mm
     ********************************************************************/
422 unsigned int rawRangeUS(void)
    {
424     return lastUSRange;
    }
426 /*! ********************************************************************
     * Function: getLastRange(void)
428  *
     * Include: Range.h
430  *
     * @brief Samples the range
432  *
     * Description: Returns the last range sample, so the data can be used without
434  *              re-sampling the range finding sensors
     *
436  * Arguments: None
     *
438  * Returns: the last Range sample data
     ********************************************************************/
440 unsigned int getLastRange(void)
    {
442     return lastRange;
    }
444
    /*! ********************************************************************
446  * Function: getTargetState(void)
     *
448  * Include: Range.h
     *
450  * Description: Returns the target state from the last range reading. E.g.
     *              Good track, or direction not quite correct as US returned,
452  *              but IR didn't and was within IR range etc.
     *
454  * Arguments: None
     *
456  * Returns: the target state
     ********************************************************************/
458 TargetState getTargetState(void)
    {
460     return current_target_state;
    }
462
    /*! ********************************************************************
464  * Function: readTargetState(void)
     *
466  * Include: Range.h
     *
468  * Description: Does the same thing as getTargetState, but actually performs
     *              a range() read
470  *
     * Arguments: None
472  *
     * Returns: the target state
474  ********************************************************************/
```

```
TargetState readTargetState(void)
{
    range();
    return current_target_state;
}

/*! ****************************************************************
 * Function: fuseRange(unsigned int us, unsigned int ir)
 *
 * Include: Range.h
 *
 * Description: Fuses the IR and Ultrasonic ranges, and sets the target state
 *
 * Arguments: us - the Ultrasonic range (mm)
 *            ir - the IR range (mm)
 *
 * Returns: the fused range
 *
 * Note: Also sets the current target state based on the reading from both
 *      the IR and ultrsonic sensors
 ******************************************************************/
static unsigned int fuseRange(unsigned int us, unsigned int ir)
{
    unsigned int range; //Store the range

    //Check which sensors have observed a target
    if (us && ir)
    {
        //Calibrate the sensors
        us += calibration_offset_US;
        ir += calibration_offset_IR;

        // CASE 1: Range: 150-450mm
        // Ignore US reading as it is inaccurate at these ranges
        if (us >= 150 && us <= 450)
        {
            range = ir;
            current_target_state = CLOSE_RANGE;
        }
        // CASE 2: Range: 1m - 1.5m
        // Don't trust the IR ranges much
        else if (us >= 1000 && ir >=1000)
        {
            /// @TODO Implement IR in here a little?
            range = us;
            current_target_state = OUT_OF_IR;
        }
        else
        {
            // CASE 3: Range: 450mm - 1m
            // Average the ultrasonic and IR ranges
            // TODO: Do we want to average these completely?
            range = DIV_2(us + ir);
            current_target_state = GOOD_TRACK;
        }

    }
```

```
532     // CASE 4: Range: 1.5m+
        // Rely on Ultrasound
534     else if (us)
        {
536         //Calibrate the ultrasonic range
            us += calibration_offset_US;
538
            range = us;
540
            //Check whether No IR is because out of IR range, or just bad direction
542         if (range > 1500) current_target_state = OUT_OF_IR;
            else current_target_state = BAD_DIR;
544     }
        else if (ir)
546     {
            //Calibrate the IR range
548         ir += calibration_offset_IR;

550         range = ir;
            current_target_state = CLOSE_RANGE;
552     }
        else
554     {
            /// @TODO: Report Error?
556         range = 0;
            current_target_state = NO_TARGET;
558     }
    }
560
    /*! ********************************************************************
562  * Function: getMaxRange()
     *
564  * Include: Range.h
     *
566  * Description: Gets the maximum range of the device
     *
568  * Arguments:
     *
570  * Returns: The max range in mm
     ********************************************************************/
572 int getMaxRange(void)
    {
574     return m_maxRange;
    }
576
    /*! ********************************************************************
578  * Function: getMinRange()
     *
580  * Include: Range.h
     *
582  * Description: Gets the minimum range of the device
     *
584  * Arguments:
     *
586  * Returns: The min range in mm
     ********************************************************************/
588 int getMinRange(void)
```

```
{
590     return m_minRange;
}
592
/*! ***************************************************************************
 * Function: setMaxRange()
 *
596 * Include: Range.h
 *
598 * Description: Sets the maximum range of the device
 *
600 * Arguments: range - The max range to set in mm
 *
602 * Returns: N/A
 ****************************************************************************/
604 void setMaxRange(int range)
{
606     m_maxRange = range;
}
608
/*! ***************************************************************************
610 * Function: setMinRange()
 *
612 * Include: Range.h
 *
614 * Description: Sets the minimum range of the device
 *
616 * Arguments: range - The max range to set in mm
 *
618 * Returns: N/A
 ****************************************************************************/
620 void setMinRange(int range)
{
622     m_minRange = range;
}
624
/*! ***************************************************************************
626 * Function: setNumSamples()
 *
628 * Include: Range.h
 *
630 * Description: sets the number of samples per range estimage
 *
632 * Arguments: the number of samples
 *
634 * Returns: None
 ****************************************************************************/
636 void setNumSamples(char samples)
{
638     numSamples = samples;
}
640
/*! ***************************************************************************
642 * Function: getNumSamples()
 *
644 * Include: Range.h
 *
```

```
646  * Description: Gets the number of samples per range estimate
     *
648  * Arguments: None
     *
650  * Returns: The number of samples
     ************************************************************************/
652 char getNumSamples()
    {
654     return numSamples;
    }
656
    /*! ************************************************************************
658  * Function: setUsSampleRate(char sampleRate)
     *
660  * Include: Range.h
     *
662  * Description: Sets the ultrasonic sampling rate
     *
664  * Arguments: the sampling rate in Hz
     *
666  * Returns: None
     ************************************************************************/
668 void setUsSampleRate(char sampleRate)
    {
670     rateUS = sampleRate;
    }
672
    /*! ************************************************************************
674  * Function: getUsSampleRate()
     *
676  * Include: Range.h
     *
678  * Description: Gets the sampling rate of the ultrasonic sensor
     *
680  * Arguments:
     *
682  * Returns: The sampling rate (In Hz)
     ************************************************************************/
684 char getUsSampleRate()
    {
686     return rateUS;
    }
```

../Code/Range.c

9.1.17 Serial.h

```
1  /*! ************************************************************************
    * File:   Serial.h
3   * Author: Grant
    *
5   * Description:
    * Contains the public interface for the serial module. This file contains
7   * all the external declarations, macros and global variables for using and
    * Interfacing with the serial module.
9   *
    * Created on 17 September 2014, 3:27 PM
```

```
11  ********************************************************************************/
13 //Ensure that there is only 1 inclusion of this file in the preprocessor execution
   #ifndef SERIAL_H
15
   //Serial interrupt flag
17 #define SERIAL_INT (TX_INT || RC_INT)
19 /*==============================================================================*/
   //External declarations of public functions
21 /*==============================================================================*/
23 /*! ********************************************************************
    * Function: configureSerial(void)
25  *
    * Include: Serial.h
27  *
    * Description: Configures the serial ready for communication
29  *
    * Arguments: None
31  *
    * Returns: None
33  **********************************************************************/
   extern void configureSerial(void);
35
   /*! ********************************************************************
37  * Function: serialISR(void)
    *
39  * Include: Serial.h
    *
41  * Description: Acts as the interrupt service routine for the serial module
    *
43  * Arguments: None
    *
45  * Returns: None
    **********************************************************************/
47 extern void serialISR(void);
49 /*==============================================================================*/
   //Transmit functions:
51 /*==============================================================================*/
53 /*! ********************************************************************
    * Function: transmit(char *string)
55  *
    * Include: Serial.h
57  *
    * Description: Begins transmitting the string over serial (interrupt driven)
59  *
    * Arguments: string - pointer to the beginning of the string to transmit
61  *
    * Returns: None
63  *
    * NOTE: Must be Null Terminated! Cannot receive a literal.
65  **********************************************************************/
   extern void transmit(char *string);
67
```

```
/*! ************************************************************************
 * Function: transmitComplete(void)
 *
 * Include: Serial.h
 *
 * Description: returns non-zero if the message has been completely transmited
 *              e.g. if the transmit buffer is empty
 *
 * Arguments: None
 *
 * Returns: non-zero if all messages have been transmitted
 *************************************************************************/
extern char transmitComplete(void);

/*! ************************************************************************
 * Function: transChar(char c)
 *
 * Include: Serial.h
 *
 * Description: Transmits a single character
 *
 * Arguments: c - character to transmit
 *
 * Returns: None
 *************************************************************************/
extern void transChar(char c);

/*=========================================================================*/
//Receive functions:
/*=========================================================================*/

/*! ************************************************************************
 * Function: receiveEmpty(void)
 *
 * Include: Serial.h
 *
 * Description: Indicates if the receive buffer is empty
 *
 * Arguments: None
 *
 * Returns: returns true if the recieve buffer is empty
 *************************************************************************/
extern char receiveEmpty(void);

/*! ************************************************************************
 * Function: receivePeek(void)
 *
 * Include: Serial.h
 *
 * Description: Returns the next character in the receive buffer without
 *              removing it from the buffer
 *
 * Arguments: None
 *
 * Returns: The next received character
 *************************************************************************/
extern char receivePeek(void);
```

```
125
126 /*! ************************************************************************
127  * Function: receivePop(void)
128  *
129  * Include: Serial.h
130  *
131  * Description: Pops the next received character from the received buffer
132  *
133  * Arguments: None
134  *
135  * Returns: The next character from the receive buffer
136  ************************************************************************/
137 extern char receivePop(void);
138
139 /*! ************************************************************************
140  * Function: receiveCR(void)
141  *
142  * Include: Serial.h
143  *
144  * Description: Indicates whether a Carriage Return has been received
145  *
146  * Arguments: None
147  *
148  * Returns: non-zero if CR has been received, zero otherwise
149  ************************************************************************/
150 extern char receiveCR(void);
151
152 /*! ************************************************************************
153  * Function: receiveEsc(void)
154  *
155  * Include:
156  *
157  * Description: Indicates whether an Escape character has been received
158  *
159  * Arguments: None
160  *
161  * Returns: non-zero if the Esc has been received, zero otherwise
162  ************************************************************************/
163 extern char receiveEsc(void);
164
165 /*! ************************************************************************
166  * Function: readString(char *string)
167  *
168  * Include: Serial.h
169  *
170  * Description: Writes all received data up to a carriage return into given
171  *              location.
172  *
173  * Arguments: string - Pointer to location to store received data
174  *
175  * Returns: Received data including the carriage return
176  *
177  * Remarks: Make sure that you reserve at least BUFFERLENGTH elements at the
178  *          location pointed to by string before calling this function.
179  ************************************************************************/
180 extern void readString(char *string);
181
```

```
/*! ***************************************************************
 * Function: popEsc(void)
 *
 * Include:
 *
 * Description: Processes the Esc command and removes any input before the
 *              Esc command.
 *
 * Arguments: None
 *
 * Returns: Non-zero if received escape character
 ****************************************************************/
extern void popEsc(void);

/*! ***************************************************************
 * Function: clearReceive(void)
 *
 * Include: Serial.h
 *
 * Description: Clears the receive buffer
 *
 * Arguments: None
 *
 * Returns: None
 ****************************************************************/
extern void clearReceive(void);

#define SERIAL_H
#endif
```

../Code/Serial.h

9.1.18 Serial.c

```
/*! ***************************************************************************
 * File:   newmain.c
 * Author: Grant
 *
 * Description:
 * Contains the functionality for the Serial module. All variables and settings
 * concerning the serial module, such as the receive and transmit circular buffers
 * are private to this module. The interface functions allow all valid access to the module.
 *
 * Duties:
 *      -Stores an received characters in the received buffer
 *      -Stores any characters to be transmitted
 *      -Transmits anything in transmit buffer via interrupts
 *      -Accessor functions for using and querying buffers
 *
 * Functions:
 *
 *
 * Created on 7 September 2014, 4:12 PM
 ****************************************************************************/

#include "Common.h"
#include <usart.h>
```

```
#include "CircularBuffers.h"

//Interrupt macros
#define TX_INT_CLEAR() PIR1bits.TXIF = 0
#define RC_INT_CLEAR() PIR1bits.RCIF = 0
#define TX_INT_ENABLE() TX_INT_CLEAR(); PIE1bits.TXIE = 1
#define RC_INT_ENABLE() RC_INT_CLEAR(); PIE1bits.RCIE = 1
#define TX_INT_DISABLE() PIE1bits.TXIE = 0
#define RC_INT_DISABLE() PIE1bits.RCIE = 0

//ASCII definitions
#define CR 0x0D
#define NL 0x0A
#define ESC 0x1B
#define TAB 0x09
#define BS 0x07

//Local Function Prototypes
static volatile circularBuffer receive_buffer;
static volatile circularBuffer transmit_buffer;
static volatile char carriageReturn = 0;
static volatile char escPressed = 0;

/*! ************************************************************************
 * Function: configureSerial(void)
 *
 * Include: Serial.h
 *
 * Description: Configures the serial ready for communication
 *
 * Arguments: None
 *
 * Returns: None
 ************************************************************************/
void configureSerial(void)
{
    INT_SETUP();

    //Initialise the serial buffers
    init(transmit_buffer);
    init(receive_buffer);
    carriageReturn = 0;
    escPressed = 0;

    //Open the USART module
#ifdef MNML
    OpenUSART(USART_TX_INT_ON & USART_RX_INT_ON & USART_BRGH_HIGH & USART_EIGHT_BIT & USART_ASY
#else
    OpenUSART(USART_TX_INT_ON & USART_RX_INT_ON & USART_BRGH_HIGH & USART_EIGHT_BIT & USART_ASY
#endif

}

/*! ************************************************************************
 * Function: transmit(char *string)
 *
 * Include: Serial.h
```

```
 *
 * Description: Begins transmitting the string over serial (interrupt driven)
 *
 * Arguments: string - pointer to the beginning of the string to transmit
 *
 * Returns: None
 *
 * NOTE: Must be Null Terminated! Cannot receive a literal.
 ************************************************************************/
void transmit(char *string)
{
    //Push the string onto the transmit buffer
    for (; *string; string++)
    {
        push(*string, transmit_buffer);
    }

    //Return if there is nothing to transmit
    if (empty(transmit_buffer)) return;

    //Enable TX interrupts
    TX_INT_ENABLE();
}

/*! ***********************************************************************
 * Function: transChar(char c)
 *
 * Include: Serial.h
 *
 * Description: Transmits a single character
 *
 * Arguments: c - character to transmit
 *
 * Returns: None
 ************************************************************************/
void transChar(char c)
{
    //Push character onto the transmit buffer
    push(c, transmit_buffer);

    //Enable TX interrupts
    TX_INT_ENABLE();
}

/*! ***********************************************************************
 * Function: receiveEmpty(void)
 *
 * Include: Serial.h
 *
 * Description: Indicates if the receive buffer is empty
 *
 * Arguments: None
 *
 * Returns: returns true if the recieve buffer is empty
 ************************************************************************/
char receiveEmpty(void)
{
```

```
138     return empty(receive_buffer);
}
140
/*! ************************************************************************
142  * Function: receivePeek(void)
 *
144  * Include: Serial.h
 *
146  * Description: Returns the next character in the receive buffer without
 *              removing it from the buffer
148  *
 * Arguments: None
150  *
 * Returns: The next received character
152  ************************************************************************/
char receivePeek(void)
154 {
    return peek(receive_buffer);
156 }

158 /*! ************************************************************************
 * Function: receivePop(void)
160  *
 * Include: Serial.h
162  *
 * Description: Pops the next received character from the received buffer
164  *
 * Arguments: None
166  *
 * Returns: The next character from the receive buffer
168  ************************************************************************/
char receivePop(void)
170 {
    char c = pop(receive_buffer);
172     return c;
}
174
/*! ************************************************************************
176  * Function: receiveCR(void)
 *
178  * Include: Serial.h
 *
180  * Description: Indicates whether a Carriage Return has been received
 *
182  * Arguments: None
 *
184  * Returns: non-zero if CR has been received, zero otherwise
 ************************************************************************/
186 char receiveCR(void)
{
188     return carriageReturn;
}
190
/*! ************************************************************************
192  * Function: receiveEsc(void)
 *
194  * Include:
```

```
 *
 * Description: Indicates whether an Escape character has been received
 *
 * Arguments: None
 *
 * Returns: non-zero if the Esc has been received, zero otherwise
 ************************************************************************/
char receiveEsc(void)
{
    return escPressed;
}

/*! ************************************************************************
 * Function: popEsc(void)
 *
 * Include:
 *
 * Description: Processes the Esc command and removes any input before the
 *              Esc command.
 *
 * Arguments: None
 *
 * Returns: Non-zero if received escape character
 ************************************************************************/
void popEsc(void)
{
    while (!empty(receive_buffer))
    {
        pop(receive_buffer);
    }
    escPressed--;
    return;
}

/*! ************************************************************************
 * Function: readString(char *string)
 *
 * Include: Serial.h
 *
 * Description: Writes all received data up to a carriage return into given
 *              location.
 *
 * Arguments: string - Pointer to location to store received data
 *
 * Returns: Received data including the carriage return
 *
 * Remarks: Make sure that you reserve at least BUFFERLENGTH elements at the
 *          location pointed to by string before calling this function.
 ************************************************************************/
void readString(char *string)
{
    char c, e;
    e = empty(receive_buffer);
    c = pop(receive_buffer);
    while (c != CR && !e)
    {
        *(string++) = c;
```

```
252         e = empty(receive_buffer);
            c = pop(receive_buffer);
254     }
        *string = c;
256     carriageReturn--;
}
258
/*! ***************************************************************************
260  * Function: transmitComplete(void)
     *
262  * Include: Serial.h
     *
264  * Description: returns non-zero if the message has been completely transmited
     *              e.g. if the transmit buffer is empty
266  *
     * Arguments: None
268  *
     * Returns: non-zero if all messages have been transmitted
270  ****************************************************************************/
char transmitComplete(void)
272 {
        return empty(transmit_buffer);
274 }

276 /*! ***************************************************************************
     * Function: serialISR(void)
278  *
     * Include: Serial.h
280  *
     * Description: Acts as the interrupt service routine for the serial module
282  *
     * Arguments: None
284  *
     * Returns: None
286  ****************************************************************************/
void serialISR(void)
288 {
        unsigned char data;
290     char last;

292     //Check which serial interrupt instance was thrown
        if (TX_INT)
294     {
            //Return if there is nothing in the transmit buffer
296         if (empty(transmit_buffer))
            {
298             TX_INT_DISABLE();
                return;
300         }

302         data = pop(transmit_buffer);
            WriteUSART(data);
304
            //Clear interrupt flag
306         TX_INT_CLEAR();
        }
308     else if (RC_INT)
```

```
    {
310         data = ReadUSART();
            last = peek(receive_buffer);
312
            if (data == ESC) escPressed++;
314         else
            {
316             //Allows the user to remove/change transmitted data
                if (data == BS && last != CR && last != ESC && last != NL )
318             {
                    pop(receive_buffer);
320             }
                else
322             {
                    //push the received data onto the received buffer
324                 push(data, receive_buffer);
                    if (data == CR) carriageReturn++;
326
                    //Clear interrupt flag
328                 RC_INT_CLEAR();
                }
330         }
    }
332 }

334 /*! ************************************************************************
 * Function: clearReceive(void)
336  *
 * Include: Serial.h
338  *
 * Description: Clears the receive buffer
340  *
 * Arguments: None
342  *
 * Returns: None
344  ************************************************************************/
void clearReceive(void)
346 {
    init(receive_buffer);
348 }
```

../Code/Serial.c

9.1.19 Temp.h

```
/*! ************************************************************************
2  * File:   Temp.h
 * Author: Grant
4  *
 * Description:
6  * Contains all the functionality and variables for the temp module. All unnecessary
 * functions and variables should be shielded from the external program, and interfaced
8  * with accessor and mutator functions
 *
10 * Contains:
 *      -Configure function
12 *      -Read temperature functions
```

```
 *      -Get (last) temperature (read) function
 *      -Calibrate Temperature function
 *
 * Created on 17 September 2014, 2:16 PM
 ******************************************************************************/

//Ensure that there is only 1 inclusion of this file in the preprocessor execution
#ifndef TEMP_H

/*=============================================================================*/
//External public access function prototypes
/*=============================================================================*/

/*! ***************************************************************************
 * Function: configureTemp(void)
 *
 * Include: Temp
 *
 * Description: Configures the temperature module for use
 *
 * Arguments: None
 *
 * Returns: None
 ******************************************************************************/
extern void configureTemp(void);

/*! ***************************************************************************
 * Function: rawTemp(void)
 *
 * Include: Temp.h
 *
 * Description: Returns the raw (uncalibrated temperature)
 *
 * Arguments: None
 *
 * Returns: Temp (in deg celsius) as an unsigned char
 ******************************************************************************/
extern unsigned char rawTemp(void);

/*! ***************************************************************************
 * Function: readTemp(void)
 *
 * Include: Temp.h
 *
 * Description: Reads the temperature from the TEMP sensor
 *
 * Arguments: None
 *
 * Returns: Temp (in deg celsius) as an unsigned char
 ******************************************************************************/
extern unsigned char readTemp(void);

/*! ***************************************************************************
 * Function: readTempx2(void)
 *
 * Include: Temp.h
 *
```

```
 * Description: Reads the temperature from the Temp sensor
 *
 * Arguments: None
 *
 * Returns: temp x 2 (in deg celsius) as an unsigned char
 *
 * @todo Test and debug this function
 *****************************************************************************/
extern unsigned char readTempx2(void);

/*! ***************************************************************************
 * Function: calibrateTemp(unsigned char reference)
 *
 * Include: Temp.h
 *
 * Description: calibrates the temperature sensor by updating the calibration
 *              offset variable
 *
 * Arguments: reference - Reference temperature in deg C
 *
 * Returns: None
 *
 * Note: This function does not perform a temperature read, but uses the last
 *       value. This is because the readTemp function automatically calibrate.
 *****************************************************************************/
extern void calibrateTemp(unsigned char reference);

/*! ***************************************************************************
 * Function: calibrationTemp(unsigned char reference)
 *
 * Include: Temp.h
 *
 * Description: calibrates the temperature sensor by updating the calibration
 *              offset variable
 *
 * Arguments: reference - Reference temperature in deg C
 *
 * Returns: None
 *****************************************************************************/
extern unsigned char getTemp(void);

#define TEMP_H
#endif
```

../Code/Temp.h

9.1.20 Temp.c

```
/*! ***************************************************************************
 * File:   Temp.c
 * Author: Grant
 *
 * Description:
 * Contains all the functionality for the Temp module.
 *
 * Duties:
 *      -Samples the temperature sensor
```

```
 *      -Stores the last temperature value
 *      -Calibrates the temperature sensor
 *
 * Functions:
 *
 *
 * Created on 7 September 2014, 4:12 PM
 ******************************************************************************/

#include "Common.h"
#include "Range.h" //Access to the configureAD() function

//Static variables
static signed char calibration_offset = 0;
static unsigned char lastTempx2;

/*! ***************************************************************************
 * Function: configureTemp(void)
 *
 * Include: Temp
 *
 * Description: Configures the temperature module for use
 *
 * Arguments: None
 *
 * Returns: None
 *****************************************************************************/
void configureTemp(void)
{
    configureAD();
}

/*! ***************************************************************************
 * Function: readTempx2(void)
 *
 * Include: Temp.h
 *
 * Description: Reads the temperature from the Temp sensor
 *
 * Arguments: None
 *
 * Returns: temp x 2 (in deg celsius) as an unsigned char
 *
 * @todo Test and debug this function
 *****************************************************************************/
unsigned char readTempx2(void)
{
    unsigned char tempx2;

    char mod;
    int tempRaw;
    int tempmV;
    char offset = -20; //0mV at 2 degrees??

    //Sets the ADC channel to read the temperature to channel 1
    ADCON0 = ADC_TEMP_READ;

```

```
    ADCON0bits.GO = 1;
68  while (ADCON0bits.GO_NOT_DONE == 1);
    tempRaw = ADRES;
70  tempmV = tempRaw * 4.88 + offset; //each incrememnt of ADRES is 4.88mV

72  // Store double to keep accuracy
    tempmV = 2*tempmV;
74  // Round last digit
    mod = tempmV % 10;
76  if (mod >= 5) tempmV = tempmV + (10 - mod);
    lastTempx2 = tempx2 =tempmV/10;
78
    return tempx2 + calibration_offset;
80 }

82 /*! ***************************************************************************
   * Function: readTemp(void)
84 *
   * Include: Temp.h
86 *
   * Description: Reads the temperature from the TEMP sensor
88 *
   * Arguments: None
90 *
   * Returns: Temp (in deg celsius) as an unsigned char
92 ****************************************************************************/
   unsigned char readTemp(void)
94 {
    unsigned char temp;
96
    //Read the temperature from the x2 function
98  temp = readTempx2();

100 //Divide the temp by two and return the result
    return DIV_2(temp);
102 }

104 /*! ***************************************************************************
   * Function: rawTemp(void)
106 *
   * Include: Temp.h
108 *
   * Description: Returns the raw (uncalibrated temperature)
110 *
   * Arguments: None
112 *
   * Returns: Temp (in deg celsius) as an unsigned char
114 ****************************************************************************/
   unsigned char rawTemp(void)
116 {
    return DIV_2(lastTempx2);
118 }

120 /*! ***************************************************************************
   * Function: calibrateTemp(unsigned char reference)
122 *
   * Include: Temp.h
```

```
124  *
     * Description: calibrates the temperature sensor by updating the calibration
126  *              offset variable
     *
128  * Arguments: reference - Reference temperature in deg C
     *
130  * Returns: None
     *
132  * Note: This function does not perform a temperature read, but uses the last
     *       value. This is because the readTemp function automatically calibrate.
134  ********************************************************************/
     void calibrateTemp(unsigned char reference)
136  {
         calibration_offset = 2 * (reference - DIV_2(lastTempx2));
138  }

140  /*! ********************************************************************
     * Function: calibrationTemp(unsigned char reference)
142  *
     * Include: Temp.h
144  *
     * Description: calibrates the temperature sensor by updating the calibration
146  *              offset variable
     *
148  * Arguments: reference - Reference temperature in deg C
     *
150  * Returns: None
     ********************************************************************/
152  unsigned char getTemp(void)
     {
154      return DIV_2(lastTempx2);
     }
```

../Code/Temp.c

9.1.21 Tracking.h

```
1  /*! ********************************************************************
   * File:   Tracking.h
3  * Author: Grant
   *
5  * Description:
   * Public interface function for the Tracking Module. This module contains the
7  * functionality and algorithms used to search for, and track the target. This
   * module makes use of PanTilt, Range and Temp modules, and configures them
9  * automatically from a call to configureTrack().
   *
11 * Contains:
   *      -Configure tracking
13 *      -Searching algorithm
   *      -Track algorithm
15 *
   * Created on 15 September 2014, 1:41 PM
17 ********************************************************************/

19 //Ensure that there is only 1 inclusion of this file in the preprocessor execution
   #ifndef TRACK_H
```

```
21
   //#define TRACK_INT (CCP2_INT)
23 #define TRACK_INT 0

25 /*=========================================================================*/
   //External declarations for the public access functions
27 /*=========================================================================*/

29 /*! ************************************************************************
    * Function: configureTracking(void)
31  *
    * Include: Tracking.h
33  *
    * Description: Configures the System to begin tracking/searching
35  *
    * Arguments: None
37  *
    * Returns: None
39  *************************************************************************/
   extern void configureTracking(void);
41
   /*! ************************************************************************
43  * Function: search(void)
    *
45  * Include: Tracking.h
    *
47  * Description: Performs an incrimental change in position. Local variables
    *              store previous movements to create a raster-like search pattern.
49  *              Then samples the range sensors and determines the next system
    *              state.
51  *
    * Arguments: state - a pointer to the current system state
53  *
    * Returns: None
55  *************************************************************************/
   extern void search(systemState *state);
57
   /*! ************************************************************************
59  * Function: track(void)
    *
61  * Include: Tracking.h
    *
63  * Description: Takes a number of samples at the previous target location, and
    *              several descrete locations around the target location and takes
65  *              a weighted average based on the sampled signal return
    *
67  * Arguments: state - A pointer to the current system state
    *
69  * Returns: TargetData - The current target information (Azimuth, inclination
    *                       and range to the target). This information is then used
71  *                       for the Display in the User interface module.
    *************************************************************************/
73 extern TrackingData track(systemState *state);

75 /*! ************************************************************************
    * Function: trackingISR(void)
77  *
```

```
 * Include: Tracking.h
 *
 * Description: Acts as the Interrupt Service Routine for the Tracking module
 *              Not currently implemented
 *
 * Arguments: None
 *
 * Returns: None
 ******************************************************************************/
extern void trackingISR(void);

#define TRACK_H
#endif
```

../Code/Tracking.h

9.1.22 Tracking.c

```
/*! ***************************************************************************
 * File:   Tracking.c
 * Author: Grant
 *
 * Description:
 * Contains all the functionality for the Tracking Module. This module contains the
 * functionality and algorithms used to search for, and track the target. This
 * module makes use of PanTilt, Range and Temp modules, and configures them
 * automatically from a call to configureTrack().
 *
 * Duties:
 *      -Control and coordinate Pan Tilt and range sensors
 *      -Implement searching and tracking algorithms
 *      -Predict next position of target???
 *
 * Functions:
 *
 *
 * Created on 15 September 2014, 1:42 PM
 *****************************************************************************/

#include "Common.h"
#include "Range.h"
#include "PanTilt.h"

#define diff 8
#define TARGET_RAD 30  //The diameter of the target in mm (slightly larger)
#include <delays.h>
#include <timers.h>

/*! ***************************************************************************
 * typedef of targetStateData struct
 *
 * @brief Stores data about the last target track
 *
 * Description:
 * Stores the number of samples of each type sampled in the previous target track
 *
 * Elements:
```

```
 *      -bad_dirs - stores the number of BAD_DIR's sampled - only 3 bits
 *      -out_of_irs - Stores the number of OUT_OF_IR's sampled - only 3 bits
 *      -good_tracks - Stores the number of GOOD_TRACK's sampled - only 3 bits
 ***************************************************************************/
typedef struct
{
    unsigned bad_dirs : 2;
    unsigned out_of_irs : 2;
    unsigned good_tracks : 2;
    unsigned centre : 2;
} TargetStateData;

static char newAngle(char angle, TargetStateData target_data);

/*! ***************************************************************************
 * Function: configureTracking(void)
 *
 * Include: Tracking.h
 *
 * Description: Configures the System to begin tracking/searching
 *
 * Arguments: None
 *
 * Returns: None
 ***************************************************************************/
void configureTracking(void)
{
    configureBase();
    configureRange();
//
//    //Set up TMR2 for prediction purposes
//    CloseTimer0();
//    OpenTimer0(T0_16BIT & T0_SOURCE_INT & T0_PS_1_256);
}

/*! ***************************************************************************
 * Function: search(void)
 *
 * Include: Tracking.h
 *
 * Description: Performs an incrimental change in position. Local variables
 *              store previous movements to create a raster-like search pattern.
 *              Then samples the range sensors and determines the next system
 *              state.
 *
 * Arguments: state - a pointer to the current system state
 *
 * Returns: None
 ***************************************************************************/
void search(systemState *state)
{
    unsigned int j;
    //Vertical and laterial incremental movements
    static Direction lateral = {diff, 0};
    static Direction vertical = {0, diff};
    static TargetState previousState;
    TargetState currentState;
```

```
    Direction dir;

    dir = getDir();

    for (j = 0; j < 5000; j++);
    //If max azimuth range, increment vertical and change azimuth direction
    if (dir.azimuth > getMaxAzimuthAngle() || dir.azimuth < getMinAzimuthAngle())
    {
        increment(vertical);                //Move up (or down) vertically 1 degree
        if (dir.azimuth < 0 && dir.azimuth < 0) lateral.azimuth = diff; //Move in the opposite
        if (dir.azimuth > 0 && dir.azimuth > 0) lateral.azimuth = -diff;
        increment(lateral);
    }
    //If max inclination range, change vertical direction and increment vertical
    else if (dir.elevation > getMaxElevationAngle() || dir.elevation < getMinElevationAngle())
    {
        if (dir.elevation < 0 && dir.elevation < 0) vertical.elevation = diff; //Move in the o
        if (dir.elevation > 0 && dir.elevation > 0) vertical.elevation = -diff;
        increment(vertical);
    }
    //Else just move in azimuth
    else
    {
        increment(lateral);
    }

    previousState = currentState;
    currentState = readTargetState();

    if (currentState == GOOD_TRACK || currentState == BAD_DIR || currentState == CLOSE_RANGE)
    {
        NEXT_STATE_PTR(TRCK, state);
    }
}

/*! ********************************************************************
 * Function: trackingISR(void)
 *
 * Include: Tracking.h
 *
 * Description: Acts as the Interrupt Service Routine for the Tracking module
 *              Not currently implemented
 *
 * Arguments: None
 *
 * Returns: None
 **********************************************************************/
void trackingISR(void)
{

}

/*! ********************************************************************
 * Function: track(void)
 *
 * Include: Tracking.h
 *
```

```
154  * Description: Takes a number of samples at the previous target location, and
     *              several descrete locations around the target location and takes
156  *              a weighted average based on the sampled signal return
     *
158  * Arguments: state - A pointer to the current system state
     *
160  * Returns: TargetData - The current target information (Azimuth, inclination
     *                       and range to the target). This information is then used
162  *                       for the Display in the User interface module.
     ***********************************************************************/
164 TrackingData track(systemState *state)
    {
166 #define sampleTargetState weight
        char i, count = 0;
168     char weight;
        unsigned int j;
170     unsigned int rng;
        Direction centre;
172     signed long int inclination = 0;
        signed long int azimuth = 0;
174     TrackingData result;
        Direction dir;
176     Direction inc[] = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}}; //Incremental change in direction NO
        char angle;
178     TargetStateData target_data;

180     rng = range();
        if (rng)
182     {
            angle = 13 * (unsigned int)300 / rng + 5;
184     }
        else
186     {
            angle = 10;
188     }

190     centre = getDir();

192     for (i = 0; i < 4; i++)
        {
194         //Calc new direction
            dir.azimuth = centre.azimuth + inc[i].azimuth * angle;
196         dir.elevation = centre.elevation + inc[i].elevation * angle;

198         //Move to new direction
            move(dir);
200
            for (j = 0; j < 4000; j++);
202
            //Get target state (stored in weight temporarily)
204         sampleTargetState = readTargetState();

206         //Calculate weighting of the sample (1 is the worst, the higher the better)
            weight = (sampleTargetState == GOOD_TRACK) * 5 + (sampleTargetState == BAD_DIR) + (sam
208
            count += weight;
210         azimuth += (int)weight * dir.azimuth;
```

```
            inclination += (int)weight * dir.elevation;
        }

        //Go back to searching if there is not a good track, else stay in track
        if (count == 0)
        {
            NEXT_STATE_PTR(SRCH, state);
        }
        else
        {
            //Calculate weighted average of the results
            centre.azimuth = azimuth / count;
            centre.elevation = inclination / count;

            move(centre);

            for (j = 0; j < 4000; j++);

            result.range = range();
            result.azimuth = centre.azimuth;
            result.elevation = centre.elevation;
        }

        return result;
    #undef sampleTargetState
    }

    /*! ***********************************************************************
     * Function: prediction(Direction current)
     *
     * Include: Local to Tracking.c
     *
     * Description: Predicts where the object is likely to be found based on previous
     *              movement and prediction algorithms. Not currently implemented
     *
     * Arguments: current - The current position of the target
     *
     * Returns: Direction - The Predicted likely location
     ************************************************************************/
    static Direction prediction(Direction current)
    {
    #define TIMER   TMR2        //General purpose timer
        static Direction previous = {0, 0};
        static unsigned int prev_time;
        static Direction next_predict;
        return next_predict;
    }

    /*! ***********************************************************************
     * Function: newAngle(void)
     *
     * Include: Local to Tracking.c
     *
     * Description: Calculates the new angle offset - So when the target position
     *              is better known the sampling locations are more file. Not currently implemente
     *
     * Arguments: angle - The previous angle
```

```
 *
 *            target_data - The data from the last target sample
 *
 * Returns: Angle - The new angle to offset the direction
 ************************************************************************/
static char newAngle(char angle, TargetStateData target_data)
{
    //Target field too narrow, and a little out, so increase range to get some direction data

    if (target_data.centre == 0)
    {
        angle += (target_data.good_tracks > 2) - (target_data.good_tracks == 1 && (target_data
    }

    if (angle < 1) return 1;
    else return angle;
}
```

../Code/Tracking.c

9.1.23 User_Interface.h

```
/*************************************************************************
 * File:   User_Interface.h
 * Author: Grant
 *
 * Description: Contains the public interface for the User_Interface module
 *
 * Created on 15 September 2014, 1:20 PM
 ************************************************************************/

//#include "Common.h"

//Ensure that there is only 1 inclusion of this file in the preprocessor execution
#ifndef USER_H

//User interface related interrupts
#define USER_INT (RB_INT || INT0_INT || INT1_INT )
#define CONFIRM_CHAR 0x0D
#define BACK_CHAR 0x1B

/*=======================================================================*/
//Public function external declarations
/*=======================================================================*/

/*! ********************************************************************
 * Function: display(TrackingData data)
 *
 * Include: User_Interface.h
 *
 * Description: displays the current tracking information
 *
 * Arguments: data - Struct which contains all known data concerning the
 *                   The position of the target. A Range of 0 indicates
 *                   that no target was observed.
 *
 * Returns: None
```

```
36  ***************************************************************************/
    extern void display(TrackingData data);
38
    /*! ***************************************************************************
40   * Function: userISR(void)
     *
42   * Include: User_Interface.h
     *
44   * Description: Acts as the ISR for the User_interface module
     *
46   * Arguments: None
     *
48   * Returns: None
     ***************************************************************************/
50  extern void userISR(void);

52  /*! ***************************************************************************
     * Function: configUSER(void)
54   *
     * Include: User_Interface.h
56   *
     * Description: Configures the user interface for use
58   *
     * Arguments: None
60   *
     * Returns: None
62   ***************************************************************************/
    extern void configUSER(void);
64

66  /*! ***************************************************************************
     * Function: userEmpty(void)
68   *
     * Include: User_Interface.h
70   *
     * Description: returns non-zero if the user interface buffer is empty (i.e.
72   *              no user input has been detected)
     *
74   * Arguments: None
     *
76   * Returns: if the user input buffer is empty
     ***************************************************************************/
78  extern char userEmpty(void);

80  /*! ***************************************************************************
     * Function: userPop(void)
82   *
     * Include: User_Interface.h
84   *
     * Description: pops a character from the user interface receive buffer
86   *
     * Arguments: None
88   *
     * Returns: a character poped from the user Interface buffer
90   ***************************************************************************/
    extern char userPop(void);
92
```

```
/*! ***************************************************************
 * Function: userPeek(void)
 *
 * Include: User_Interface.h
 *
 * Description: returns a character in the user_Interface buffer without
 *              removing it from the buffer
 *
 * Arguments: None
 *
 * Returns: a character in the user interface buffer
 ****************************************************************/
extern char userPeek(void);

/*! ***************************************************************
 * Function: readDial(void)
 *
 * Include:
 *
 * Description: Read the position of the dial by splitting the value
 *              returned from the ADC into even discrete steps based on the
 *              given argument. For instance, an input of 7 will return
 *              a value from 0 to 7 based on the value found from the potentiometer
 *
 * Arguments: int max - The number of maximum state to split values into
 *
 * Returns: The dial position scaled by the maximum number
 ****************************************************************/
extern unsigned int readDial(unsigned int max);

#define USER_H
#endif
```

../Code/User_Interface.h

9.1.24 User_Interface.c

```
/*! ***************************************************************************
 * File:   User_Interface.c
 * Author: Grant
 *
 * Description:
 * Contains all the functionality for the User_Interface module.
 * Works the same way as the serial module, but for the user interface.
 *
 * Duties:
 *      -Stores any local user input in a receive buffer
 *      -Sends display data to the LCD
 *
 * Created on 15 September 2014, 1:21 PM
 ****************************************************************************/

#include "Common.h"
#include "CircularBuffers.h"
#include "Range.h"
#include "User_Interface.h"
#include "Serial.h"
```

```
22 #define CONFIRM_PRESS INTCONbits.INT0IF
   #define BACK_PRESS INTCON3bits.INT1F
24 //#define BACK_PRESS RB_INT

26 #define ADC_MAX 1023  // Max number allocated by a 10 bit number

28 //Store anything entered by the user
   circularBuffer receive;
30
   /*! ************************************************************************
32  * Function: display(TrackingData data)
    *
34  * Include: User_Interface.h
    *
36  * Description: displays the current tracking information
    *
38  * Arguments: data - Struct which contains all known data concerning the
    *                   The position of the target. A Range of 0 indicates
40  *                   that no target was observed.
    *
42  * Returns: None
    ************************************************************************/
44 void display(TrackingData data)
   {
46
   }
48
   /*! ************************************************************************
50  * Function: userISR(void)
    *
52  * Include: User_Interface.h
    *
54  * Description: Acts as the ISR for the User_interface module
    *
56  * Arguments: None
    *
58  * Returns: None
    ************************************************************************/
60 void userISR(void)
   {
62     if (CONFIRM_PRESS)
       {
64         push(CONFIRM_CHAR, receive);
           CONFIRM_PRESS = 0;
66     }
       if (BACK_PRESS)
68     {
               push(BACK_CHAR, receive);
70             BACK_PRESS = 0;
       }
72     return;
   }
74
   /*! ************************************************************************
76  * Function: configUSER(void)
    *
```

```
78  * Include: User_Interface.h
    *
80  * Description: Configures the user interface for use
    *
82  * Arguments: None
    *
84  * Returns: None
    ********************************************************************/
86 void configUSER(void)
   {
88     //Initialises the receive buffer
       init(receive);
90
       configureAD();
92
   //    // Set RB 0 and 1 to input
94     TRISB = 0;
       PORTB = 0;
96     TRISB = 0xFF;

98     INTCONbits.GIE_GIEH = 1;     /*enable high priority interrupts*/
       INTCONbits.PEIE_GIEL = 1;    /*enable low priority interrupts*/
100
       // Enable INT0 interrupt
102    INTCON2bits.INTEDG0 = 1;     // Interrupt on rising edge
       INTCONbits.INT0IE = 1;
104
       // Enable INT1 interrupt
106    INTCON3bits.INT1IP = 1;
       INTCON2bits.INTEDG1 = 1;     // Interrupt on rising edge
108    INTCON3bits.INT1E = 1;
   }
110
   /*! ********************************************************************
112 * Function: readDial(void)
    *
114 * Include:
    *
116 * Description: Read the position of the dial by splitting the value
    *              returned from the ADC into even discrete steps based on the
118 *              given argument. For instance, an input of 7 will return
    *              a value from 0 to 7 based on the value found from the potentiometer
120 *
    * Arguments: int max - The number of maximum state to split values into
122 *
    * Returns: The dial position scaled by the maximum number
124 ********************************************************************/
   unsigned int readDial(unsigned int max)
126 {
       unsigned int value;
128    int divisor;

130    //Sets the AD channel for the DIAL
       ADCON0 = ADC_DIAL_READ;
132
       //Begin convertion
134    ConvertADC();
```

```
    while(BusyADC());
    value = ReadADC();
    if (value >= (ADC_MAX - 9)) return max;
    divisor = ADC_MAX/(max + 1);
    // Return a number from 0 - max
    return (value/divisor);
}

/*! ************************************************************************
 * Function: readDialForMEnu(void)
 *
 * Include:
 *
 * Description: Read the position of the dial for Navigation menus. Currently unused.
 *
 * Arguments: None
 *
 * Returns: Returns a value from 1 to max used in Navigation menus
 ************************************************************************/
unsigned int readDialForMenu(unsigned int max)
{
    return readDial(max -1 ) + 1;
}

/*! ************************************************************************
 * Function: userEmpty(void)
 *
 * Include: User_Interface.h
 *
 * Description: returns non-zero if the user interface buffer is empty (i.e.
 *              no user input has been detected)
 *
 * Arguments: None
 *
 * Returns: if the user input buffer is empty
 ************************************************************************/
char userEmpty(void)
{
    return empty(receive);
}

/*! ************************************************************************
 * Function: userPop(void)
 *
 * Include: User_Interface.h
 *
 * Description: pops a character from the user interface receive buffer
 *
 * Arguments: None
 *
 * Returns: a character poped from the user Interface buffer
 ************************************************************************/
extern char userPop(void)
{
    char c;
    c = pop(receive);
    return c;
```

```
192 }
194 /*! ***********************************************************************
    * Function: userPeek(void)
196 *
    * Include: User_Interface.h
198 *
    * Description: returns a character in the user_Interface buffer without
200 *              removing it from the buffer
    *
202 * Arguments: None
    *
204 * Returns: a character in the user interface buffer
    ************************************************************************/
206 extern char userPeek(void)
    {
208     return peek(receive);
    }
```

../Code/User_Interface.c