Technical Manual
Yavin IV Defence System

Team Dirac

November 3, 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. 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 document, 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
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

¡¡¡¡¡¡¡ HEAD The system is broken into ¡Give a technical description of the function of the whole system, in terms of its constituent parts, here termed modules. Generally, a module will have hardware and software parts. =======

The Yavin IV Orbital Tracking System's primary function is to track the 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 ¿¿¿¿¿¿¿ 50f34201c84e59bb9ccad87b59cc1f5e35c54be5

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

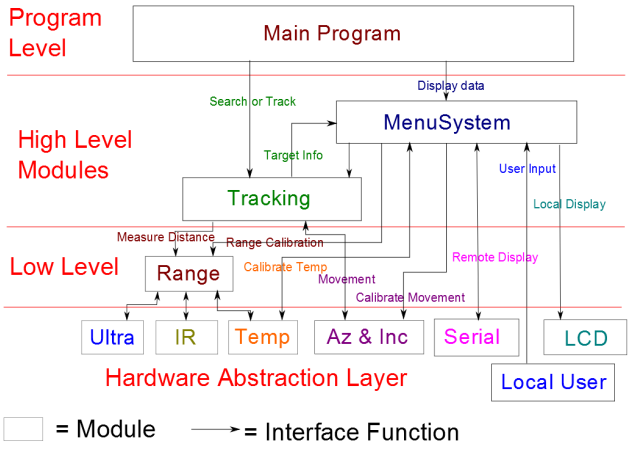

Figure 2.1: Conceptual Diagram of the module breakdown and interaction between modules.

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.

2.3 System Requirements

The operational scenarios considered place certain requirements on the whole Yavin IV Defence system, and on the modules that comprise it. ¡Statement of requirements that affect the system as a whole, and are not restricted to only a subset of its modules.¿

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

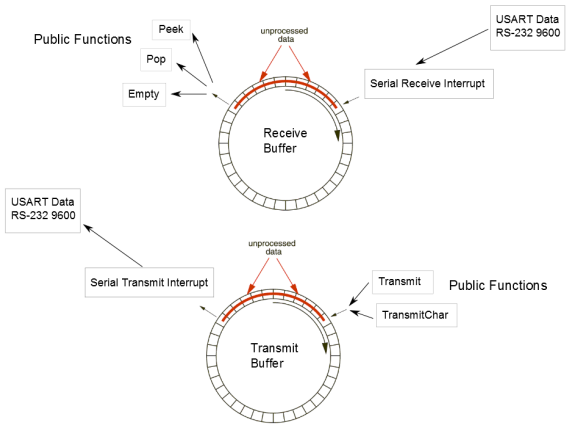

Figure 2.2: Shows Conceptual Diagram of Serial Module -¿ Received input stored by interrupts into circular buffer to await pop commands. Transmit data pushed onto buffer which is transmitted via interrupts

- 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 Technical User 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's

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

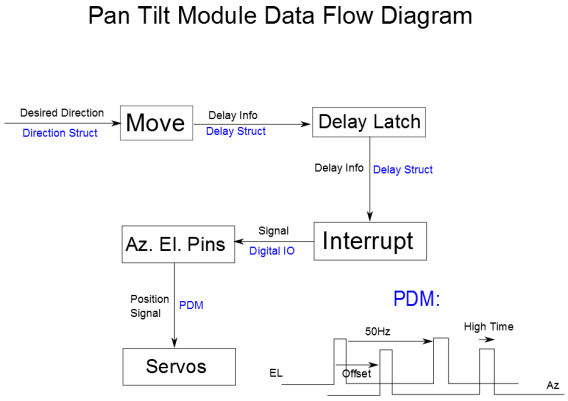

Figure 2.3: Dataflow diagram showing the transition from inputted direction to servo direction

- 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 Technical User 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 Functional Requirements

Inputs

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

- Analogue input from the IR sensor
- Time delayed echo signal from the Ultrasonic sensor

Processes

The range module must be capable of performing the following processes:

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

Outputs

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

- The calculated and fused range

Timing

The range module must be capable of precisely timing the delay from the starting of the ultrasonic sensor, and the return of the echo signal. In this implementation this is done via an input capture interrupt, which stores the timer value in hardware as soon as the event is triggered.

Failure Modes

The range system must be capable of timing out if no return echo signal is returned to avoid infinite loops. In this implementation we have a maximum delay before the sample times out.

2.9.2 Non-Functional Requirements

Performance

Interfaces

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

2.10 Conceptual Design: Range

2.10.1 Description

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

2.10.2 Overall Design

The range module was designed to sample the Ulrasonic range a specified number of times at a specified frequency. The module then samples the IR sensor as many times as possible at a specified frequency while waiting for the US echo to return. The data is fused and outputted to the user program.

2.10.3 Detailed Design

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.

2.10.4 Implemented Functionality

Implemented Basic Functionality

The following basic functionality has been implemented for the range module:

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

Implemented Additional Functionality

The following additional functionality has been implemented for the range module:

- 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

2.10.5 Assumptions Made

The range module assumes that the IR sensor output is inverse linear with respect to range, that the delay time is linear with target distance, and that the speed of sound is linear with temperature. None of these assumptions is valid in general, however, as we are only dealing with small deviations around a set point, it is a simple matter to linearise the actual functions around this value with a taylor series expansion. The linear approximations match the actual values quite closely for small deviations, as shown in the testing document.

2.10.6 Hardware

The range module makes use of the following hardware:

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

Pin Assignments

The range module uses the following pins to interface with hardware:

-

2.10.7 Constraints On Range Performance

The main constraint on the performance of the range module is the rate at which the ultrasonic sensor can sample, due to the return time, and the maximum specified sampling rate without breaking the sensor.

2.10.8 Interface

List of Inputs

The range module takes the following inputs:

- The calibration range (in mm as an unsigned int)
- The number of samples to take per estimate

List of Outputs

The range module returns the following outputs:

- The measured, fused range to the target in mm (as an unsigned int)
- An enumeration describing the type of signal return for searching and tracking purposes

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

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

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:

-

2.15.7 Interface

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

2.18 Module Requirements: MenuSubsystem

2.18.1 Functional Requirements

Inputs

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

-

Processes

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

-

Outputs

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

-

2.18.2 Non-Functional Requirements

Performance

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

-

Interfaces

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

-

2.19 Conceptual Design: MenuSubsystem

2.19.1 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. 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.

2.19.2 Overall Design

2.19.3 Detailed Design

2.19.4 Assumptions

2.19.5 Constraints On the MenuSubsystem

2.19.6 Interface

2.20 Module Requirements: LCD

2.20.1 Functional Requirements

Inputs

Processes

Timing

Failure Modes

2.20.2 Implemented Basic Functionality

Performance

Interfaces

2.21 Conceptual Design: LCD

2.21.1 Description

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

2.21.2 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.

2.21.3 Software

Timing

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

Reading Operation

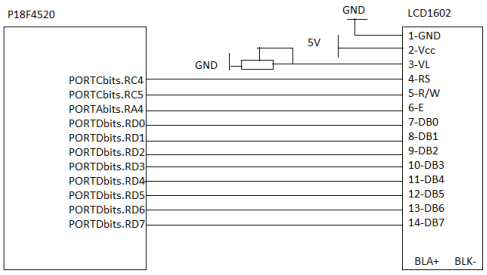

Figure 2.5: LCD Wiring Diagram - Shows how the 1602 LCD is wired up, and which pins are used to interface to it using the 8 bit mode

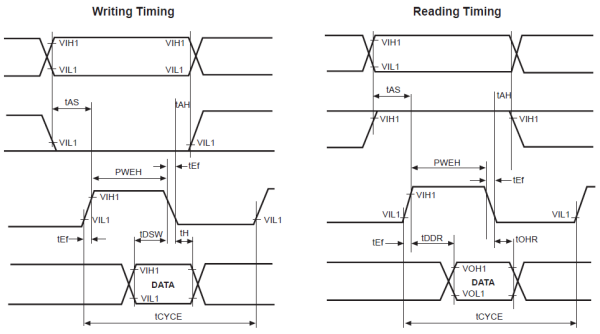

Figure 2.6: The LCD module requires precise timing, particularly for the initial configuration. This figure shows a timing diagram for the device taken from the datasheet

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.

Chapter 3

User Interface Design

¡Give a detailed description of the design of the user interface. This will gave the reader a god view of how the system functions from the user's perspective.¿

3.1 Classes of User

¡If there are different user interfaces presented to different classes of users, define there user casses, and how access by the various user classes is enabled or disabled.

3.2 Interface Design ¡User Class Y¿

3.2.1 User Inputs and Outputs

¡Description of how the user presents inputs to the system, and how the system responds to those inputs. Include a description of how the user knows the state of the system.¿

3.2.2 Input Validation and Error Trapping

¡Describe how the system validates user input, and how operator errors are trapped and can be recovered from¿

Chapter 4

Hardware Design

The system hardware consists of two main components: the local user interface controller, and the housing for the sensors and microcontroller.

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 all hardware is encased in an insulating enclosure that is not designed to be opened easily by users.
The power supply to the sensors and other hardware was split up into two independent and regulated 5V power rails, one exclusively for the ultrasonic module. This was done to limit the large surge currents the ultrasonic sensor draws from interfering with the rest of the circuit. A large capacitor was added across the ultrasonic power lines to limit this surge current from being drawn from the battery and causing voltage spikes.
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.

!!!Need to look up the power specifications for the regulators (e.g. max and min supply voltages) and include them here!!!

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

Actuator Hardware

Operator Input Hardware

Operator Output Hardware

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

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

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

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. 8.1. 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. 8.1.

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.

5.1.3 Software Quality Assurance

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

5.1.4 Software Design Description

Architecture

Describe the high-level architecture of the software - that is, the top-level flow of control, and how the various functional modules communicate. In this section, you can put state transition diagrams, sequence diagrams, etc.

Software Interface

Describe the public interface of each software module

Software Components

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

5.1.5 Preconditions for Software

Preconditions for System Startup

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

Preconditions for System Shutdown

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

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.

6.2 State of the System as Delivered

A statement of your group's opinion of the conformance of the system with the specification.

6.3 Future Improvements

Present a prioritised list of improvements to be made in future releases, giving reasons for the improvement and priority rank

Chapter 7

Safety Implications

Must identify foreseeable safety hazards associated with the equipment and then assess and control th identified risks - By law (NSW Occupational Health and Safety Act 2000)

7.1 Hazard Risk Table

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

Chapter 8

Conclusions

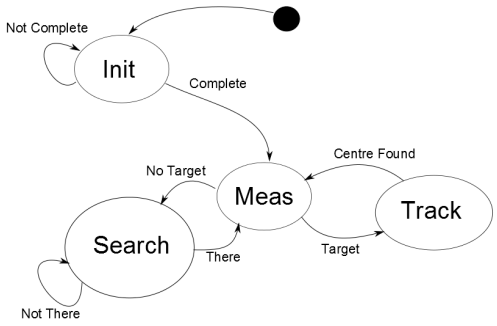

Figure 8.1: System overview state diagram