Technical Manuel
Yavin IV Defence System

Team Dirac

November 3, 2014

Chapter 1

Introduction

1.1 Document Identification

This document describes the design of the Yavin IV Defence System. This document is prepared by Group DIRAC for assessment in MTRX3700 in 2014.

1.2 System Overview

¡A brief statement of the purpose of the system or subsystem to which this document applies¿ The Yavin IV Defence System is designed to provide accurate, low cost tracking for Death Stars and other similar objects.

1.3 Document Overview

¡A short "road map" of the document, to provide an orientation for the reader. Summarise the purpose and contents of this document.¿ This document describes the detailed technical specifications and functionality for the entire system. This includes the entire system design, implementation and usage.

1.4 Reference Documents

The present document is prepared on the basis of the following reference documents, and should be read in conjunction with them. ¡Insert relevant documents¿

1.4.1 Acronyms and Abbreviations

Acronym	Meaning
Thing	Meaning of Thing
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 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.

2.2 Operational Scenarios

¡Describe how the system is to be used. There may be several different ways that it ca be used perhaps involving different users, or classes of user. Present case diagrams here if you are using them. Each operational scenario is a part through a use case diagram - a way of using the system, with different outcomes or methods of use. You should also consider the various failures that may occur, and the consequences of these failures. ¿

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.¿

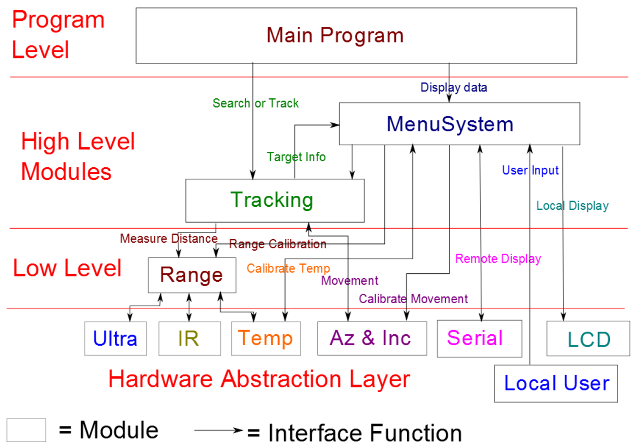

Figure 2.1: Conceptual Diagram of the module breakdown and interaction between modules.

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 Menu System

2.5.1 Description

This module handles the creation of all the menu system used in both the serial user interface, the local user interface and the factory user interface. It handles the parsing of user inputs, navigation between menus, and interfacing with the other modules to change system values and relay important information to the user.

2.5.2 Functional Requirements

Inputs

The menu system receives a variety of different inputs from the modules discussed below, as it must receive information from each subsystem and relay it to the user. When the menu is initialised, it requires a pointer to the tracking state of the system, so that the menu can know if the system is tracking or searching, and so that the system can enable and disable tracking mode.

The menu interfaces with the Range module and PanTilt module to access system variables such as the current maximum and minimum ranges so that the current values can be presented to the user, assisting the user with their ability to fine tune the system.

The menu system module also handles the ability to switch between a the remote serial user interface and the factory user interface, which contains hidden functions not available to a normal user.

Processes

Outputs

Timing

2.5.3 Non-Functional Requirements

2.5.4 Conceptual Design

Menu Structures

2.5.5 Software Interface

2.6 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 Functional Requirements

Inputs

The only external input to the system is characters or strings to transmit.
Either transmit, or transmitROM must be used depending on if the string is in RAM or ROM. The incorrect usage will result in printing nothing.
Sending a string which in not null terminated will result in unexpected behaviour, most probably filling and overflowing the buffer until a null is reached.
Sending a different datatype such as a float instead of a char will result in syntax errors, but other integer types may be casted into a char, truncating data and resulting in unexpected results. Again, the function only takes char's and integers must first be converted to a char array.

Processes

The following processes must be operational:

- The circular buffer functionality - pushing and popping characters from buffers must work properly for the module to operate.

Outputs

The system must output the following to work correctly:

- exact characters received over the serial line in the correct order.

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

Implemented Basic Functionality

The serial module implements the following basic functionality:

- Functioning receive and transmit serial 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

2.6.3 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

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)

2.6.4 Conceptual Design:

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

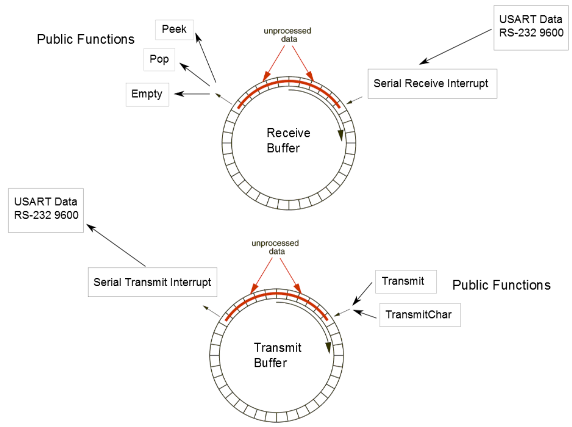

Figure 2.2: Shows Conceptual Diagram of Serial Module -¿ Received input stored by interrupts into circular buffer to await pop commands. Transmit data pushed onto buffer which is transmitted via interrupts

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.

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.

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.

2.6.5 Interface

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

2.7 Local User Interface

2.7.1 Description

2.8 Pan Tilt

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.

2.8.1 Functional Requirements

Inputs

The module takes the following inputs

- The Direction as a direction struct

Processes

The following processes must be operational for the functionality of the module

-

Outputs

The only program output is the current pan tilt direction.
The only other output is the PDM outputs to the servo which dictate their position.

Timing

The Pan Tilt module must have very precise timing to ensure that the servo positions are maintained precisely.

- 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

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

2.8.2 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

Implemented Additional Functionality

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

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

2.8.3 Conceptual 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. **??**.

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 PICDEM 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.

Pan Tilt Module Data Flow Diagram

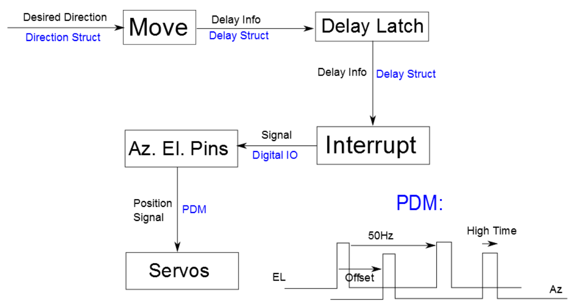

Figure 2.3: Dataflow diagram showing the transition from inputted direction to servo direction

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.

Interface

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

2.9 Range

2.9.1 Description

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

2.9.2 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

2.9.3 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

2.9.4 Conceptual 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.

Assumptions Made

Interface

2.10 Interrupts

2.10.1 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

2.10.2 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)

2.10.3 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

2.10.4 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

2.11 Temp

2.11.1 Description

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¿

3.3 Menu Design

3.3.1 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 **??**.

3.3.2 Menu Appearance

Chapter 4

Hardware Design

¡Give a detailed description of the design of hardware. The description should include mechanical drawings, location diagrams, electrical circuit schematics, circuit simulation or test results, PCB overlays, wiring diagrams, connector pin-out lists, pneumatic/hydraulic circuit diagrams¿

4.1 Scope of the ¡Something¿ System Hardware

Statement of what is, and what is not, being designed and described here.

4.2 Hardware Design

4.2.1 Power Supply

Power supply method and rating, fusing, distribution, grounding and protective earth as appropriate.

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, repeatability, 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)

Chapter 8

Conclusions

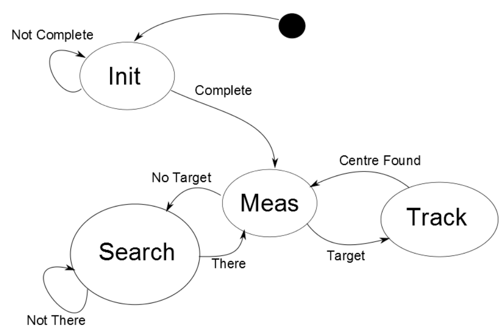

Figure 8.1: System overview state diagram