Technical Manual

Grant Louat

November 3, 2014

Abstract

This document describes how each of the modules (including both software and hardware) work, and how to use them.

Part I

Overview

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.

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 Orbital Tracking System design, of its division into hardware and software modules, and of its development and implementation.

2.1 Introduction

2.2 Operational Scenarios

2.3 System Requirements

2.4 Module Design

2.5 Module Requirements: Module X

2.5.1 Functional Requirements

Inputs

Processes

Outputs

Timing

Failure Modes

2.5.2 Non-Functional (Quality of Service) Requirements

Performance

Interfaces

Design Constraints

2.6 Conceptual Design: Module X

2.6.1 Assumptions Made

2.6.2 Constraints on Module X Performance

Part II

Introduction

2.7 Document Identification

2.8 System Overview

2.9 Document Overview

2.10 Reference Documents

2.10.1 Acronyms and Abbreviations

Part III

System Description

2.11 Introduction

2.12 Operational Scenarios

Chapter 3

Software Overview

3.1 Description

This system is a state machine; there is a primary loop in main() which repeatedly calls functions based on the current state. Each of these functions can alter the state to transition based on their function.

3.2 State Functions

- Initialisation
- Search - Tracking Module
- Track - Tracking Module

Part IV

Functionality

Chapter 4

Factory Mode

4.1 Description

The Yavin IV Death Star Tracker includes a Factory Mode not available to general users. This mode is used for in-factory calibration and testing.

4.2 Entering/Exiting the Factory Mode

The factory mode can be entered by ... Similarly ... will exit the factory mode, and the system will return to the standard operational mode.

4.3 Functionality

The Factory Mode adds the following functionality and commands to the standard Mode

-

Part V

Software

Chapter 5

Interrupts

5.1 Description

Many of the functions within different modules use interrupts, however there is only one service routine for high and low priority. Thus the ISR's and vectors have been moved into their own separate module, which call Module scope ISR functions.

5.2 How It Works:

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.

5.3 How To Use It:

Follow the following steps to use an interrupt within a module.

1. Make sure Interrupts.c is included in the build
2. Define/Update the module macro in the module header file
3. Check header file is included in Interrupts.c

4. Check module macro is tested in ISR's (High and low ISR's)

5. Use module ISR as general ISR

5.4 List of Public Functions

This module has no public functions, or any real public interface as nothing within this module is called by anything within the program. Rather the list of included header files, and the macro checks must be updated when a new module is added.

5.5 Example Code

5.5.1 Defining Module macro

```
//Serial Interrupt macro Defn. within Serial.h
#define SERIAL_INT (TX_INT || RC_INT)
```

5.5.2 Updating Interrupt Includes

```
//Includes at the top of Interrupt.c
//Any module that uses interrupts will need to be included here:
#include "Tracking.h"
#include "Range.h"
#include "User_Interface.h"
#include "Serial.h"
#include "PanTilt.h"
#include "Temp.h"
#include "Menusystem.h"
//Add your module header file here...
```

5.5.3 Updating ISR Check

```
if (SERIAL_INT)
{
serialISR();
}
if (PAN_TILT_ISR)
{
panTiltISR();
}
if (RANGE_INT)
{
rangeISR();
}
```

```
if (USER_INT)
{
userISR();
}
//Add additional checks here
```

Chapter 6

Serial

6.1 Description:

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

6.2 How It Works:

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.
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.

6.3 How To Use It:

- Make sure Interrupts.c is included in the build
- Call the configureSerial() function to set up serial
- Call transmit() to begin transmitting something
- Use receiveEmpty() to check if anything has been received
- Use readString() to read everything received up to a carriage return

6.4 List of Public Functions

Below is a definitive list of all the public functions contained in this module, their use, their arguments and their returns.

6.4.1 configureSerial

Usage:

Used to set up the serial module for use. This function MUST be called before any serial functions will work. It need only be called ONCE at the beginning of the program.

Arguments:

This function takes no arguments

Return Value:

This function does not return anything.

6.4.2 transmit

Usage:

Used to transmit a string over serial UART. NOTE: The string MUST be NULL TERMINATED. The function will continue pushing data onto the transmit buffer until it hits a null terminator, which could fill the buffer and overwrite the given string if it is not properly null terminated.
IMPORTANT: This function CANNOT be passed a literal. E.g. the following code will not work:

```
transmit("Some Message");
```

The literal is stored in ROM, and the pointer argument the function uses cannot access ROM memory. This will result in unexpected behaviour. Instead use the following code:

```
char message[] = "Some Message";
transmit(message);
```

This loads the literal into RAM when the program begins.

Arguments:

A pointer to the start of a null-terminated string to begin transmitting.

Return value:

No return value

6.4.3 transmitted

Usage:

Used to determine if a transmit action has been completed. For example if coordination between serial transmissions and other actions is required then this function must be used.

Arguments:

This function takes no arguments

Return Value:

non-zero (true) if all transmit actions have been completed (i.e. the transmit buffer is empty).

6.4.4 transChar

Usage

Used to transmit a single character over serial. Works exactly the same way as transmit, and will thus not interfere with any other transmit function. Use of this function will remove the necessity of defining a string and null terminating it when only a single character has to be transmitted.

Arguments:

A character to transmit over serial

Return Value:

No return value

6.4.5 receiveEmpty

Usage:

Used to determine if anything has been received over serial.

Arguments:

No Arguments

Return Value:

Non-zero if the the receive buffer is empty (nothing new has been received).

6.4.6 receivePeek

Usage:

Used to inspect the next character received over serial without removing it from the buffer. (e.g. a subsequent receivePeek, or receivePop operation will return the same character).

Arguments:

No arguments

Return Value:

The next character received over serial

6.4.7 receivePop

Usage:

Used to read a character received over serial.

Arguments:

No Arguments

Return Value:

the next character received over serial

6.4.8 receiveCR

Usage:

Used to determine whether a Carriage Return has been detected over serial. Carriage Return is used as a "user input confirm" button.

Arguments:

No Arguments

Return Value:

Non-zero if a carriage return has been detected (i.e. there is a CR in the receive buffer)

6.4.9 readString

Usage:

Used to read an entire string received over serial.

Arguments:

Pointer to memory address to store received characters.
NOTE: To avoid possible memory errors, Make sure to reserve the maximum number of characters that can be returned. The macro BUFFERLENGTH (the length of the receive buffer) is defined in CircularBuffers.h. E.g. the following code is advisable:

```
#include "CircularBuffers.h"
...
char [BUFFERLENGTH] receivedData;        //Reserve the max number of characters
readString(receivedData);                //Read string into address
```

Return Value:

No return value

6.5 Example Code:

```
#include "Common.h"
#include "Serial.h"
#include "CircularBuffers.h"     //Only used for BUFFERLENGTH definition

void main()
{
        char message[] = "Serial Example";
        char received[BUFFERLENGTH];

        //Set up the Serial module for use
        configureSerial();

        //Transmit the message
```

```
        transmit(message);

        //Wait for a carriage return
        while(!receiveCR());

        //Read in returned string
        readString(received);

        //Transmit received string
        transmit(received);

        for (;;)
        {
                //Poll received data
                while (receiveEmpty());

                //Re-transmit received character
                transChar(receivePop());
        }
}
```

Chapter 7

PanTilt

7.1 Description:

The Pan Tilt module is the module which controls the servo actuators which point the sensors in the correct direction. As such it generates the PDM's (Pulse Duration Modulated waves) necessary to run the servo's, and has public functions such as move, which directs the Pan Tilt mechanism to a certain direction.

7.2 How It Works:

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.

7.3 How To Use It:

- Ensure Interrupt.c is included in the build
- Include PanTilt.h in the file you want to use the module
- Call configureBase() function to set up the module
- Call Move() to send the base to an absolute direction
- Call increment() or incrementFine() to move the base to a relative direction (relative to its current)

- CalibratePanTilt() to calibrate the base direction
- RawDir() returns the calibrated direction
- getDir() returns the current direction

7.4 List of Public Functions

Below is a definitive list of all the public functions contained in this module, their use, their arguments and their returns.

7.4.1 configureBase

Description:

Configures the Pan Tilt module. After this function is called the module moves to its default position. Then the module is fully possibly moving to any desired direction with the move or the increment functions. This function need only be called once at the beginning of the program (or whenever you want to start using the PanTilt).

Usage:

This function is used to setup the Pan Tilt module so that one can direct the sensors to any direction desired.

Arguments:

This function takes no arguments.

Return Value:

This function takes no arguments.

7.4.2 Move

Description

This function commands the pan tilt mechanism to the given position. For this function (0, 0) is defined as the midpoint in both the elevation and azimuth. In azimuth this means pointing along the length of the pan tilt mechanism, and in elevation this means pointing 45 degrees above the ground.

Usage

This function is used when you want to point at an absolute direction.

Arguments

This function takes a single object of type direction which contains the desired azimuth and elevation in degrees

Return Value

This function returns nothing.

7.4.3 Increment

Description

This function adds an offset onto the current pan tilt direction

Usage

This function is used when you want to move the direction pointed to by some amount, without particularly caring about the absolute direction.

Arguments

This function takes a single object of type Direction, which contains the incremental value in both the azimuth and elevation directions in degrees.

Return Value

This function returns nothing

7.4.4 IncrementFine

Description:

This function does the same thing as the Increment function, but it increments in the smallest possible resolution of the pan Tilt module, not degrees.

Usage

This function is used in lieu of increment when greater angular resolution is required, such as the tracking algorithm.

Arguments

This function takes a single object of type Direction (defined in common.h) which represents the desired relative direction. Unlike the increment function this argument is *not in degrees*, but the smallest possible resolution, individual clock ticks on the high time. For the 10MHz clock on the minimal board this is expected to be 0.04deg.

Return Value

None

7.4.5 CalibratePanTilt

Description:

This function calibrates the pan tilt offset to the given value. Calling this function ensures that future calls to the move function with the same direction passed as reference will return to this point.

Usage

This function is used to change the reference position of the pan tilt mechanism.

Arguments

Takes a single argument in the form of a direction struct (defined in common.h). This direction corresponds to the reference direction to which the pan tilt mechanism is calibrated.

Return Value

This function does not return anything.

7.4.6 CalibratePanTiltRange

Description:

This function calibrates the pan tilt mechanism by changing the range of the mechanism. This function acts very similarly to the CalibratePanTilt function except that the reference position stays the same and the mechanism increments are changed so that any future move call to the same reference will return to this position.

Usage

This function is used to calibrate the pan tilt mechanism once the desired reference position has been set

Arguments

Takes a single argument in the form of a direction struct (defined in common.h). This direction corresponds to the reference direction to which the pan tilt mechanism is calibrated.

Return Value

This function does not return anything.

7.4.7 SetMaxMin

Description

This function sets the value of any accessible setting in the PanTilt module, such as the Maximum and Minimum Azimuth or inclination

Usage

This function is used to alter any accessible setting in the pan tilt module, primarily from user input

Arguments

The function takes the new value and an enumeration of the setting in which to write the value

Return Value

This function does not return anything

7.4.8 GetMaxMin

Description

This function returns the value in any accessible setting within the PanTilt module, such as the maximum and minimum azimuth and elevation

Usage

This function is used whenever a setting value needs to be accessed. Primarily for outputting to the user

Arguments

The setting (as an enumeration) to return

Return Value

The value currently in the queried setting (as a character).

7.5 Example Code:

```
#include "Common.h"
#include "PanTilt.h"

void main()
{
        Direction dir;
        configureBase();        //Configure the base module

        dir.azimuth = 0;        //Define a direction to move to
        dir.inclination = 0;

        move(dir);                      //Move the base to the defined direction

        dir.azimuth = 1;        //Define an increment
        dir.inclination = 1;

        increment(dir);         //Increment the current direction by 1 degree in
}
```

Chapter 8

Temp

8.1 Description

The Temperature module controls the sampling and calibration of the temperature sensor.

8.2 How It Works:

The temperature module is very simply and mostly revolves around the AD sampling of the temperature sensor. The temperature is not expected to fluctuate much over a short period of time. For this reason the temperature is only sampled over reasonably long intervals (e.g. a minute). The temperature module simply stores the last measured temperature and returns this in the getTemp() function.

8.3 How To Use It:

- Include Temp.h in your source code file
- Call configureTemp() to setup the module
- Call readTemp() or readTempx2() to re-sample the temperature
- Call getTemp() to return the last sample temperature

8.4 List of Public Functions

Below is a definitive list of all the public functions contained in this module, their use, their arguments and their returns.

8.4.1 configureTemp

Description

Configures the temperature module so that it can be used

Usage:

This function is used once at the beginning of the program to configure the temperature module for use.

Arguments:

None

Return Value:

None

8.4.2 rawTemp

Description:

This function returns the raw temperature, the sensor data without any calibration offset.

Usage

This function is primarily for when the user wants to display the raw temperature reading

Arguments

None

Return Value

The raw temperature reading in degrees

8.4.3 readTemp

Description

This function takes a sensor read of the temperature, returns the result and stores it in the temperature module

Usage

This function is called periodically to re-sample the temperature, to account for environmental differences during use.

Arguments

None

Return Value

The temperature read by the sensor (and calibrated).

8.4.4 readTempx2

Description

This function does the same thing as readTemp, but it returns twice the temperature in degrees, allowing for better resolution.

Usage

Used in lieu of readTemp when better resolution is required

Arguments

None

Return Value

Twice the temperature x2

8.4.5 calibrateTemp

Description

Calibrates the current sensor reading to a given value

Usage

Used to calibrate an offset in the temperature sensor module

Arguments

Takes the temperature to calibrate the temperature sensor to

Return Value

None

8.4.6 getTemp

Description

Returns the temperature from the last sample taken

Usage

Used to get the temperature without having to re-sample the temperature sensor.

Arguments

None

Return Value

The temperature from the last sample

8.5 Example Code:

```
#include "Temp.h"

void main(void)
{
        unsigned char temperature;

        configureTemp();
        calibrateTemp(25); //Calibrate temperature sensor to 25deg

        temperature = readTemp();

        //Just keeps returning the same temperature
        for (;;) temperature = getTemp();
}
```

Chapter 9

Tracking

9.1 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.

9.2 How It Works:

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 tracking algorithm tracks the centre of the target by finding the edges in both azimuth and elevation, and moving back to the centre.
This module makes use of the panTilt, range and temp modules which in turn use interrupts. This means that Interrupts.c must be included it the build for the tracking module to work. A user need not worry about configuring the modules used by tracking, as everything is set up in the configureTracking function

9.3 How To Use It:

- Include "Tracking.h"
- Call configureTracking()
- Repeatedly Call Search()

9.4 List of Public Functions

Below is a definitive list of all the public functions contained in this module, their use, their arguments and their returns.

9.4.1 ConfigureTracking

Description:

Configures the tracking module, and all subsequent modules used by the tracking module.

Usage:

This function must be called before making use of any functions within the tracking module.

Arguments:

None

Return Value:

None

9.4.2 Search

Description:

Performs a single increment of the Pan Tilt mechanism in a defined search pattern.

9.5 Example Code:

Chapter 10

User_Interface

10.1 Description

The user interface module is the equivalent of the serial module for the local user interface. It has a circular buffer and stores all the user inputs to the system to be handled later.

10.2 How It Works:

10.3 How To Use It:

-

10.4 List of Public Functions

Below is a definitive list of all the public functions contained in this module, their use, their arguments and their returns.

10.4.1

Usage:

Arguments:

Return Value:

10.5 Example Code:

Chapter 11

Range

11.1 Description

The Rang module is responsible for sampling the range from the IR and ultrasonic sensors. It is also used to detect the object as the functions return zero if the object is out of range.

11.2 How It Works:

11.3 How To Use It:

-

11.4 List of Public Functions

Below is a definitive list of all the public functions contained in this module, their use, their arguments and their returns.

11.4.1

Usage:

Arguments:

Return Value:

11.5 Example Code:

Chapter 12

MenuSystem

12.1 Description

The Menu System Module is responsible for keeping track of the menu state, and parses all user inputs. The module interfaces with the serial and user interface modules to handle all inputs to the system.

12.2 How It Works:

12.3 How To Use It:

-

12.4 List of Public Functions

Below is a definitive list of all the public functions contained in this module, their use, their arguments and their returns.

12.4.1

Usage:

Arguments:

Return Value:

12.5 Example Code:

Chapter 13

LCD

13.1 Description

The LCD module is responsible for interfacing with the LCD component and displaying LCD data in the local user mode.

13.2 How It Works:

13.3 How To Use It:

-

13.4 List of Public Functions

Below is a definitive list of all the public functions contained in this module, their use, their arguments and their returns.

13.4.1

Usage:

Arguments:

Return Value:

13.5 Example Code:

Part VI

Hardware

Chapter 14

Assembly Instructions

Follow the following instructions to assemble the system. (see the following sections for detailed information about each step)

1. Connect the Power
2. Connect the Servos
3. Connect the ultrasonic
4. Connect the IR sensor
5. Connect the Temperature sensor
6. Plug in Serial cable (if used)

14.1 Power Connections

The system runs of a 5v regulated line powered by a 9v battery. The ultrasonic sensor has its own regulator to account for its power surges, so it is treated differently. All other components can be powered off the remaining 5v bus.

1. Make sure battery is inserted, and has charge
2.

Chapter 15

Pin Connections

15.0.1 Ultrasonic

The ultrasonic sensor uses the following pins:

- INIT - RC3 (Ultrasound pin 4)
- ECHO - RC2 (Ultrasound pin 7)
- V+ - Ultrasonic 5v bus (Ultrasound pin 9)
- GND - Ultrasonic 0v (Ultrasound pin 1)
- BINH - GND (Ultrasound pin 8)
- BLNK - GND (Ultrasound pin 2)

Refer to the diagrams 15.0.1 below for the Ultrasonic and other pan tilt module pins.

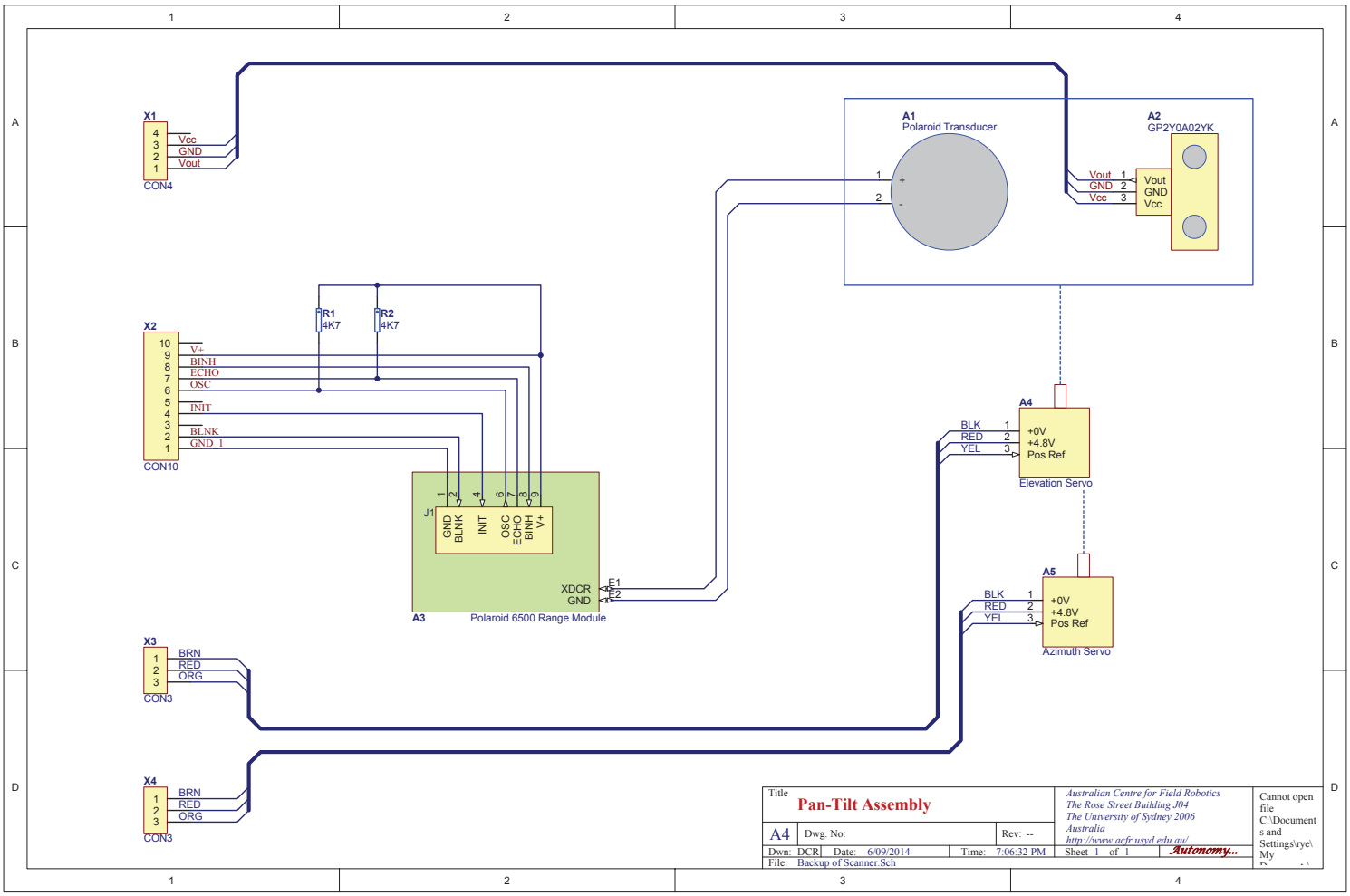


Figure 2: Electrical Schematic of Scanner Assembly.

;

15.0.2 IR

The IR uses the following pins:

- Output - AN0 (IR pin 1)
- Vcc - 5v bus (IR pin 3)
- GND - 0v bus (IR pin 2)

15.0.3 Servos

The servos make use of the following pins:

- PDM (Az.) - RC0 (YEL)
- PDM (El.) - RC1 (YEL)
- V+ - 5v bus (RED)
- GND - 0v bus (BWN)

15.0.4 Temperature Sensor

The temperature sensor makes use of the following pins:

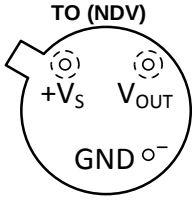
- Output - AN1
- V+ - 5v bus
- GND - 0v bus

Refer to the diagram below for the pin placements of the temperature sensor.

These devices have limited built-in ESD protection. The leads should be shorted together or the device placed in conductive foam during storage or handling to prevent electrostatic damage to the MOS gates.

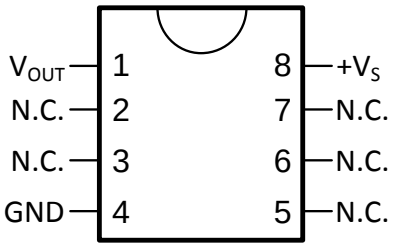
CONNECTION DIAGRAMS

METAL CAN PACKAGE
TO (NDV)



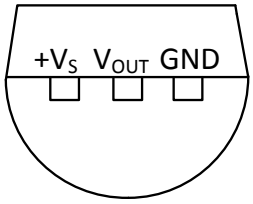
Case is connected to negative pin (GND)

SMALL-OUTLINE MOLDED PACKAGE
SOIC-8 (D)
TOP VIEW

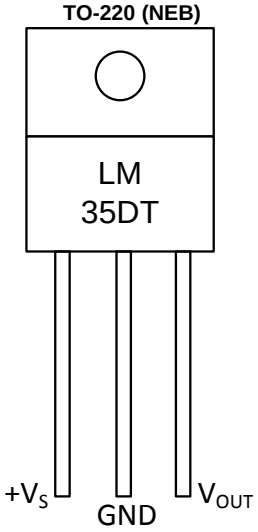


N.C. = No connection

PLASTIC PACKAGE
TO-92 (LP)
BOTTOM VIEW



PLASTIC PACKAGE
TO-220 (NEB)



Tab is connected to the negative pin (GND).

NOTE: The LM35DT pinout is different than the discontinued LM35DP

;

15.0.5 LCD

15.0.6 Buttons (Interface)

Chapter 16

Power Bus

16.0.7 Description

The power bus is responsible for supplying power to the entire system. It must be capable of running off a single 9v battery, and supplying the regulated 5v required for all of the components used.

16.0.8 Circuit Diagram

Fig. 16.1 is a schematic of the power bus used in this system. The bypass capacitors are used to reduce current draw during sudden voltage switching. The ultrasonic sensor has a separate voltage regulator as even the 1mF bypass capacitor cannot completely regulate the 2A power spike when the ultrasonic sensor emits. By decoupling the regulators all other components aren't affected by current draw on the ultrasonic unless the internal battery voltage drop lowers the input of the regulator below 6v.

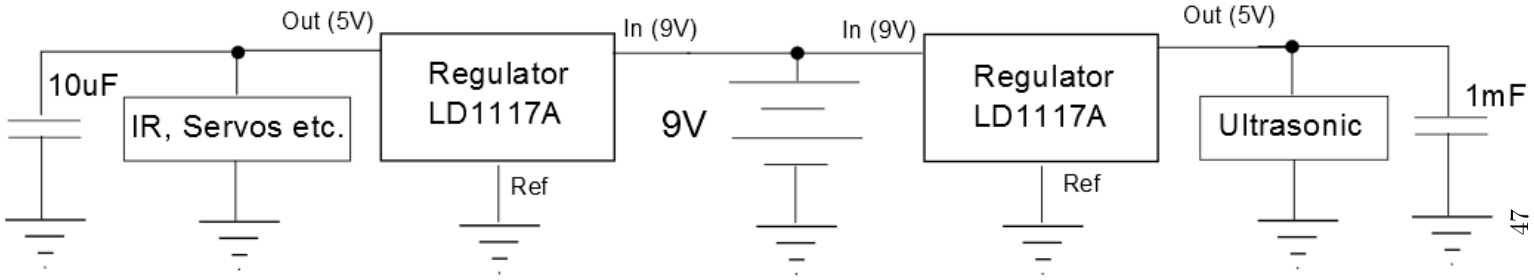


Figure 16.1: Power Bus Circuit Diagram

Part VII

Trouble Shooting

Chapter 17

Hardware

17.1 Unexpected Target Address Programming PIC

17.1.1 Description

While trying to program the PIC with a PICkit, or ICD a message just keeps coming up saying the target address is 0x00 or something, and cannot program the device.

17.1.2 Cause

This is a generic issue for just about any communications problem between the PIC and the programming device. David has mentioned several times that it can be caused by interference on the bus. Personally I have found it is usually the capacitor on MCLR to be the issue. MCLR is the external reset for the PIC, and is connected to the programmer so that the device can be reset to run the program etc. If the capacitor is too large the device cannot change the value fast enough, and you will just get zeros. If the cap is too small it may float somewhat and get random values.

17.1.3 Solution

The nominal cap value on MCLR (at least for the PICkit) is 0.1μF. If this is not what you have, perhaps try 0.1μF. We have also found success by removing the cap altogether. Otherwise see if you can shorten the cable, and/or check the programmer output on an oscilloscope to see if it is broken.

17.2 'Sneezing' Motion

17.2.1 Description

The servos seem to jump a little whenever the Ultrasonic sensor fires, giving the impression of 'sneezing'.

17.2.2 Cause

This problem is a result of connecting the servos and ultrasonic sensor to the same power bus. When the ultrasonic sensor

res it draws 2A, which is enough to case a sizeable voltage drop on the power bus. This is enough to effect the position of the servos, and probably the IR sensor, although this is not as noticeable.

17.2.3 Solution

Solve this problem by connecting the ultrasonic sensor to the separate power bus provided.

17.3 Jittery Servos

17.3.1 Description

The servos are jittering around all over the place and everything in the software seems to be correct.

17.3.2 Cause

This may be caused by not supplying a common ground to the servos. The servos use a control PDM signal sent from the PIC. But the servos may be powered from a different power source and thus the voltage between the grounds can float.

17.3.3 Solution

Solve this by creating a common ground between the servos and the PIC - by connecting the grounds together.

17.4 Servos Going Crazy

17.5 Description

The pan Tilt module has been configured and used as documented, but the servos (or one of the servos) are going haywire.

17.5.1 Cause

Plugging the PDM outputs to the servos may be distorting the PDM signals. It is not entirely known why this is the case; check this on an oscilloscope with the PDM outputs both connected and disconnected to the servo inputs.

17.5.2 Solution

Try placing a voltage follower between the PDM output and the servo inputs. This acts as a buffer and attempts to prevent the servos input characteristics impacting the output signal from the PIC.

Chapter 18

Software

18.1 Module Just Not Working

18.1.1 Description

A module, or functions within the module are just not working, and you have no idea why.

18.1.2 Cause

You may have forgotten to call the associated configure function, or left out the Interrupts.c file.

18.1.3 Solution

Make absolutely sure that the relevant configure function has been called. Also make sure that the Interrupts.c file has been added to the build (won't matter for non interrupt driven modules).

18.2 Public Interface Functions Not Declared

18.2.1 Description

Whenever you try to use the documented public interface functions the compiler tells you that they are not declared.

18.2.2 Cause

You may have forgotten to include the relevant header file, which contains external declaration of all the functions so that you can see/use them elsewhere in the program.

18.2.3 Solution

Include the relevant header file.

18.3 Serial Not Transmitting

18.3.1 Description

Serial has been set up as documented, but the transmit function is not working, or works some of the time. May also be transmitting random stuff.

18.3.2 Cause

A likely source of issue is trying to pass a literal to the transmit function. As detailed in the sections above, a literal is stored in program memory. The pointer passed to the function can only access RAM.

18.3.3 Solution

To solve this problem initialise a string variable and pass a pointer to the variable. The variable is stored in RAM and initialised from the program memory literal.

18.4 Serial Transmitting Garbage

18.4.1 Description

All of the serial module steps have been followed, the serial is being transmitted and received, but all data is garbage.

18.4.2 Cause

A likely source for this problem is an incorrect baud rate being set.

18.4.3 Solution

Make sure that MNML is declared in Common.h if using the minimal board, or not set if using a picdem. Also check that the terminal is receiving at 9600 baud. If an external clock or oscillator is used that is neither 10MHz, nor 4MHz then small changes must be made in the serial module.

18.4.4 Cause

Another possible cause is that the string you are passing is not correctly null terminated. In this case you should expect to see some number (up to BUFFER-

LENGTH) characters transmitted which may or may not include the desired string.