Testing Document

Grant Louat

October 14, 2014

Abstract

This document details the testing of all functions, and their current state. Each function will be tested in isolation. This document should detail by what criteria the function has been deemed correct, the date and who did the testing. It should also have a listing of the working code so there is no confusion over different versions. This will quickly tell everyone which functions have been confirmed correct, and the rigour to which that assertion was made. It will also tell them when it was confirmed in case they have been working on a local copy of the code, and who deemed it correct in case they have questions. This document should also have a description of the function so that people know what the function should be doing when it is working.
The document deals only with the isolation testing of each function. The functions will also need to be tested in an integration phase later.
WARNING: DO NOT INCLUDE ANYTHING IN THIS DOCUMENT UNLESS YOU ARE CERTAIN THAT IT IS COMPLETELY ERROR FREE!
Even if it is a simple one line function there could be a type issue or something. Anything document will be assumed working, and if a single error remains it will nullify the entire document. If a function is too large or complex to test and verify in complete certainty then it should be split into smaller functions to solve parts of the problem. Make sure also to include WHY you verified the function correct, as your rigour may not meet the standard of someone else's.

0.1 Common

0.1.1 DIV_X Macros:

Status:

Working as of 11:30 25/9/2014 - Grant

Description:

This set of Macros are designed to divide an integer by a power of two.

Criteria:

The value of 1024 was tested for each of the macros, and due to the simplicity and nature of the macros no other values need be tested.

Working code:

```
#define DIV_2(v) ((v) >> 1)          //Divide by 2
```

0.1.2 SWAP Macro

Status:

Working as of 10:00pm 6/10/2014 - Grant

Description:

This macro provides an efficient way of swapping the values in two variables using xor operations.

Criteria:

The macro was tested on the values x = 7, y = 3, the macro returned x = 3, y = 7. As this was the expected result and any error in the macro would have manifested an error in the result, the macro was thus assumed to be correct.

Working Code:

```
#define SWAP(x, y) (y = (y ^ (x = (x ^ (y = (x ^ y))))))
```

0.2 Temperature Module:

0.2.1 ReadTempx2:

Status:

Currently untested as of 8am 26/9/14 - Grant

Description:

Returns the temperature (x2) in degrees Celsius by performing an A/D conversion on the analogue output from the temperature sensor.

0.2.2 ReadTemp:

Status:

Working as of 8am 26/9/2014 - Grant

Criteria:

This function was tested in isolation of the ReadTempx2, and the assertion only applies to this function. A ReadTempx2 stub function was written to simply return the value of 40 to test the ReadTemp function.

Description:

The function simply calls the ReadTempx2 function and divides the result by 2.

Working Code:

```
unsigned char readTemp(void)
{
unsigned char temp;

//Read the temperature from the x2 function
temp = readTempx2();

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

0.2.3 RawTemp:

Status:

Working as of 8:30am 26/9/2014 - Grant

Criteria:

The function was tested in isolation of the ReadTemp functions. A dummy function placed the value of 60 into the lastTempx2 variable, and the RawTemp function worked as described.

Description:

This function returns the uncalibrated result of the last temperature read. E.g. the raw sensor output. A temperature read is performed by either a ReadTemp or ReadTempx2 call.
: This function cannot be completely tested in isolation as it requires an external static declaration of the lastTempx2 variable. It also requires the ReadTemp and ReadTempx2 functions to write to this variable when they perform a read.

Working Code:

```
unsigned char rawTemp(void)
{
return DIV_2(lastTempx2);
}
```

0.2.4 Calibrate Temp:

Status:

Working as of 10:30am 26/9/2014 - Grant

Description:

Calibrates the last temperature read to the passed reference value by updating a static variable.

Criteria:

This function was tested in isolation of the readTemp functions. a dummy readTempx2 function which just returned 15deg (and set the static variable) was written, and the calibration function called to calibrate it to 20 deg, and then 10 deg. In both cases a second call of readTempx2 returned the desired value even though the 'raw data' was simply hard coded in.

Working Code:

```
void calibrateTemp(unsigned char reference)
{
calibration_offset = 2 * (reference - DIV_2(lastTempx2));
}
```

0.2.5 GetTemp:

Status:

Working as of 10:30 26/9/2014 - Grant

Description:

Returns the result of the last temperature read.

Criteria:

This is just an accessor function, so it simply returns the value of a static variable.

Working Code:

```
unsigned char getTemp(void)
{
return DIV_2(lastTempx2);
}
```

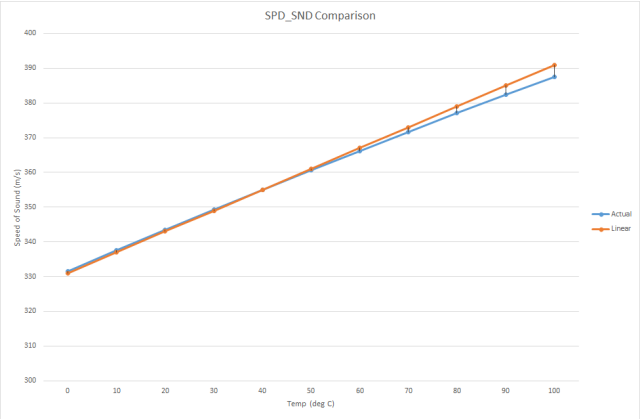

Figure 1: Comparison between SPD_SND() Macro output and actual function value

0.3 Range

0.3.1 SPD_SND Macro

Status:

Working as of 10:30 25/9/14 - Grant

Description:

This macro performs a linear approximation of the given function for the sake of computational efficiency. This means the computations performed are very different from the function provided, but the results closely approximate those of the actual function within the relevant range. See Fig. 1 for a comparison.

Criteria:

Each temperature from 0 to 100 deg C was tested and compared to the actual formula result. There is some deviation, but not much to make a significant difference. See Fig. 1.

Working Code:

```
#define SPD_SND(T) (DIV_1024(T * (unsigned int)614) + 331)
```

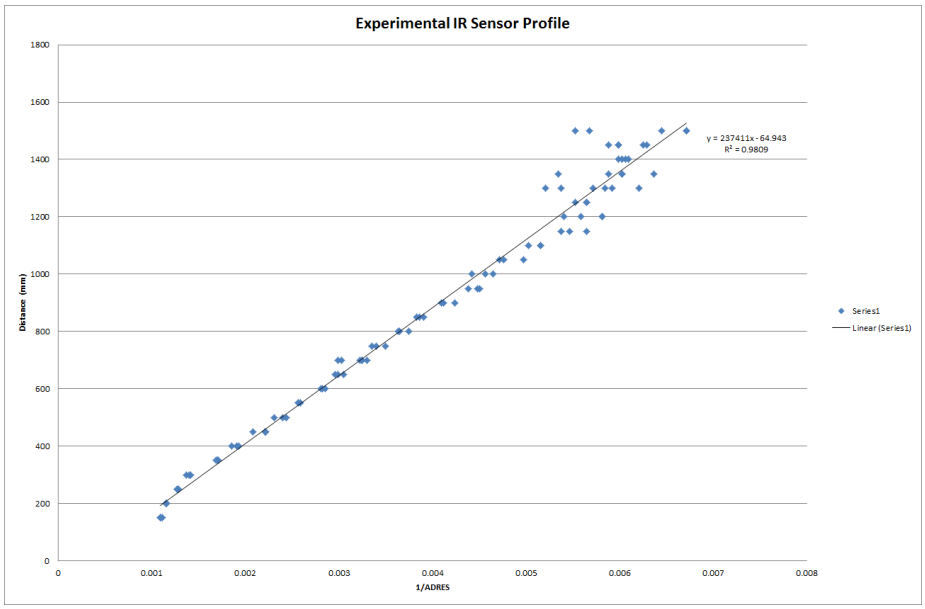

Figure 2: IR Experimental Data - Sensor output profile

0.3.2 IR_CONV Macro:

Status:

Working as of 1:40pm 30/9/2014 - Grant
As of 13/10/2014 The calibration on the minimal board seems to differ. As of yet we have no idea why, and mean to re-calibrate it

Description:

This macro converts the AD result into a range in mm, based on experimental readings from the IR Sensor.

Criteria:

This macro has been tested with a range of different input values. Each input returned a distance within a couple of centimetres of the experimental data (which was within the experimental error). See Fig. 2 for the sensor profile.

Working Code:

```
#define IR_CONV(ad) ((unsigned long)237411 / (ad) - 65)
```

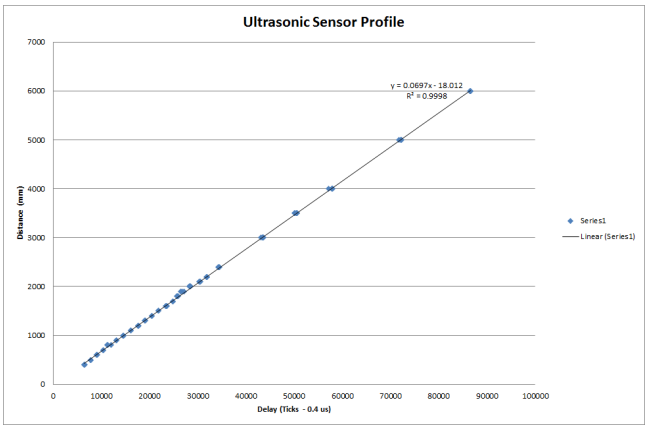

Figure 3: Ultrasonic Range profile. The measured distance against the time delay (in system ticks at 2.5MHz)

0.3.3 US_CONV

Status:

Currently working as of 3:15 13/10/2014 - Grant

Description:

This function converts a timing value returned by the input capture into an actual range for the ultrasonic sensor.
NOTE: This macro was designed for, and tested on the MINIMAL board. The ticks on the demo board will be 4x longer (as the clock is 4MHz instead of 10MHz). If using the demo board, multiply the system ticks by 4.
This macro also uses a linearisation of the effect on speed of sound by temperature to change the gradient of the delay to distance conversion. This has not been tested as there is no easy way to change the temperature and test the result.

Criteria:

This function has been tested at different ranges, by comparing the range result to the measured distance to a target. The function was created from experimental data and a linear fit as seen in Fig. 3.

Working Code:

```
#define ULTRA_CONV(tme, T) DIV_65536(tme * (unsigned long)(DIV_65536(519078 * T)
```

0.3.4 calibrateRange:

Status:

Currently Untested as of 12:10pm 10/10/2014 - Grant

Description:

Calculates the offset to calibrate a calculated range to an argument.

Criteria:

0.3.5 rawRange:

Status:

untested as of 11am 26/9/2014 - Grant

Description:

Accessor function - simply returns the uncalibrated sensor output from the last range calculation.

Criteria:

0.3.6 range:

Status:

Untested as of 11am 26/9/14 - Grant

Description:

This function performs a read of both the ultrasonic and IR sensors, fuses them and calibrates the result. The current version simply averages the IR and ultrasonic ranges as an initial fusion method.

Criteria:

0.3.7 rangeIR:

Status:

Working as of 11am 26/9/14 - Grant
We have found the calibration to have changed (on the minimal board) since the last we tested the sensor - 13/10/2014 Grant. The operation of the function is otherwise fine.

Description:

This function performs a read of the IR sensor, calibrates it and returns the result.

Criteria:

This function has been tested with a range of different ranges (using a book and measuring the distance from the IR sensor with a tape measure). In each case the calculated distance was within around 30mm of the actual distance.

Working Code:

```
unsigned int rangeIR(void)
{
unsigned int ad_result;
unsigned int range;

ad_result = sampleIR(10);

//Return 0 if there is no target detected
if (ad_result < 100) return 0;

//Convert voltage (0-5v) into range (mm)
range = IR_CONV(ad_result);

return range + calibration_offset_IR;
}
```

0.3.8 SampleIR:

Status:

Working as of 4pm 30/9/2014 - Grant

Description:

This function performs a specified number of reads of the IR sensor and returns the average.

Criteria:

This function has been tested using different ranges for the IR sensor, and each time it has returned the (approximately) correct result.

Working Code:

```
unsigned int sampleIR(char numSamples)
{
unsigned long int sum = 0;
unsigned int temp;
char i = 0;

//Multiplex onto the IR sensor
SetChanADC(ADC_IR_READ);

//Perform numSamples samples
for (i = 0; i < numSamples; i++)
{
ADCON0bits.GO = 1;
while (ADCON0bits.GO_NOT_DONE);
temp = ADRES >> 6;
sum += temp;
}

//Average all samples taken
temp = sum / (unsigned int)numSamples;
return temp;
}
```

0.3.9 rangeUS:

Status:

Working as of 13/10/2014 - Grant

Description:

This function performs a read of the Ultrasonic sensor (if not already started), calibrates it and returns the result.

Criteria:

This function has been tested over a number of different ranges, and a calibration curve fitted as seen in Fig. 3.

Working Code:

```
unsigned int rangeUS(unsigned char temp)
{
unsigned int range;
unsigned long int t;
//Continue to poll while measurement is still in progress
while (measuringUS);
```

```
if (CCPR1 < 0x1770) return 0;

//Perform calculation (ReadCapture in us, speed of sound in m/s->um)
// um/1024 = ~mm

t = DIV_4096((unsigned long int) CCPR1 * (unsigned long int) 285) - 18;

range = (unsigned int) t;

return range;
}
```

0.3.10 ConfigureAD:

Status:

Working as of 4pm 30/9/2014 - Grant

Description:

This function configures the ADC so that it can subsequently be used (immediately after).

Criteria:

There are few testing methods that can be applied to this function in isolation, but after running this function the ADC sampling functions are running properly, so we assume that it is correct. Any future ADC problems should begin with an analysis of this function.

Working Code:

```
void configureAD(void)
{
int i = 0;
TRISA = 0xFF;

//Write the configuration values into the configuration registers
ADCON1 = 0x8E;
ADCON0 = 0x41;

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

0.3.11 BeginUS:

Status:

Working as of 12:30pm 14/10/2014 - Grant

Description:

This function begins an ultrasonic scan and returns. The returning echo will be serviced by an interrupt and the value will be stored in the range module for the next rangeUS call.

Criteria:

This function was deemed to be working correctly as the ultrasonic read is successfully initiated, which is evidenced by the correct result in other subsequent Ultrasonic functions. E.g. the entire rangeUS function is working, and matching measured data.

Working Code:

```
void beginUS(void)
{
CCPR1H = 0;
CCPR1L = 0;
//Set the INIT_PIN high to begin ultrasonic 'read'
INIT_PIN = 1;

//Clear the timer, so the CCP input value is the delay
TMR1H = 0;
TMR1L = 0;

PIE1bits.CCP1IE = 1;

//Set the measuring flag
measuringUS = 1;
}
```

0.3.12 rangeISR:

Status:

Untested as of 12pm 26/9/2014 - Grant

Description:

This function acts as the service routine for the range module. The exact function will depend on the implementation of the range module and what interrupts are used.

Criteria:

0.4 Tracking

0.4.1 ConfigureTracking:

Status:

Currently incomplete and untested as of 12:15pm 26/9/2014 - Grant

Description:

This function configures the pan tilt mechanism (which is acting as the base) so that it is ready for initial use.

Criteria:

0.4.2 Search:

Status:

Currently untested as of 12:15pm 26/9/2014 - Grant

Description:

This function increments the pan tilt mechanism in a search pattern.

Criteria:

0.4.3 trackingISR:

Status:

Untested as of 1:20pm 26/9/2014 - Grant

Description:

This function acts as the ISR for the tracking module. The exact functionality will depend on what interrupts are associated with the tracking module, and how they are implemented.

Criteria:

0.4.4 Edge:

Status:

Currently untested as of 1:20pm 26/9/2014

Description:

This function finds the edge of the target in both the azimuth and declination degrees of freedom, and uses that to find (and track) the centre of the target. It could also be used as a rudimentary target verification based on the approximate size of the target.

Criteria:

0.5 Circular Buffers

0.5.1 Push Macro:

Status:

Working as of 1:45pm 26/9/2014 - Grant

Description:

This macro pushes a byte onto a circular buffer. If the buffer is full then it begins overwriting the first data in the buffer.

Criteria:

This macro has been tested by pushing a byte onto an empty buffer, a partially filled buffer and a completely filled buffer. In each case the result was as documented.

Working code:

```
#define push(byte, buf) buf.data[buf.head] = byte; if(full(buf)) incMod(buf.tail
```

0.5.2 Pop Macro:

Criteria:

Working as of 2:00pm 26/9/2014 - Grant

Description:

This macro pops a byte from a circular buffer. If the buffer is empty it returns whatever is in the first element location and does not change the pointers.

Criteria:

This macro has been tested with both empty and non-empty buffers and in each case it functions as documented.

Working Code:

```
#define pop(buf) buf.data[buf.tail]; if (!empty(buf)) incMod(buf.tail)
```

0.5.3 Init Macro:

Status:

Working as of 2pm 26/9/2014 - Grant

Description:

This macro initialises a buffer before it can be used - It initializes the pointers to zero. This function will also completely clear a circular buffer.

Criteria:

This macro was tested by simply operating on a circularBuffer which was not previously initialized and making sure that it was subsequently initialised.

Working Code:

```
#define init(buf) buf.head = 0; buf.tail = 0
```

0.5.4 Peek Macro:

Status:

Working as of 2:05pm 26/9/2014 - Grant

Description:

This macro returns the next byte that will be popped off the buffer, without removing that element from the buffer. If the buffer is empty it will simply return whatever happens to be in the first element.

Criteria:

This macro was tested by simply operating on a macro several times (between which push and pop operations were implemented) and making sure the correct thing was returned.

Working Code:

```
#define peek(buf) buf.data[buf.tail]
```

0.5.5 Full Macro:

Status:

Working as of 2:05pm 26/9/2014 - Grant

Description:

This macro returns non-zero is the buffer is full, and zero otherwise.

Criteria:

This macro was tested by operating on a number of circular buffers, both full and not full (including empty), and in each case it returned the correct thing.

Working Code:

```
#define full(buf) (buf.tail == (buf.head + 1) % BUFFERLENGTH)
```

0.5.6 Empty:

Status:

Working as of 2:10pm 26/9/2014 - Grant

Description:

This macro returns non-zero if the buffer is empty. Otherwise it returns zero.

Criteria:

This macro was tested on both empty and non-empty buffers, including full buffers. In all cases the macro returned the expected result.

Working Code:

```
#define empty(buf) (buf.head == buf.tail)
```

0.5.7 IncMod Macro:

Status:

Working as of 2:30pm 26/9/2014 - Grant

Description:

This macro increments a buffer pointer and takes the modulus with the buffer length (defined by BUFFERLENGTH), which will loop back to zero, making the buffer circular.

Criteria:

This macro has been tested with a range of values from 0 to BUFFERLENGTH. In each case the macro returned the response as documented.

Working Code:

```
#define incMod(ptr) (ptr = ++ptr % BUFFERLENGTH)
```

0.6 Serial

0.6.1 ConfigureSerial:

Remark:

The baud rate set in this function is dependant on the clock speed of the processor. On the PICDEM board, and other PIC's this is 4MHz. On the minimal board it is a 10MHz clock.

Status:

Working as of 11:25am 1/10/2014 - Grant

Description:

This function sets up and configures the serial UART for communication to the remote terminal.

Criteria:

While there is no way to properly test this function in isolation the serial is working after this function has configured the serial.

Working Code:

```
void configureSerial(void)
{
INTCONbits.GIEH = 1;
INTCONbits.GIEL = 1;
RCONbits.IPEN = 1;

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

//Open the USART module
OpenUSART(USART_TX_INT_ON & USART_RX_INT_ON & USART_BRGH_HIGH & USART_EIGHT_BIT &
//OpenUSART(USART_TX_INT_ON & USART_RX_INT_ON & USART_BRGH_HIGH & USART_EIGHT_BI
}
```

0.6.2 Transmit:

Status:

Working as of 11:20am 1/10/2014 - Grant

Description:

This function pushes a string onto the serial transmit buffer and enables the transmit interrupt to begin transmitting the buffer (if not already transmitting).

Criteria:

This function has been tested by sending strings of different lengths and observing the transmission on terra term. The function appears to be transmitting all strings properly.

Working Code:

```
void transmit(char *string)
{
//Push the string onto the transmit buffer
for (; *string; string++)
{
push(*string, transmit_buffer);
}

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

//Enable TX interrupts
TX_INT_ENABLE();
}
```

0.6.3 transChar

Status:

Working as of 12:30pm 10/10/2014 - Grant

Description:

This function transmits a single character over serial

0.6.4 Criteria:

This function has been tested by testing on several different characters, including just after a transmit function call.

0.6.5 Working Code:

```
void transChar(char c)
{
//Push character onto the transmit buffer
```

```
push(c, transmit_buffer);

//Enable TX interrupts
TX_INT_ENABLE();
}
```

0.6.6 receiveEmpty:

Status:

Working as of 2:50pm 26/9/2014 - Grant

Description:

This function returns non-zero if the receive buffer is empty. Otherwise it returns 0.

Criteria:

This function was tested in isolation by creating a circular buffer and calling the function between push and pop actions to test empty, non-empty and full cases. In each case the the function returned the expected result.

Working Code:

```
char receiveEmpty(void)
{
return empty(receive_buffer);
}
```

0.6.7 receivePeek

Status:

Working as of 2:55pm 26/9/2014 - Grant

Description:

This function applies the peek macro to the receive buffer and returns the result.

Criteria:

The function was tested in isolation on a circular buffer in full, empty and non-empty cases, and each time the function returned the correct result.

Working Code:

```
char receivePeek(void)
{
return peek(receive_buffer);
}
```

0.6.8 receivePop:

Status:

Working as of 3pm 26/9/2014 - Grant

Description:

This function applies the pop macro to the receive buffer and returns the result. Note: the push, pop and init macros are multiple lines and cannot by used in a single return statement like peek and empty.

Criteria:

The function was tested in isolation on a circular buffer in full, empty and non-empty cases. Each time the result was as outlined in the documentation.

Working Code:

```
char receivePop(void)
{
char c = pop(receive_buffer);
return c;
}
```

0.6.9 serialISR:

Status:

Working as of 11:15am 1/10/2014 - Grant

Description:

This function acts as the interrupt service routine for the serial module. The primary purpose will be to handle the receive and transmit interrupts. This function also allows the user to remove previously transmitted characters by popping the last received character from the receive buffer if a backspace is detected. It will not however remove Carriage returns, new lines or escape characters.

Criteria:

This function has been tested with transmit and receive operations. The module has been tested transmitting strings and receiving data, so the ISR function is working as all serial goes through the interrupt.

Working Code:

```
void serialISR(void)
{
unsigned char data;
char last;

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

data = pop(transmit_buffer);
WriteUSART(data);

//Clear interrupt flag
TX_INT_CLEAR();
}
else if (RC_INT)
{
data = ReadUSART();
last = peek(receive_buffer);

//Allows the user to remove/change transmitted data
if (data == BS && last != CR && last != ESC && last != NL )
{
pop(receive_buffer);
}
else
{
//push the received data onto the received buffer
push(data, receive_buffer);
if (data == CR) carriageReturn++;

//Clear interrupt flag
```

```
RC_INT_CLEAR();
}
}
}
```

0.6.10 receiveCR

Status:

Currently untested as of 10:30 1/10/2014 - Grant

Description:

This function returns non-zero if a carriage return has been received. Otherwise it returns zero.

Criteria:

0.6.11 Transmitted

Status:

Currently Untested as of 5pm 10/10/2014 - Grant

Description:

Returns true if all transmission actions have been completed - i.e. if there is nothing in the transmit buffer

Criteria:

0.6.12 readString

Status:

Currently untested as of 10:30 1/10/2014 - Grant

Description:

This function returns all received data up to a carriage return, placing it in the location pointed to by *string.

Criteria:

0.7 Pan Tilt:

0.7.1 SERVO_TOGGLE Macro

REVISION:

In the latest version of the code this macro is no longer included. It is kept in this document for record purposes. Grant 4:10pm 30/9/2014.

Status:

Working as of 12:30 28/9/2014 - Grant

Description:

Toggles the pins used to generate the PWM signal to the servos. There is a SERVO_INIT() macro which initialises the pins, and AZ_PWM_PIN and IN_PWM_PIN macros to pass this this macro. At the moment this macro uses PORTC for the PWM pins. The exact pins in PORTC are determined by the AZ_PWM_PIN and IN_PWM_PIN macros.

Criteria:

This macro was tested with different combinations of AZ_PWM_PIN and IN_PWM_PIN arguments, as well as different starting conditions. In each case the result was as documented.

Working Code:

#define SERVO_TOGGLE(tog) (PORTC = (PORTC & PWM_PINS) — ((PORTC & PWM_PINS) $\hat{}$ (tog)))

0.7.2 ConfigureBase:

REVISION:

This function has changed significantly in the latest version. The previous version is kept for archive purposes.

Status:

Working as of 11:15pm 28/9/2014 - Grant

Description:

Configures the PWM mechanisms to control the pan tilt servo's.

Criteria:

This function has been tested by calling the function and checking the output on RC0 and RC1, the PWM outputs.

Working code:

```
void configureBase(void)
{
unsigned char config;

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

INTCONbits.GIEH = 1;
INTCONbits.GIEL = 1;

RCONbits.IPEN = 1;

SERVO_INIT();

config = T1_16BIT_RW & T1_PS_1_1 & T1_OSC1EN_OFF & T1_SYNC_EXT_OFF & T1_SOURCE_I

OpenTimer1(config);

//Timer1 source for CCP1, and timer3 source for CCP2
SetTmrCCPSrc(T1_CCP1_T3_CCP2);

config = COM_INT_ON & COM_UNCHG_MATCH;

OpenCompare1(config, PWM_PERIOD);

azimuth_angle_max = 45;
azimuth_angle_min = -45;
elevation_angle_max = 45;
elevation_angle_min = -45;
}
```

Working code (OBSOLETE):

```
void configureBase(void)
{
unsigned char config;
DirectionState zero = 14, 0 ;
global_delay = direction2Delay(zero);
INTCONbits.GIEH = 1;
```

INTCONbits.GIEL = 1;
RCONbits.IPEN = 1;
SERVO_INIT();
config = T1_16BIT_RW & T1_PS_1_1 & T1_OSC1EN_OFF & T1_SYNC_EXT_OFF
& T1_SOURCE_INT;
OpenTimer1(config);
SetTmrCCPSrc(T1_CCP1_T3_CCP2);
config = COM_INT_ON & COM_UNCHG_MATCH;
OpenCompare1(config, PWM_PERIOD);
}

0.7.3 panTiltISR:

REVISION:

In the latest version this function has changed significantly - while its function has remained the same. For archive purposes the original code is kept.

Status:

Working as of 11:20pm 28/9/2014 - Grant

Description:

This function acts as the Interrupt Service Routine for the pan tilt mechanism. This function creates the PWM's on the output pins.

Criteria:

This function has been tested on a range of different periods on each of the degrees of freedom.

Working Code:

```
void panTiltISR(void)
{
unsigned int timer_value;
static Delay current_delay;
unsigned char i = 0;

if (CCP1_INT)
{
timer_value = ReadTimer1();

if (timer_value > PWM_PERIOD)
{
IN_PWM_PIN = 1;
```

```
WriteTimer1(0);       //Clear timer2
current_delay = global_delay;    //update the static delay
changed = 1;          //Indicate the change has been loaded
OpenCompare1(COM_INT_ON & COM_UNCHG_MATCH, current_delay.InclinationDelay);
}
else if (timer_value > current_delay.AzimuthDelay)
{
AZ_PWM_PIN = 0;
OpenCompare1(COM_INT_ON & COM_UNCHG_MATCH, PWM_PERIOD);
}
else if (timer_value > PWM_HALF_PERIOD)
{
AZ_PWM_PIN = 1;
OpenCompare1(COM_INT_ON & COM_UNCHG_MATCH, current_delay.AzimuthDelay);
}
else if (timer_value > current_delay.InclinationDelay)
{
IN_PWM_PIN = 0;
OpenCompare1(COM_INT_ON & COM_UNCHG_MATCH, PWM_HALF_PERIOD);
}

PIR1bits.CCP1IF = 0;
}
}
```

Working Code (OBSOLETE):

```
void panTiltISR(void)
{
unsigned int timer_value;
static Delay current_delay;
unsigned char i = 0;

    if (CCP1_INT)
{
timer_value = ReadTimer1();

    if (timer_value ¿ PWM_PERIOD - LATENCY)
{
SERVO_INIT();
WriteTimer1(0);
current_delay = global_delay;
OpenCompare1(COM_INT_ON & COM_UNCHG_MATCH, current_delay.short_delay.delay_time);
}
else if (timer_value ¿ current_delay.long_delay.delay_time)
{
SERVO_TOGGLE(current_delay.long_delay.toggle_bits);
```

```
OpenCompare1(COM_INT_ON & COM_UNCHG_MATCH, PWM_PERIOD);
}
else if (timer_value ¿ current_delay.short_delay.delay_time)
{
SERVO_TOGGLE(current_delay.short_delay.toggle_bits);
for (; i ¡ current_delay.micro_delay.iterations; i++);
SERVO_TOGGLE(current_delay.micro_delay.toggle_bits);
OpenCompare1(COM_INT_ON & COM_UNCHG_MATCH, current_delay.long_delay.delay_time);
}

    PIR1bits.CCP1IF = 0;
}
}
```

0.7.4 direction2Delay:

REVISION:

This function has been changed significantly in the most recent revision. The previous version has been retained in this document for archival purposes.

Status:

Working as of 11:25pm 28/9/2014 - Grant

Description:

This function converts a given Azimuth and Inclination into a delay struct which is then used to create the desired PWM which will result in the given azimuth and inclination.

Criteria:

This function has been tested with a range of different azimuths and inclinations both mathematically, and by inputting the PWM's into the servo's to check the actual position.

Working Code:

```
Delay direction2Delay(DirectionState dir)
{
Delay result;
unsigned int az, inc;

az = 1000 + (dir.azimuth + DIV_2(arcRange.azimuth) + calibration_offset.azimuth)
inc = 1000 + (-dir.inclination + DIV_2(arcRange.inclination) + calibration_offse
```

```
validate(&az);
validate(&inc);

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

return result;
}
```

Working Code (OBSELETE):

```
Delay direction2Delay(DirectionState dir)
{
Delay result;
unsigned int az_delay;
unsigned int in_delay;
unsigned int micro_delay;

    az_delay = dir.azimuth * (unsigned long int)1000 / ARC_RANGE + 1000;
in_delay = dir.inclination * (unsigned long int)1000 / ARC_RANGE + 1000;

    validate(&az_delay);
validate(&in_delay);

    if (az_delay ¡ in_delay)
{
result.short_delay.delay_time = az_delay - LATENCY;
result.long_delay.delay_time = in_delay - LATENCY;
result.short_delay.toggle_bits = AZ_PWM_PIN;
result.long_delay.toggle_bits = IN_PWM_PIN;
}
else
{
result.long_delay.delay_time = az_delay - LATENCY;
result.short_delay.delay_time = in_delay - LATENCY;
result.long_delay.toggle_bits = AZ_PWM_PIN;
result.short_delay.toggle_bits = IN_PWM_PIN;
}

    micro_delay = result.long_delay.delay_time - result.short_delay.delay_time;
if (micro_delay ¡ min_sep)
{
result.short_delay.toggle_bits = AZ_PWM_PIN — IN_PWM_PIN;
result.micro_delay.iterations = 0;
result.micro_delay.toggle_bits = 0;
result.long_delay.delay_time = PWM_PERIOD;
```

```
}
else if (micro_delay ¡ LATENCY)
{
result.micro_delay.iterations = (micro_delay - min_sep) / min_inc;
result.micro_delay.toggle_bits = az_delay ¿ in_delay ? AZ_PWM_PIN : IN_PWM_PIN;
result.long_delay.delay_time = PWM_PERIOD;
}
else
{
result.micro_delay.iterations = 0;
result.micro_delay.toggle_bits = 0;
}

    return result;
}
```

0.7.5 delay2Direction:

REVISION

The code has changed somewhat since 30/9/2014. The obselete code is kept for archival purposes.

Status:

Working as of 2:15pm 14/10/2014 - Grant

Description:

This function converts a delay back into into the direction object that would be used to create it via the direction2Delay function.

Criteria:

This function has been tested by specifying a direction, then converting it into a delay via the direction2delay function. This was then sent to the servos to make sure the delay is correct, and then sent to the delay2Direction function and compared to the original direction object. This was tested for a number of different directions.

Working Code:

```
Direction delay2Direction(Delay dly)
{
Direction ret;
```

```
ret.azimuth = ((dly.AzimuthDelay - DUTY_CYCLE_TIME - PWM_HALF_PERIOD) * (long in
ret.inclination = ((dly.InclinationDelay - DUTY_CYCLE_TIME + LATENCY) * (long in

ret.azimuth = ret.azimuth - DIV_2(arcRange.azimuth) - calibration_offset.azimuth
ret.inclination = -(ret.inclination - DIV_2(arcRange.inclination) - calibration_

return ret;
}
```

Working Code (OBSELETE):

```
DirectionState delay2Direction(Delay dly)
{
DirectionState ret;

    ret.azimuth = ((dly.AzimuthDelay - PWM_HALF_PERIOD - 1000) * (long
int)arcRange.azimuth + 500) / 1000 - DIV_2(arcRange.azimuth) - calibration_offset.azimuth;
ret.inclination = ((dly.InclinationDelay + LATENCY - 1000) * (long int)arcRange.inclination
+ 500) / 1000 - DIV_2(arcRange.inclination) - calibration_offset.inclination;

    return ret;
}
```

0.7.6 Validate:

Status:

Currently working as of 1:30pm 26/9/2014 - Grant

Description:

This function validates a given delay to ensure that it is between 1000ms and 2000ms, coercing it if not.

Criteria:

This function was tested by passing it values of 100, 1500 and 10000, which includes one of each possible case. In each case the function responded appropriately and gave the desired response.

Working Code:

```
void validate(unsigned int *delay)
{
if (*delay < 1000)
{
*delay = 1000;
```

```
}
if (*delay > 2000)
{
*delay = 2000;
}
}
```

0.7.7 move:

Status:

Working as of 10/10/2014 - Grant

Description:

This function will move the pan tilt mechanism to the given inclination and declination.

Criteria:

This code has been extensively tested, and calibrated for this assembly. Calibration data exists somewhere, probably with Bas.

Working Code:

```
void move(Direction destination)
{
global_delay = direction2Delay(destination);
current_direction = destination;
}
```

0.7.8 Increment:

Status:

Currently untested as of 5pm 10/10/2014 - Grant

Description:

This function increments the direction of the pan tilt sensor using the previous direction as a starting position.

Criteria:

0.7.9 IncrementFine:

Status:

Currently untested as of 5pm 10/10/2014 - Grant

Description:

This function increments the direction of the pan tilt sensor using the previous direction as a starting position. The increment argument is not specified in deg, but the smallest resolution available to generate the PWM's, which is around 0.04 deg (see)

0.7.10 getDir

Status:

Untested as of 2:30pm 14/10/2014 - Grant

Description:

Returns the current direction of the pan tilt assembly. Note: This just goes off the current delay, and does not sense the actual position of the servos. Also This value may differ from the commanded position e.g. due to maximum and minimum angle ranges. The function also does not allow time for the servos to move, or for the new delay to be put into the interrupt. The updated function does some of these.

Criteria:

0.7.11 RawDir

Status:

Untested and incomplete as of 2:30pm 14/10/2014 - Grant

Description:

This function returns the raw servo direction, that is without the calibration. This is essentially the same as the getDir function.

Criteria:

0.7.12 Updated:

Status:

Untested as of 2:40pm 14/10/2014 - Grant

Description:

This function returns true if the latest move direction (either a move function, or a increment function call) has been loaded into the interrupt delay. This is because a new set of PDM delays is only loaded into the interrupt at the end of the cycle to ensure the 50Hz frequency.

Criteria:

0.8 MenuSystem:

0.8.1 Initialise Menu

Status:

Untested as of 1pm 14/10/2014 - Grant

Description:

Initialises the menu system for operation when the system is turned on. It sets the menu state to the default setting, and initialises the serial, LCD, and user interface modules.

Criteria:

0.8.2 service Menu

Status:

Currently untested 1pm 14/10/2014 - Grant

Description:

Checks if anything as been inputted to the system, updates the outputs.

Criteria:

0.8.3 menu ISR

Status:

Untested as of 1pm 14/10/2014 - Grant

Description:

Handles any interrupt calls associated with the menu system

Criteria:

0.8.4 parse Input

Status:

Untested as of 1pm 14/10/2014 - Grant

Description:

This function takes any input detected by the system and performs the associated response

Criteria:

0.8.5 parse Numeric:

Status:

untested and incomplete as of 1pm 14/10/2014- Grant

Description:

Parses data a user has inputted into the system into an integer value

Criteria:

0.8.6 State Entry:

Status:

Untested as of 1pm 14/10/2014 - Grant

Description:

Performs actions needed for when the system first enters a menu state

0.9 User Interface

0.10 Interrupts

0.10.1 Description:

This file is not really a module, the interrupt ISR's and vectors have simply been moved into a separate file for readability.

0.10.2 highVector

Status:

Working as of 1:15 pm 14/10/2014 - Grant

Description:

This is a vector function which simply writes a goto statement pointing to the high ISR at the address 0x0008 using pragmas.

Criteria:

This function has been deemed to work as multiple functions requiring ISR's are working, and the interrupts are definitely firing, and calling the ISR's.

Working Code:

```
#pragma code highPriorityInterruptAddress=0x0008
void highVector(void)
{
_asm GOTO highISR _endasm
}
```

0.10.3 lowVector

Status:

Working as of 1:15 pm 14/10/2014 - Grant

Description:

This is a vector function which simply writes a goto statement pointing to the low ISR at the address 0x0018 using pragmas.

Criteria:

This function has been deemed to work as multiple functions requiring ISR's are working, and the interrupts are definitely firing, and calling the ISR's.

Working Code:

```
#pragma code lowPriorityInterruptAddress=0x0018
void lowVector(void)
{
_asm GOTO lowISR _endasm
}
```

0.10.4 lowISR

Status:

Working as of 1:15pm 14/10/2014 - Grant

Description:

This is the ISR for all low priority interrupts on the PIC. This function checks which interrupt has been called, and matches the interrupt to the module which called it. Modules define macros such as SERIAL_INT which are the criteria for one of its interrupts having fired.

Criteria:

This function has been deemed to work because multiple functions requiring ISR's are working, and the interrupts are definitely firing, and calling the ISR's.

Working Code:

```
#pragma interruptlow lowISR
void lowISR(void)
{
if (SERIAL_INT)
{
serialISR();
}
if (PAN_TILT_ISR)
{
panTiltISR();
}
if (RANGE_INT)
{
rangeISR();
}
if (USER_INT)
{
userISR();
}
}
```

0.10.5 highISR

Status:

Working as of 1:15pm 14/10/2014 - Grant

Description:

This is the ISR for all high priority interrupts on the PIC. This function checks which interrupt has been called, and matches the interrupt to the module which called it. Modules define macros such as SERIAL_INT which are the criteria for one of its interrupts having fired.

Criteria:

This function has been deemed to work because multiple functions requiring ISR's are working, and the interrupts are definitely firing, and calling the ISR's.

Working Code:

```
#pragma interrupt highISR
void highISR(void)
{
if (PAN_TILT_ISR)
{
panTiltISR();
}
if (SERIAL_INT)
{
serialISR();
}
if (RANGE_INT)
{
rangeISR();
}
if (USER_INT)
{
userISR();
}
}
```

0.11 LCD