Final Report

Plant Hydrator

ECE3232 Group 12
Chris Ruff 3608564 - 32.5%
Jack Shaw 3608381 - 17.5%
Luke Smith 3623903 - 23%
Reid Hurlburt 3620186 - 27%

Table of Contents

Background Information

Group 12 has determined that there is difficulty in remembering to water plants and significant time being lost in the weathering process. Our goal is to alleviate both issues by creating a device that can automatically water plants for a given interval, called the Plant Hydrator. Group 12 currently believes there is no device on the market to satisfy the needs of the team and are hoping to provide a solution that is not only valuable to themselves, but also a wider market. Hydrating plants can be a mundane and repetitive task, and using a microcontroller and circuit can be automated, giving the user more free time to spend doing other things. The target audience for the device is any typical indoor plant owner, and thus weather proofing will not be required. Design Team 12 is looking only to supply the control mechanism, and not the tubing or water storage solutions. Since this product will involve circuitry potentially around water, and will require set up to function properly, the recommended audience are those ages 16-80. This document acts as a detailed specification of the requirements for the auto-watering device called the Plant Hydrator.

User Manual

Following is a basic description of the needed customer input and steps in order to operate the Plant Hydrator successfully and safely. It is anticipated that the following directions are sufficient information for a user to receive and utilize the system in a properly functional manner, and provide suggestions for where to start should there be any apparent errors in functionality.

Principle of Operation

The Plant Hydrator is an automatic watering system, with the ability to set watering duration and
frequency to provide optimal water to specific house plants.

1. Customer obtains tubes and water reservoir, and initializes set up above the target plant.

2. Consumer connects valve to the previously installed system via the hanging tube

3. Plug in power adapter to enable power to the control unit, set power switch to on to initialize system

4. Use IR remote to input desired watering wait time. This timing increment should be reflected in the 7 segment displays. Only inputs of 4 distinct values will be captured, and if there is a pause of more than 5 seconds the timer will continue

with the previously defined wait duration. The default setting is 50 seconds between watering.

5. A dial on the device controls the amount of time the valve will stay open for and should be adjusted to suit the plant being watered. This will be communicated using the 6 LEDs on the internal electronics. Each LEDs represents 10 seconds, up to a total of 60 seconds. Partial LEDs are allowed but not displayed, meaning that a user is able to set values of time between each interval. E.g. 15 seconds, or 21 seconds, but this is not communicated on the LEDs.

The system is now configured for a standard watering cycle and the 7 segment display should now be showing how much time is left until the next watering cycle. Once the Plant Hydrator enters a watering state, 'POUR' will be shown on the 7 segment display, and a small alarm will sound to notify the consumer. If you would like to change the state of the device from waiting to watering or vice-versa, you can press the manual override button. If the manual override button is pressed to turn from waiting to watering, the system will stop watering once the button is pressed again. A small alarm is played at the end of each watering as well.

Troubleshooting

• If the system does not turn on as intended, ensure there is nothing inhibiting the power transfer from the plug in to the system, ex. Debris in the plug, less than snug connections, etc.

• If when attempting to set wait times with the IR remote the signal is not passing through and there is no resultant change on the displays, ensure there is a clean path for the remote signal to pass, ex. No debris on the remote, nothing sending interference waves nearby, etc.

• If when pressing the manual override button there is no response from the system, ensure the button is being pressed all the way down firmly, and that there is no debris wedged underneath the button surface that could be interfering with the systems ability to read its input.

• Should the system not be watering when expected, or watering longer than expected, attempt to re-input the desired increments of time through either the IR remote for the wait time or the adjustable dial for the water time. Recall that if the dial is turned all the way to the left the reading for watering time will be zero, and that the remote input is the format MM:SS, where M is minutes, and S is in seconds.

• In the case of the valve not closing completely, ensure that there is nothing prohibiting the motor from turning the valve, ex. debris in the motor or valve rotation mechanism.

Safety Warnings

• The tubes and water reservoir are expected to be checked to ensure no water will leak onto the Plant Hydrator control unit.

• The valve must be connected to the tube and fit to ensure there is no leakage when it is closed.

• Keep hands and clothing away from the valve and motor when in operation to avoid pinching and potential injury.

Hardware Block Diagram

Diagram

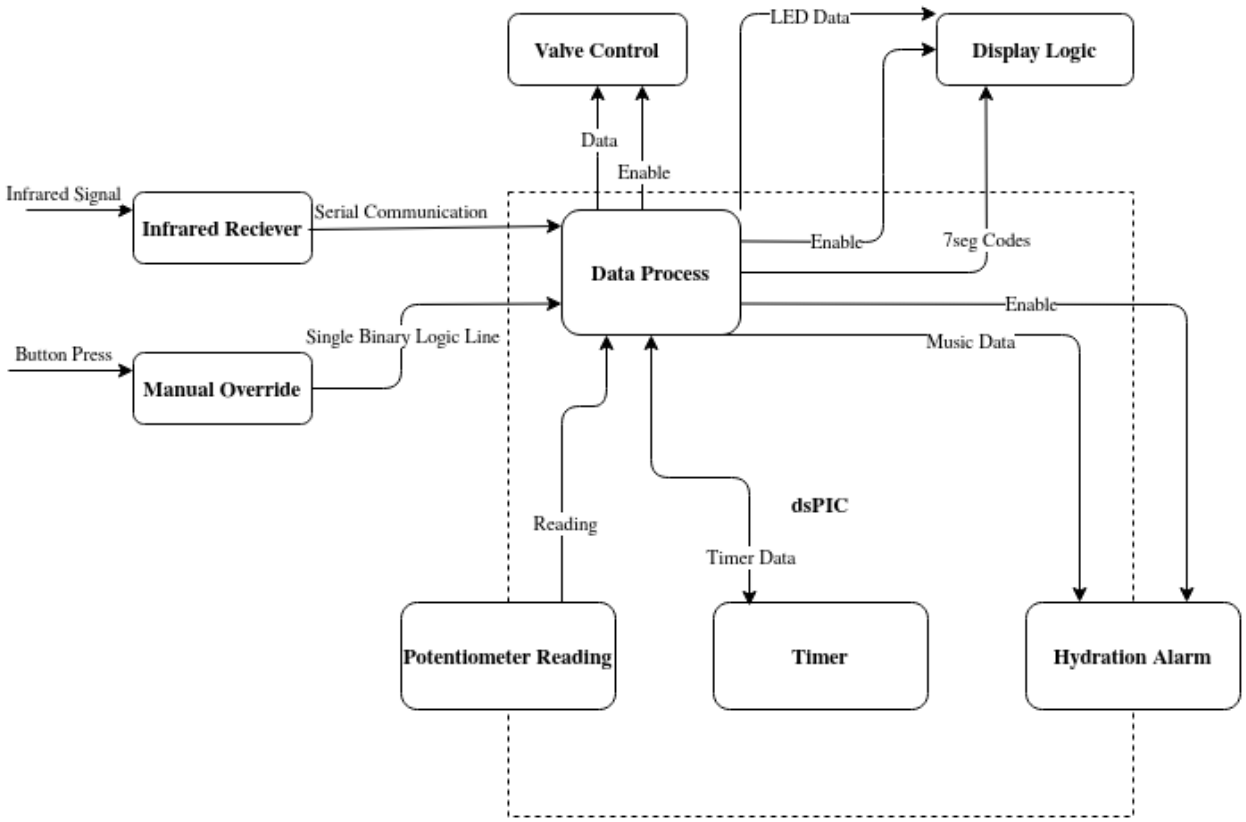

Fig.1: Hardware Block Diagram

Functional Descriptions

IR Block

This block is an IR receiver connected to the board along with an IR remote that will be operated within 5 meters of the receiver. The IR receiver requires three pins, Vcc, ground, and an output pin. The IR receiver will parse the IR remote's number signals and transmit them to the dsPIC chip to be processed using the output pin.

Timer Block

The timer block is integrated into the dsPIC and will be used to track how long the device has been idle since it last dispensed water. The duration of this timer will be determined by the parsed IR input by the dsPIC. The output will be sent back to the dsPIC to be distributed to the LEDs and seven segments. When it is time for the system to dispense water, the timer will be replaced with the duration of pouring which is stored in the dsPIC from the potentiometer. The output will be sent back to the dsPIC to be tracked and displayed on the LEDs.

Potentiometer Reading Block

One of the twenty-four ADC channels will be utilized to convert the output of the 10kΩ potentiometer to a digital input. This analog input pin will be connected to a potentiometer with additional connections to Vcc, and ground.

Display Logic Block

This block is a combination of 7 segment display and 6 red LEDs. There will be 11 wires for the 7 segment display, in addition to a wire for each LED totalling 6, for a combined total of 17 wires. These will be driven directly by the dsPIC microchip as input. The seven segment will count down with the interval until the next water cycle, this will require the seven segment to update each second, and 'Pour'(abcdefg)[0x67, 0x7E, 0x3E, 0x66] while the system is dispensing water. The LEDs will be used to indicate the state of the system. Each will simply require two wires, one being connected to an input pin from the dsPIC and another to a 10kΩ resistor to ground. It will receive current input to illuminate when the system is in the action of pouring water, and have no current flowing otherwise.

Manual Override Block

This block is implemented via a pushbutton, which will require three pins. These pins will be the shared Vcc, the shared ground and a digital input into the dsPIC. Should an active low

value be read (indicating a press of the pushbutton), it will cause the system to change states through the internal logic in the data processing block.

Hydration Alarm Block

The Hydration Alarm block will not require any outside wiring as it is integrated directly via the UNB Dev Board. It will be activated via internal signal from the dsPIC, and can be active at a frequency up to 2.73 kHz via a single tone operating mode. This signal will be generated with a triangle wave passed into the dsPIC's integrated DAC. The DAC itself consists of a PDM unit and a digitally controlled multiphase RC filter. The DACOUT1 pin on the dsPIC is used to transmit this data to the speaker.

Valve Control Block

This block is a servo motor that will be responsible for rotating the valve controlling the water flow. The motor has three wires, a wire connected to Vcc, one connected to ground, and another connected to a PWM pin on the dsPIC that will send a signal to the servo motor. The servo will provide 2.5kg/cm of torque on the valve.

Data Process Block

This block represents the software that Group 12 created for the system. This block is mainly responsible for converting inputs from the various input sources into signals usable for our outputs. This will convert the timer value from its format to a binary value in minutes and hours to be sent to the Binary to 7-Segment Display block. This block will also be responsible for resetting the timer with the next duration it will wait for. There are two values this could be, the digital value from the potentiometer reading block, or the parsed value that was transmitted over the Infrared receiver block. The block will also contain the music data for the Hydration alarm and the hardcoded duty cycles required to rotate the motor for water flow. This block is the hub of all the components and transfers the value from components to each other. Another purpose of the Data Process block is to enable and disable the components when the timer has finished in order to successfully water the plant and display the important information to the user.

Software Block Diagram

Diagram

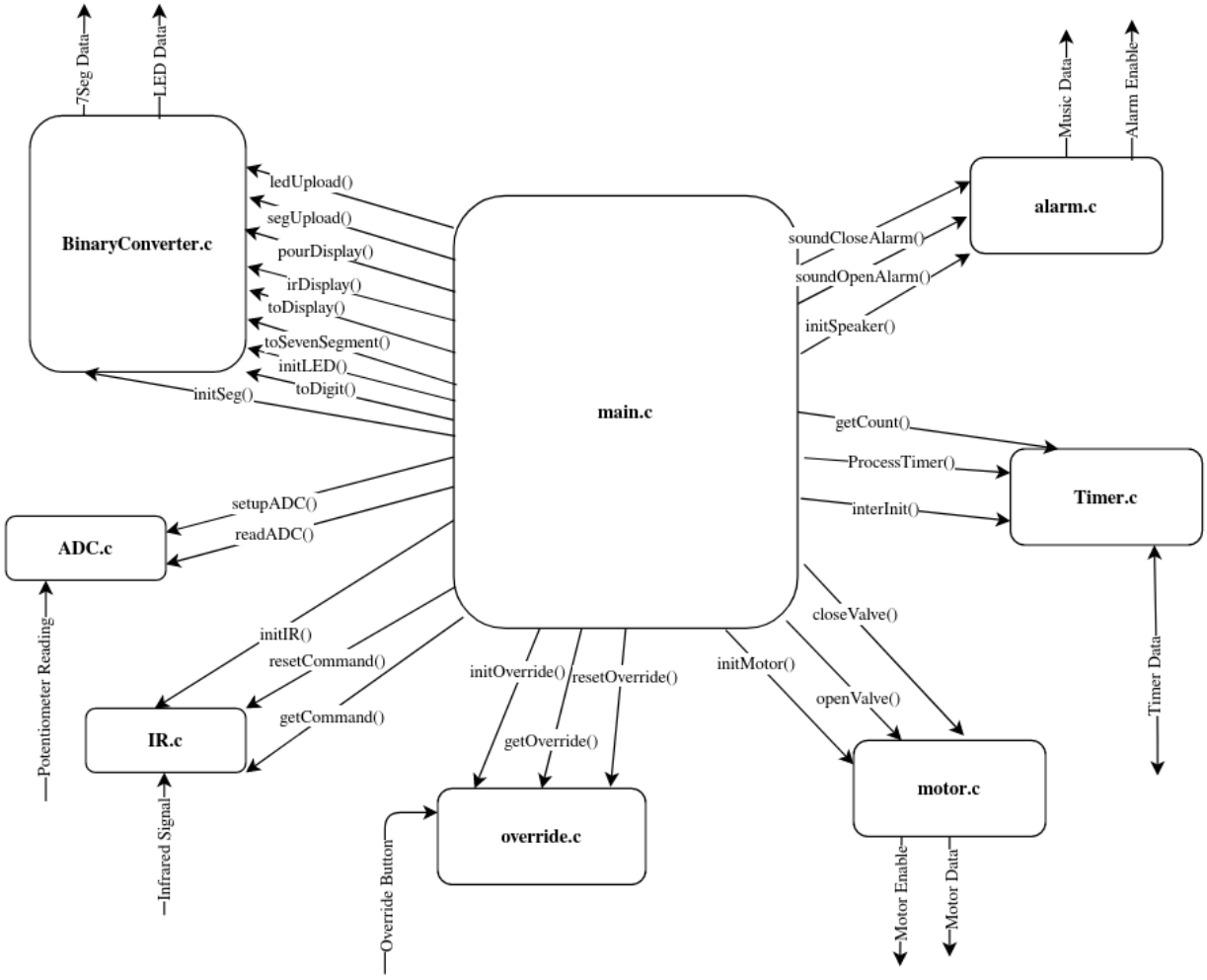

Fig2: Software Block Diagram

Functional Descriptions

BinaryConverter.c

This block is contained within the system software, and is simply a converter of information. It takes the timing information provided by the software through the main.c to timer.c function connections and puts it into a binary form which is readable by a 7-segment display or LED with which it is communicating to have the desired timing information shown as output.

ADC.c

ADC.c allows us to take an instantaneous reading from the potentiometer and pass it back to main.c to set the amount of time the system will water for. The raw data retrieved from the ADC will be divided down to yield a value from 1-6. This will be used to set the watering time to an increment of 10 seconds up to a max of 60 seconds. The current setting will be displayed to the user via 6 LEDs. The ADC must be instantiated to work within the necessary requirements with the setupADC() function. Only then is the readADC() function useful to be called to retrieve the current potentiometer reading translated into the form representing the desired watering time.

IR.c

IR.c is a module which handles the input of data via the IR remote serial communication channel. This is done by first initializing the standard baseline measures and data for the IR modules functionality. This is done via the initIR() function being called by the main.c system controller module. The function getCommand() within the IR.c module is a way of communicating the IR serial information which has been read via the inner module parseIR() function to the main.c entity in a format which is usable for the continuing system functionality. Finally the IR module is reset to a ready to read state via the resetIR() function, clearing the current data which had most recently been read and preparing the module to read again the next serial values which are to be user inputted.

Override.c

Override.c allows the system to be manually overwritten by the user to allow them to open the valve and pour water for however long they wish at the press of a button. The override is controlled by a push button which when pressed, opens the valve and pours water until the button is pressed again to stop the flow and close the valve. To implement and use this

functionality the main.c module must initialize the state of the override module via calling the function initOverride().

Motor.c

The motor module is called from the main.c project module based on the need of the system. The openValve() and closeValve() functions are used to accomplish this, as they contain the logic and motor control information to complete either opening or closing the valve, dependent of course upon the specific function call. Before the motor can be used, the initMotor() function must be called, which will set up the motor's underlying standard functionality, and prepare it to undergo state transitions based on the aforementioned dynamic function calls.

Timer.c

Timer.c provides our system with the capability to control a timer which can indicate either a set amount of time to have the valve open and watering, or a set delay time between waterings. This module is not concerned with which action it is undergoing, but rather exists independent of these states. It accepts a delay value in seconds from main.c and starts a timer for that amount. The timer is controlled by the clock1 module and has a set delay of 1 second. It counts down in these 1 second increments and upon reaching the desired amount of seconds enters an interrupt state which alerts main.c of the system readiness for a state change.

Alarm.c

Alarm.c is the system module which contains the hydration alarm functionality and information. This module must first be called using the initAlarm() function, which sets up the standard baseline speaker information. This baseline data will be used in the other called functions and manipulated as appropriate to produce the desired sound based on each respective function call. Said functions include soundOpenAlarm(), in which Alarm.c manipulates the baseline data to output a sound signifying the opening of the valve, and soundCloseAlarm(), where Alarm.c manipulates the baseline data differently to produce an output sound signifying the closing of the valve.

Description of the System

A general way to look at the functionality of the entire system and the interactions which form it would be that the dsPIC takes information from the input blocks both analog and digital, and processes them in order to determine appropriate signals to send to the output blocks. We can analyze this further by looking into exactly what signals come from each block, how they are processed, and the actions taken in response to them.

The IR block will transmit signals received from the IR remote provided in the kit. This signal will reference certain specifications which the system is to provide, such as change in time increments for wait times and pour times. The dsPIC will take this information and process it to determine what variations must be made to the current timing plan. It will then make these alterations within the timer block of the system, where this information will now be stored and used.

Included in the system will be a potentiometer reading block. This block will contain a potentiometer, whose values will be read directly into the dsPIC. Once this information is contained within the dsPIC it will be processed by the internal ADC which takes the analog input from its corresponding block and parses this data, converting to a manageable digital form. Once converted, this input information is tracked and sent to the variable stored in main.c representing wait time, as well as being used to influence the time span over which the valve control block is active.

The manual override block provides the user with a digital input to impact the functionality of the system. Should the button be pressed it will transition program execution into the alternate state, and signal to such outputs as the motor and digital output blocks to begin or end the water pouring sequence.

The DAC will interpret the desired sound requirements through the dsPIC and manipulate them in such a way that they can be interpreted by the hydration alarm block. This interaction will be triggered via a signal which the dsPIC will develop based on the timer block, as the desired jingle will be signifying the pouring sequence hence the end of another timing cycle.

It is worthwhile to further analyze the output blocks which have been mentioned above. The display logic is a block which includes an LED to signify pouring, a 4 piece 7-seg display for displaying times. The LED is controlled via the data processing block, and specifically by the signals it receives from the timer block, or in some cases the manual override block. Additionally, the 7-seg displays are constantly used to communicate the values of the timer to the user, via the dsPIC blocks signal based on the timer block. The motor-block PWM output is activated via a dsPIC signal which is sent based on the processing of the information being passed by the timer block. Finally, the hydration alarm output is also triggered by the dsPIC as it

is integrated into the board directly, and when it is queued is dependent upon the information passed from the timer block.

Additionally, we should take a step inside the dsPIC and see what the system is doing inside. Essentially, the system operates in a combination of 4 states: waiting, watering, receiving IR command, and the transition state between watering and waiting. In the waiting and watering states, a timer is initially set, then the program will enter a while loop until the timer reaches 0. While the system is in this loop waiting for the timer, it will be continuously checking for a manual override button press. If a button press is detected, it will immediately exit the current state and continue on to the next. In the case a button press is detected while in the wait state, a watering duration value of 1000 seconds is introduced, which encourages the user to press the button again to stop the watering cycle and return to a normal wait cycle. Finally, the receiving IR state will initiate when an IR command is sent and will remain in it until the user has input four numbers or 5 seconds have passed since an input. If the user enters four numbers, they will be used as the new wait time between watering.

System Analysis

To determine the functionality of the system, a few different aspects of the system were analysed. It was determined that for a 5V source, which will be used to power the device, the optimal range of pull-up resistors is 1-5kΩ, with a typical value of 4.7kΩ.

It was determined that the optimal volume of the hydration alarm would be about 55-60db. A normal conversation occurs at about 60db, so this range will be acceptable for the hydration alarm, as to not be a dangerous volume, since volumes of 85db and above can be harmful if exposed to for extended periods of time.

It was observed that the maximum range of operation for the IR remote was 5m. The range can be extended by purchasing an IR extender, which can boost the range, typically in the range of $10-$60 depending on range extension distance.

In the system diagram, it was shown that all signals are routed to the data processing block, which will be the code designed to convert all the required inputs to the desired outputs. It was determined that this would be the best implementation of this part of the system, as it keeps the system more organized and improves overall system efficiency.

Detailed Block Design

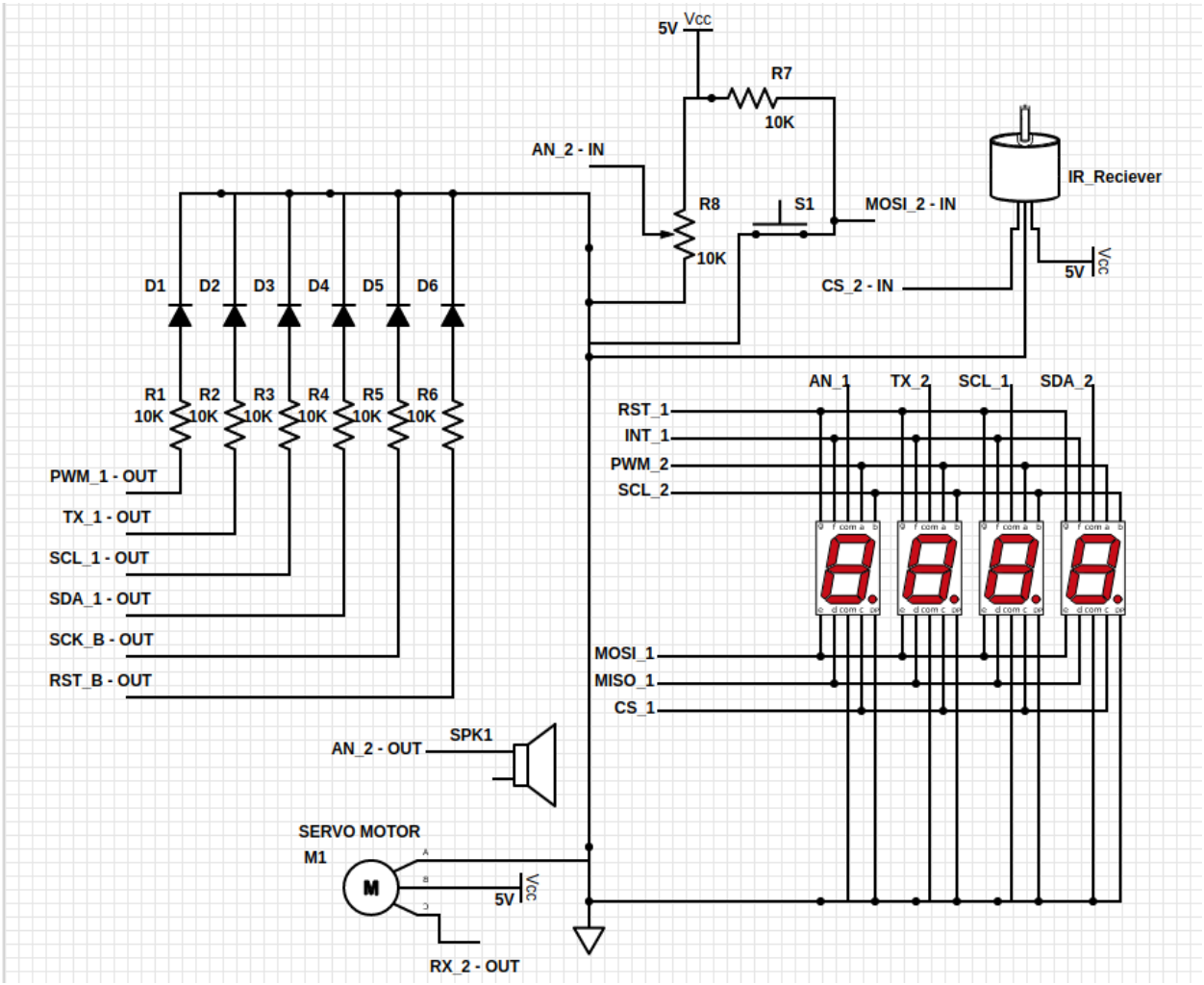

Fig 3: Hardware Schematic DIagram

System Testing

System Test Plan

BinaryConverter module

BC-1: Pour Display

Purpose: Ensure display is correct when pouring.

Input: Enter watering mode(allow display to reach zero).

Expected Result: Seven segment displays “POUR” and LEDs begin counting. down, one for each 10 seconds.

BC-2: IR Display

Purpose: Ensure display is correct when the infrared receiver is getting input.

Input: Enter IR mode(use the remote).

Expected Result: Seven segments display a ‘-’ for each empty digit and fills in while the user enters their value.

BC-3: Regular Display

Purpose: Ensure display is correct when the system is waiting for next watering.

Input: System is counting down regularly.

Expected Result: Seven segments display the duration until next watering and LEDs display the duration of watering.

IR module

IR-1: Minimum Value Case

Purpose: Ensure that the wait time in between watering is essentially 0 seconds

Input: “0000”

Expected result: Zero delay watering

IR-2: Maximum Value Case:

Purpose: Ensure that only the last 4 digits input are captured

Input: “12345678”

Expected result: “5678” captured and processed

ADC module

ADC-1: Minimum Value Case:

Purpose: Ensure that the minimum value retrieved from the ADC is 0x0

Input: Potentiometer turned all the way to the left

Output: Value zero sent to main.c

ADC-2: Maximum Value Case:

Purpose: Confirm that the maximum value retrieved from the ADC is 0xFC0 and said value is seen upon the potentiometer being turned completely to the right

Input: Potentiometer turned all the way to the right
Output: Value of 0xFC0 (4032) sent to main.c

Timer module

Timer-1: Long Duration Test:
Purpose: Ensure the timer can handle a long wait time being set and will be accurate for any extended waiting periods
Input: Wait time of 90 minutes synced with external timer.
Expected result: Completion of 90 minutes simultaneously between timers.
Timer-2: Quick Switch Test:
Purpose: Ensure that the timer can handle an edge case of very fast back to back cycles of watering and waiting
Input: Short ‘wait’ time followed by a short ‘watering’ time to evaluate timer ability to count down one after another in rapid succession
Expected Result: Timer accurately bounces between one wait time and another time seamlessly

Motor module

Motor-1: Open deviation
Purpose: Ensure that when the motor is signalled to close the valve it will do so to a significant degree of accuracy
Input: Begin watering is initiated.
Expected result: position of servo arm is within 2mm of dispensing position every time.
Motor-2: Close deviation
Purpose: Ensure that when the motor is signalled to close the valve it will do so to a significant degree of accuracy
Input: Watering is concluded.
Expected result: position of servo arm is within 2mm of resting position every time.

Manual Override Module

MO-1: Force Water Start
Purpose: Confirm the manual override is effective to force the system into a watering state
Input: Press the button in a ‘wait’ state
Expected Result: Poll input from the button state indicating a press of the button and have that indicator be relayed via the proper channels
MO-2: Force Water Stop
Purpose: Confirm the manual override is effective to force the system into a wait state
Input: Press the button in a ‘watering’ state

Expected Result: Poll input from the button state indicating a press of the button and have that indicator be relayed via the proper channels

System Integration Testing

Int-1: Timer representation

Purpose: Confirm that the main.c module is properly connecting the timer and display logic modules so that the user is being displayed the appropriate timer information

Input: Set a timer wait time value

Expected Result: See the initially set wait time on the 7 segment display and be able to watch it accurately count down in 1 second increments

Int-2: ADC Timer Display Link

Purpose: Confirm that the ADC.c module is properly parsing the inputted values from the potentiometer and that the main.c module is properly connecting this data with the display logic to show the user the correct information based on the input

Input: Manipulate the potentiometer, inputting a value greater than 0

Expected Result: The number of LEDs illuminated during a 'watering' timer cycle should reflect the degree of potentiometer turn (i.e close to 0, few LEDs, as gets closer to a full turn more LEDs)

Int-3: IR Timer Display Link

Purpose: Confirm that the IR.c module is properly parsing the inputted values from the IR remote and that the main.c module is properly connecting this data with the display logic to show the user the correct information based on the input

Input: A 'wait' time value inputted via the IR remote serial communication

Expected Result: See the inputted wait time on the 7 segment display and confirm that it begins to count down, and can be changed with a new input from the IR remote

Int-4: ADC IR Timer Display Link

Purpose: Confirm that the system controller module main.c is operating as intended when receiving data from the analog and serial communication channels, and having this inputted data properly parsed then communicated to the display logic, as well as confirming its ability to perform these operations in a cyclic fashion.

Input: A 'wait' time value inputted from the IR remote in addition to an inputted 'water' time value from the potentiometer

Expected Result: See the inputted wait time on the 7 segment displays, watch it accurately decrement each second, then see the LEDs turn on with the amount of said LEDs illuminated reflecting the degree of rotation of the potentiometer. Watch the number of these illuminated decrement every 10 seconds until completion, then observe this cycle repeating

Int-5: Force Commandments

Purpose: Confirm that the manual override functionality has been properly integrated and initiates the proper sequencing of the overall product

Input: Press the manual override button in a ‘wait’ state to begin watering, then again amidst the watering state to resume the wait

Expected Result: Upon the first press of the button the display logic should switch from showing the remaining wait time on the 7 segments to instead the remaining water time on the LEDs. The valve should open and the opening sound should be projected from the speaker. Upon the second button press, the closing alarm should sound and the valve should close, and the display should transfer back to the 7 segment, now showing a rest wait time

System Test Log

Test Case #	Test Date	Test Result	Notes
BC-1	3/15/21	The seven segment displays ‘POUA’	This is because the Seven segments can’t *really* display R like we had hoped
BC-1_a	3/20/21	The seven segment displays ‘POUR’	
BC-1_b	4/1/21	The LEDs count down to zero based on the potentiometer starting duration	
BC-2	3/20/21	Seven segments display ‘-’ for each empty entry until they are filled	
BC-3_a	3/21/21	Segments count down until 0.	
BC-3_b	4/1/21	LEDs reflect the potentiometer value.	
IR-1	4/5/21	Motor open and closed continuously just giving the motor enough time to open and close	
IR-2	3/28/21	Timer was set to wait for 6039 seconds which could not be displayed.	Fixed with a patch to lower inputs to be within 99

			minutes and 60 seconds
IR-2	3/29/21	Timer was set to wait for 5999 seconds	
Motor-1	4/7/21	Motor opens within 2mm	
Motor-2	4/7/21	Motor closes within 2mm	
ADC-1	3/29/21	ADC yields value of 0x0	Value of 0x0 is captured for the majority of the available potentiometer positions across the left half
ADC-2	3/29/21	ADC yields value of 0xFC0	
Timer-1	3/18/21	Timer decrements at a pace mirroring external timer	
Timer-2	3/18/21	Timer is running concurrently causing overlaps	Must ensure one processes prior to the other
Timer-2	3/19/21	Timer swaps between states non-concurrently	
MO-1	4/4/21	Button reading is bouncing oddly	Consider re-configuring the hardware set up… resistor?
MO-1	4/4/21	Button reading consistent, usable	
MO-2	4/4/21	Button reading consistent, usable	
Int-1	3/29/21	Timer.c counter values are reflected properly in the 7 segment displays	
Int-2	3/30/21	Have analog input, not currently equipped to broadcast	LEDs perhaps?

Int-2	3/31/21	Analog input is reflected in the row of LEDs, counting accordingly	
Int-3	3/31/21	Input of 0x1234 is input through the IR module and also shown on the 7-segment display	
Int-4	4/1/21	Serial input and analog input are parsed concurrently, entering the cycles of the appropriate lengths	
Int-5	4/4/21	Manual override button causes infinite watering cycle	We should implement the manual override to be read directly in main.c
Int-5	4/5/21	Can use button input to signal rest of system to adjust states	

Conclusion

Throughout the development of the Plant Hydrator system, Group 12 ran into various difficulties. In particular, the LCD proved to be very difficult to implement into our project. Without access to an I^2C converter module and excess ports on either MikroBus, it was deemed a low priority feature that was never fully realized, and was ultimately discarded. There were also a number of problems that ended out being successful in the end. When implementing the ADC module, it appeared to us that all the necessary configurations were in place, though no results were coming through. The only thing that actually gave us results was to set all the applicable config registers through hex instead of the built-in 'bits' method. We suspect there were a number of unexpected default values in the ADC config registers that must have only been cleared after setting the whole registers. In addition, the motor was a difficult feature that through much trial and error, was implemented. To overcome the hurdle the clock was divided down to a lower frequency; originally the project used the 32MHz oscillating crystal as the clock, but in order to satisfy the servo motors required frequency it was changed to the FRCDIVN and divided down to a 4MHz clock. This proved to be much more manageable as it could be divided down further in the motor controller to 1MHz and have a 1us period, allowing a PWM period of 20000 cycles to represent the required 20ms period for the servo motor. This allowed the group to move the servo motor to the exact locations required for the valve control.

Overall, all of Group 12 can agree that this experience has been regarded as useful. We feel as though the most important skill learned was how to appropriately digest information from formal data sheets. Since the dsPIC model and the nature of our project is different from typical beginner microcontroller projects, we were unable to get any useful information from online resources. Instead, it forced us to spend hours looking into the details given in the datasheet, which often appears as a daunting task when first starting out. At the end of the project, the documentation that appeared so complex at the start of the project appeared much more understandable.

Appendix A

System Software Source Code:

Globals.h

```
#ifndef GLOBALS
#define GLOBALS

#define FOSC 4000000UL
#define FCY FOSC/2
#define TCY 1/FCY
#include <xc.h>
#include <libpic30.h>

// Define wire placements

#define LED1 LATCbits.LATC13  // PWM on MikroBus A
#define LED2 LATBbits.LATB12  // TX on MikroBus A
#define LED3 LATDbits.LATD3    // SCL on MikroBus A
#define LED4 LATDbits.LATD4   // SDA on MikroBus A
#define LED5 LATCbits.LATC6   // SCK on MikroBus B
#define LED6 LATDbits.LATD11  // RST on MikroBus B


// Define wire placements
#define SEGD1  LATCbits.LATC0  // AN on MikroBus A
#define SEGA   LATCbits.LATC12  // PWM on MikroBus B
#define SEGF   LATCbits.LATC14  // INT on MikroBus A
#define SEGD2  LATBbits.LATB13  // TX on MikroBus B
#define SEGD3  LATBbits.LATB7  // SCK on MikroBus A
#define SEGB   LATCbits.LATC9  // SCL on MikroBus B

#define SEGE   LATBbits.LATB9  // MOSI on MikroBus A
#define SEGD   LATBbits.LATB8  // MISO on MikroBus A
#define SEGC   LATBbits.LATB2  // CS on MikroBus A
#define SEGG   LATCbits.LATC7  // RST on MikroBus A
#define SEGD4  LATCbits.LATC8  // SDA on MikroBus B

typedef struct seg_bits
{
    unsigned int A : 7; // Leftmost seven seg
    unsigned int B : 7;
    unsigned int C : 7;
    unsigned int D : 7; // Rightmost seven seg
}SegBits;

// Define ADC wires
#define ADC_in TRISDbits.TRISD10  // AN on MikroBus B
```

```
#define ADC_an ANSELDbits.ANSELD10

#define IR_IN PORTCbits.RC3 // CS on MikroBus B
#define OVERRIDE PORTCbits.RC1 //MOSI on MikroBus B
#define MOTOR LATBbits.LATB14 // RX on MikroBus B

#define SPEAKER LATDbits.LATD15 // SPEAK_ENABLE
#endif /* GLOBALS */
```

ADC.h

```
#ifndef ADC_H
#define ADC_H

#include "Globals.h"
#include <stdio.h>

void setupADC();
void triggerADC();
unsigned short readADC();

#endif /* ADC_H */
```

ADC.c

```
#include "Globals.h"
#include "ADC.h"

void setupADC()
{
    ADCON1L = 0x8000;
    ADCON1H = 0x0;
    ADCON2L = 0x0;
    ADCON2H = 0x0;
    ADCON3L = 0x0212;
    ADCON3H = 0x4080;
    ADCON4L = 0x0;
    ADCON4H = 0x0;
    ADCON5L = 0x80;
    ADCON5H = 0x80;
    ADMOD0L = 0x0;
    ADMOD0H = 0x0;
    ADMOD1L = 0x0;
    ADMOD1H = 0x0;
    ADIEL = 0x0;
```

```
    ADIEH = 0x0;
    ADC_an = 1;
    ADC_in = 1;
}

void triggerADC()
{
    ADCON3Lbits.CNVRTCH = 1;
}

unsigned short readADC()
{
    triggerADC();
    while(ADSTATHbits.AN18RDY!=1);
    return ADCBUF18 / 67.2;
}
```

Alarm.h

```
#ifndef ALARM_H
#define ALARM_H
#include "Globals.h"

void initSpeaker();
void soundOpenAlarm();
void soundCloseAlarm();

#endif /* ALARM_H */
```

Alarm.c

```
#include "Alarm.h"
void initSpeaker()
{
    //Speaker
    TRISDbits.TRISD15 = 0; // Set as output

    //Configure the source clock for the APLL
    ACLKCON1bits.FRCSEL = 1;
    // Configure the APLL prescalar, APLL feedback divider, and both APLL postscalars
    ACLKCON1bits.APLLPRE = 1; //N1 = 1
    APLLFBD1bits.APLLFBDIV = 125; // M = 125
```

```
    APLLDIV1bits.APOST1DIV = 2; // N2 = 2
    APLLDIV1bits.APOST2DIV = 1; // N3 = 1
    // Enable APLL
    ACLKCON1bits.APLLEN = 1;

    //Setup the DAC
    DACCTRL1Lbits.CLKSEL = 2;
    DACCTRL1Lbits.DACON = 1;
    DAC1CONLbits.DACEN = 1;
    DAC1CONLbits.DACOEN = 1;

    //Use Triangle Waves for the DAC
    SLP1DATbits.SLPDAT = 0x2; // Slope rate, counts per step
    SLP1CONHbits.TWME = 1;
    SLP1CONHbits.SLOPEN = 1;
}
void soundOpenAlarm()
{
    DAC1DATHbits.DACDAT = 0xDFF; // Upper data value
    LATDbits.LATD15 = 1;
    __delay_ms(150);
    LATDbits.LATD15 = 0;
    DAC1DATHbits.DACDAT = 0xD32; // Upper data value
    __delay_ms(150);
    LATDbits.LATD15 = 1;
    __delay_ms(200);
    LATDbits.LATD15 = 0;
    __delay_ms(150);
    LATDbits.LATD15 = 1;
    __delay_ms(200);
    LATDbits.LATD15 = 0;
}
void soundCloseAlarm()
{
    DAC1DATHbits.DACDAT = 0xD32; // Upper data value
    LATDbits.LATD15 = 1;
    __delay_ms(150);
    LATDbits.LATD15 = 0;
    DAC1DATHbits.DACDAT = 0xDFF; // Upper data value
    __delay_ms(150);
    LATDbits.LATD15 = 1;
    __delay_ms(200);
    LATDbits.LATD15 = 0;
    __delay_ms(150);
    LATDbits.LATD15 = 1;
    __delay_ms(200);
    LATDbits.LATD15 = 0;
}
```

BinaryConverter.h

```h
#ifndef BINARY_CONVERTER_H
#define BINARY_CONVERTER_H

#include "Globals.h"

void initSeg();
void initLED();
/**
 * Convert a single 0-9 digit to seven segment format
 * @param in digit to be converted
 * @return 0b0000000 - 0b1111111
 */
int toDigit(char in);

/**
 * Converts value given from the timer to seven segment value
 * @param binary value from the timer 0x00000000 -> 0xFFFFFFFF
 * @return binary value for all the 7segs
 * INT(31-24)(ABCDEFG)1
 * INT(23-16)(ABCDEFG)2
 * INT(15-8) (ABCDEFG)3
 * INT(7-0)  (ABCDEFG)4
 */
SegBits toSevenSegment(int binary);

/**
 * Send timer value to display
 * @param timer binary val
 */
void toDisplay(int time, int duration);

void irDisplay(int* commands, int n);

void pourDisplay();

void segUpload(SegBits segs);
void ledUpload(int duration);

#endif /* BINARY_CONVERTER_H */
```

BinaryConverter.c

```c
#include "BinaryConverter.h"
void initSeg()
{
    TRISCbits.TRISC0 = 0;
```

```
    TRISCbits.TRISC8 = 0;
    TRISCbits.TRISC12 = 0;
    TRISCbits.TRISC9 = 0;
    TRISCbits.TRISC7 = 0;
    TRISBbits.TRISB13 = 0;
    TRISBbits.TRISB7 = 0;
    TRISBbits.TRISB2 = 0;
    TRISBbits.TRISB8 = 0;
    TRISBbits.TRISB9 = 0;
    TRISCbits.TRISC14 = 0;

    ANSELBbits.ANSELB2 = 0; // Disable analog
    ANSELBbits.ANSELB7 = 0;
    ANSELBbits.ANSELB8 = 0;
    ANSELCbits.ANSELC0 = 0;
    ANSELCbits.ANSELC7 = 0;
}

void initLED()
{
    TRISCbits.TRISC6 = 0; // Set pins as output
    TRISCbits.TRISC13 = 0;
    TRISCbits.TRISC15 = 0;
    TRISBbits.TRISB12 = 0;
    TRISDbits.TRISD3 = 0;
    TRISDbits.TRISD4 = 0;
    TRISDbits.TRISD11 = 0;

    ANSELCbits.ANSELC6 = 0;
    ANSELDbits.ANSELD11 = 0;

    LED1 = 0;
    LED2 = 0;
    LED3 = 0;
    LED4 = 0;
    LED5 = 0;
    LED6 = 0;
}
int toDigit(char in)
{
    switch(in)
    {
        case 0:
            return 0b1111110 ^ 127;
        case 1:
            return 0b0110000 ^ 127;
        case 2:
            return 0b1101101 ^ 127;
        case 3:
            return 0b1111001 ^ 127;
        case 4:
            return 0b0110011 ^ 127;
```

```
        case 5:
            return 0b1011011 ^ 127;
        case 6:
            return 0b1011111 ^ 127;
        case 7:
            return 0b1110000 ^ 127;
        case 8:
            return 0b1111111 ^ 127;
        case 9:
            return 0b1111011 ^ 127;
        case 10: // '-'
            return 0b0000001 ^ 127;
        case 11: // 'P'
            return 0b1100111 ^ 127;
        case 12: // 'O'
            return 0b1111110 ^ 127;
        case 13: // 'U'
            return 0b0111110 ^ 127;
        case 14: // 'R'
            return 0b1100110 ^ 127;
        default:
            return 0b1111111 ^ 127;
    }
}

SegBits toSevenSegment(int time)
{
    char A,B,C,D;
    if(time == -1) // "----"
    {
        A = 10;
        B = 10;
        C = 10;
        D = 10;
    }
    else if(time == -2) // "POUR"
    {
        A = 11;
        B = 12;
        C = 13;
        D = 14;
    }
    else
    {
        // Convert binary value to time in seconds based on clock speed
        char mins = time / 60;
        char secs = time % 60;
        A = mins / 10 % 10;
        B = mins % 10;
        //char D = time / 1000 % 10;
        //char C = time / 100 % 10;
        C = secs / 10 % 10;
```

```
        D = secs % 10;
    }


    SegBits outBits;
    outBits.A = toDigit(A);
    outBits.B = toDigit(B);
    outBits.C = toDigit(C);
    outBits.D = toDigit(D);
    return outBits;
}



void toDisplay(int time, int duration)
{
    SegBits segs = toSevenSegment(time);

    segUpload(segs);
    ledUpload(duration);
}
void irDisplay(int* commands, int n)
{
    SegBits segs;
    switch(n)
    {
        case 1:
            segs.D = toDigit(commands[0]);
            segs.C = toDigit(10);
            segs.B = toDigit(10);
            segs.A = toDigit(10);
            break;
        case 2:
            segs.D = toDigit(commands[1]);
            segs.C = toDigit(commands[0]);
            segs.B = toDigit(10);
            segs.A = toDigit(10);
            break;
        case 3:
            segs.D = toDigit(commands[2]);
            segs.C = toDigit(commands[1]);
            segs.B = toDigit(commands[0]);
            segs.A = toDigit(10);
            break;
        default:
            segs.D = toDigit(commands[3]);
            segs.C = toDigit(commands[2]);
            segs.B = toDigit(commands[1]);
            segs.A = toDigit(commands[0]);
            break;
    }
    segUpload(segs);
```

```
}
void pourDisplay(int remaining)
{
    SegBits segs;
    segs.A = toDigit(11);
    segs.B = toDigit(12);
    segs.C = toDigit(13);
    segs.D = toDigit(14);
    segUpload(segs);
    ledUpload(remaining);
}


void segUpload(SegBits segs)
{
    int i;
    for(i = 0; i < 4; ++i)
    {
        switch(i)
        {
            case 0:
                SEGD1=1;
                SEGD2=0;
                SEGD3=0;
                SEGD4=0;
                SEGA = (segs.A & ( 1 << 7-1 )) != 0;
                SEGB = (segs.A & ( 1 << 6-1 )) != 0;
                SEGC = (segs.A & ( 1 << 5-1 )) != 0;
                SEGD = (segs.A & ( 1 << 4-1 )) != 0;
                SEGE = (segs.A & ( 1 << 3-1 )) != 0;
                SEGF = (segs.A & ( 1 << 2-1 )) != 0;
                SEGG = (segs.A & ( 1 << 1-1 )) != 0;
                break;
            case 1:
                SEGD2=1;
                SEGD1=0;
                SEGD3=0;
                SEGD4=0;
                SEGA = (segs.B & ( 1 << 7-1 )) != 0;
                SEGB = (segs.B & ( 1 << 6-1 )) != 0;
                SEGC = (segs.B & ( 1 << 5-1 )) != 0;
                SEGD = (segs.B & ( 1 << 4-1 )) != 0;
                SEGE = (segs.B & ( 1 << 3-1 )) != 0;
                SEGF = (segs.B & ( 1 << 2-1 )) != 0;
                SEGG = (segs.B & ( 1 << 1-1 )) != 0;
                break;
            case 2:
                SEGD3=1;
                SEGD1=0;
                SEGD2=0;
                SEGD4=0;
                SEGA = (segs.C & ( 1 << 7-1 )) != 0;
```

```
                SEGB = (segs.C & ( 1 << 6-1 )) != 0;
                SEGC = (segs.C & ( 1 << 5-1 )) != 0;
                SEGD = (segs.C & ( 1 << 4-1 )) != 0;
                SEGE = (segs.C & ( 1 << 3-1 )) != 0;
                SEGF = (segs.C & ( 1 << 2-1 )) != 0;
                SEGG = (segs.C & ( 1 << 1-1 )) != 0;
                break;
            case 3:
                SEGD1=0;
                SEGD2=0;
                SEGD3=0;
                SEGD4=1;
                SEGA = (segs.D & ( 1 << 7-1 )) != 0;
                SEGB = (segs.D & ( 1 << 6-1 )) != 0;
                SEGC = (segs.D & ( 1 << 5-1 )) != 0;
                SEGD = (segs.D & ( 1 << 4-1 )) != 0;
                SEGE = (segs.D & ( 1 << 3-1 )) != 0;
                SEGF = (segs.D & ( 1 << 2-1 )) != 0;
                SEGG = (segs.D & ( 1 << 1-1 )) != 0;
        }
        __delay_ms(5);
    }
    SEGD1=0;
    SEGD2=0;
    SEGD3=0;
    SEGD4=0;


}
void ledUpload(int duration)
{
    LED1 = 0;
    LED2 = 0;
    LED3 = 0;
    LED4 = 0;
    LED5 = 0;
    LED6 = 0;
    switch(duration/10)
    {
        default:
        case 6:
            LED6 = 1;
        case 5:
            LED5 = 1;
        case 4:
            LED4 = 1;
        case 3:
            LED3 = 1;
        case 2:
            LED2 = 1;
        case 1:
            LED1 = 1;
```

```
        case 0:
            break;
    }
}
```

IR.h

```
#ifndef IR_H
#define IR_H

#include "Globals.h"

#include "BinaryConverter.h"

static int command;
static int waitTime;

int getCommand();
void resetCommand();
void initIR();

// This method was adapted from
//
https://github.com/chayanforyou/NEC-IR-Receiver-PIC/blob/master/NEC%20IR%20Receiver%20PIC16F
877A/NEC%20IR%20Receiver%20PIC16F877A.c
void parseIR();
void __attribute__((interrupt, no_auto_psv)) _CCP1Interrupt( void );


#endif /* IR_H */
```

IR.c

```
#include "IR.h"

int getCommand()
{
    switch(command)
    {
        case 0x6897:
            return 0;
        case 0x30CF:
            return 1;
        case 0x18E7:
```

```
            return 2;
        case 0x7A85:
            return 3;
        case 0x10EF:
            return 4;
        case 0x38C7:
            return 5;
        case 0x5AA5:
            return 6;
        case 0x42BD:
            return 7;
        case 0x4AB5:
            return 8;
        case 0x52AD:
            return 9;
        default:
            return -1;

    }
}
void resetCommand() { command = -1; }
void initIR()
{
    // CS_2 is input
    TRISCbits.TRISC3 = 1;
    ANSELCbits.ANSELC3 = 0;

    // Set PORTC3 to be IC1
    RPINR3bits.ICM1R = 51;

    CCP1CON1Lbits.T32 = 1; // 32 bit capture
    CCP1CON1Lbits.CCPMOD = 1;
    CCP1CON1Lbits.CCPON = 1; // Turn on capture
    CCP1CON1Lbits.CCSEL = 1; // Use input
    CCP1CON2Hbits.ICS = 0;   // Use IC1, which is configured to PORTC3
    CCP1CON2Hbits.AUXOUT = 3; // Input event capture

    // Setup input capture interrupts
    IEC0bits.CCP1IE = 1;
    IFS0bits.CCP1IF = 0;
    IPC1bits.CCP1IP = 7;
    command = -1;
}
void parseIR()
{
    if(command != -1) // If the command hasn't been reset don't overwrite
        return;

    // Check 4.5ms space (remote control sends logic low)
    int c = 0;
    while(IR_IN && (c <= 90)) { c++; __delay_us(50); }
    if((c > 90) || (c < 40)) return;
```

```
    int n;
    for(n = 0; n < 32; ++n)
    {
        // Check for low
        c = 0;
        while(!IR_IN && (c <= 23)){ c++; __delay_us(50); }
        if((c > 22) || (c < 4)) return;

        // Find duration of pulse 500us=0 and 1.3ms=1 high
        c = 0;
        while(IR_IN && (c <= 45)){ c++; __delay_us(50); }
        if((c > 44) || (c < 8)) return;

        // If width is more than 1ms then write 1, else 0
        if(c > 21)
        {
            command |= 1ul << (31-n);
        }
        else
        {
            command &= ~(1ul << (31-n));
        }
    }
}
void __attribute__((interrupt, auto_psv)) _CCP1Interrupt(void)
{
    parseIR();
    IFS0bits.CCP1IF = 0;
}
```

Motor.h

```
#ifndef MOTOR_H
#define MOTOR_H

#include "Globals.h"
#include "Alarm.h"
void openValve();
void closeValve();
void initMotor();

#endif /* MOTOR_H */
```

Motor.c

```c
#include "Motor.h"
void initMotor()
{
    /*PWM control register configuration*/

    /* Motor uses FRCDIVN, we'll set it to 1MHz*/
    CLKDIVbits.FRCDIV = 1; // 8MHz FRC divided by 2 = 4MHz
    PCLKCONbits.DIVSEL = 0b01; // 1:4 divide ratio -> 4MHz / 4 = 1MHz
    PCLKCONbits.MCLKSEL = 0;   // Use FOSC
    PG1CONLbits.CLKSEL = 1;    // Use FOSC

    PG1CONLbits.MODSEL = 0b000; /*Independent edge triggered mode */


    PG1IOCONHbits.PMOD = 1; // Independant mode
    PG1IOCONHbits.PENH = 1; // Controls the PWM1H pin

    /*PWM uses PG1DC, PG1PER, PG1PHASE registers*/
    /*PWM Generator does not broadcast UPDATE status bit state or EOC signal*/
    /*Update the data registers at start of next PWM cycle (SOC) */
    /*PWM Generator operates in Single Trigger mode*/
    /*Start of cycle (SOC) = local EOC*/
    /*Write to DATA REGISTERS*/
    PG1CONH = 0x0000;

    // Period of 20000 * (1/1MHz) = 20ms for servo motor
    PG1PER = 20000;
    PG1DC = PG1PER * 0.50; /* 50% duty - 180 degrees - Start Closed*/
    PG1PHASE = 0; /*Phase offset in rising edge of PWM*/
    PG1DTH = 0; /*Dead time on PWMH */
    PG1DTL = 0; /*Dead time on PWML*/
    PG1CONLbits.ON = 1; // Start PWM
}
void openValve()
{
    soundOpenAlarm();
    PG1CONLbits.ON = 0;
    __delay_ms(5);
    PG1DC = (int)(PG1PER * 0.1); /* 10% duty - 0 degrees*/
    __delay_ms(5);
    PG1CONLbits.ON = 1;
}
void closeValve()
{
    soundCloseAlarm();
    PG1CONLbits.ON = 0;
    __delay_ms(5);
    PG1DC = (int)(PG1PER * 0.50); /* 50% duty - 180 degrees*/
    __delay_ms(5);
    PG1CONLbits.ON = 1;
```

```
}
```

Timer.h

```
#ifndef TIMER_H
#define TIMER_H

#include "Globals.h"

// TIme in seconds on timer
static int countTime = 0;
static int done = 0;

int getCount();

//Receive a value in minutes from main representing time
//Recieve a type value from main, 0 for wait and 1 for water
//Timer counts down the given time, resturns a 1 when done wait
//Returns a 0 when done watering
void ProcessTimer(int time);

//Initialize the timer settings and defaults
void InterInit();

//Interrupt to handle the timer
void __attribute__((interrupt(auto_psv))) _T1Interrupt(void);
```

Timer.c

```
#include "Timer.h"

int getCount() { return countTime; }

void ProcessTimer(int time)
{
    countTime = time;
    done = 0;
    InterInit();
}


void InterInit(){
    T1CONbits.TECS=1;       //Timer1 TCY
    T1CONbits.TCS=1;        //Timer1 use external clock
    T1CONbits.TCKPS=3;      //Timer1 prescale 1:256
```

```
    T1CONbits.TSYNC=0;      //Timer1 Sycnronize the clock
    IFS0bits.T1IF=0;        //reset interrupt flag
    IEC0bits.T1IE=1;        //enable _TimerInterrupt
    TMR1=0x0;               //ensure timer register is cleared
    PR1=FCY/256;             //count period, 1 second delay
    T1CONbits.TON=1;        //Timer1 enable
}

void __attribute__((interrupt(auto_psv))) _T1Interrupt(void){
    if(countTime == 0){
        done = 1;
        IFS0bits.T1IF = 0;
        T1CONbits.TON = 0;
        return;
    }
    countTime --;
    PR1 = FCY/256;
    IFS0bits.T1IF = 0;
}
```

Override.h

```
#ifndef OVERRIDE_H
#define OVERRIDE_H
#include "Globals.h"

void initOverride();

#endif  /* OVERRIDE_H */
```

Override.c

```
#include "Override.h"

void initOverride(void){
      // MOSI_2 is input
    TRISCbits.TRISC1 = 1;
    ANSELCbits.ANSELC1 = 0;
}
```

main.c

```
// DSPIC33CK256MP506 Configuration Bit Settings

// 'C' source line config statements

// FSEC
#pragma config BWRP = OFF               // Boot Segment Write-Protect bit (Boot Segment may
be written)
#pragma config BSS = DISABLED           // Boot Segment Code-Protect Level bits (No
Protection (other than BWRP))
#pragma config BSEN = OFF               // Boot Segment Control bit (No Boot Segment)
#pragma config GWRP = OFF               // General Segment Write-Protect bit (General
Segment may be written)
#pragma config GSS = DISABLED           // General Segment Code-Protect Level bits (No
Protection (other than GWRP))
#pragma config CWRP = OFF               // Configuration Segment Write-Protect bit
(Configuration Segment may be written)
#pragma config CSS = DISABLED           // Configuration Segment Code-Protect Level bits (No
Protection (other than CWRP))
#pragma config AIVTDIS = OFF            // Alternate Interrupt Vector Table bit (Disabled
AIVT)

// FBSLIM
#pragma config BSLIM = 0x1FFF           // Boot Segment Flash Page Address Limit bits (Enter
Hexadecimal value)

// FSIGN

// FOSCSEL
#pragma config FNOSC = FRCDIVN              // Oscillator Source Selection (Primary
Oscillator (XT, HS, EC))
#pragma config IESO = OFF               // Two-speed Oscillator Start-up Enable bit (Start
up with user-selected oscillator source)

// FOSC
#pragma config POSCMD = HS              // Primary Oscillator Mode Select bits (HS Crystal
Oscillator Mode)
#pragma config OSCIOFNC = OFF           // OSC2 Pin Function bit (OSC2 is clock output)
#pragma config FCKSM = CSDCMD           // Clock Switching Mode bits (Both Clock switching
and Fail-safe Clock Monitor are disabled)
#pragma config PLLKEN = ON              // PLL Lock Status Control (PLL lock signal will be
used to disable PLL clock output if lock is lost)
#pragma config XTCFG = G3               // XT Config (24-32 MHz crystals)
#pragma config XTBST = ENABLE           // XT Boost (Boost the kick-start)

// FWDT
// RWDTPS = No Setting
#pragma config RCLKSEL = LPRC           // Watchdog Timer Clock Select bits (Always use
LPRC)
#pragma config WINDIS = ON              // Watchdog Timer Window Enable bit (Watchdog Timer
operates in Non-Window mode)
```

```
#pragma config WDTWIN = WIN25           // Watchdog Timer Window Select bits (WDT Window is
25% of WDT period)
// SWDTPS = No Setting
#pragma config FWDTEN = ON              // Watchdog Timer Enable bit (WDT enabled in
hardware)

// FPOR
#pragma config BISTDIS = DISABLED       // Memory BIST Feature Disable (mBIST on reset
feature disabled)

// FICD
#pragma config ICS = PGD2               // ICD Communication Channel Select bits
(Communicate on PGC2 and PGD2)
#pragma config JTAGEN = OFF             // JTAG Enable bit (JTAG is disabled)
#pragma config NOBTSWP = DISABLED       // BOOTSWP instruction disable bit (BOOTSWP
instruction is disabled)

// FDMTIVTL
#pragma config DMTIVTL = 0xFFFF         // Dead Man Timer Interval low word (Enter
Hexadecimal value)

// FDMTIVTH
#pragma config DMTIVTH = 0xFFFF         // Dead Man Timer Interval high word (Enter
Hexadecimal value)

// FDMTCNTL
#pragma config DMTCNTL = 0xFFFF         // Lower 16 bits of 32 bit DMT instruction count
time-out value (0-0xFFFF) (Enter Hexadecimal value)

// FDMTCNTH
#pragma config DMTCNTH = 0xFFFF         // Upper 16 bits of 32 bit DMT instruction count
time-out value (0-0xFFFF) (Enter Hexadecimal value)

// FDMT
#pragma config DMTDIS = OFF             // Dead Man Timer Disable bit (Dead Man Timer is
Disabled and can be enabled by software)

// FDEVOPT
#pragma config ALTI2C1 = OFF            // Alternate I2C1 Pin bit (I2C1 mapped to SDA1/SCL1
pins)
#pragma config ALTI2C2 = OFF            // Alternate I2C2 Pin bit (I2C2 mapped to SDA2/SCL2
pins)
#pragma config ALTI2C3 = OFF            // Alternate I2C3 Pin bit (I2C3 mapped to SDA3/SCL3
pins)
#pragma config SMBEN = SMBUS            // SM Bus Enable (SMBus input threshold is enabled)
#pragma config SPI2PIN = PPS            // SPI2 Pin Select bit (SPI2 uses I/O remap (PPS)
pins)

// FALTREG
#pragma config CTXT1 = OFF              // Specifies Interrupt Priority Level (IPL)
Associated to Alternate Working Register 1 bits (Not Assigned)
#pragma config CTXT2 = OFF              // Specifies Interrupt Priority Level (IPL)
```

```
Associated to Alternate Working Register 2 bits (Not Assigned)
#pragma config CTXT3 = OFF              // Specifies Interrupt Priority Level (IPL)
Associated to Alternate Working Register 3 bits (Not Assigned)
#pragma config CTXT4 = OFF              // Specifies Interrupt Priority Level (IPL)
Associated to Alternate Working Register 4 bits (Not Assigned)

// FBTSEQ
#pragma config BSEQ = 0xFFF             // Relative value defining which partition will be
active after device Reset; the partition containing a lower boot number will be active
(Enter Hexadecimal value)
#pragma config IBSEQ = 0xFFF            // The one's complement of BSEQ; must be calculated
by the user and written during device programming. (Enter Hexadecimal value)

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.


#include "BinaryConverter.h"
#include "Timer.h"
#include "IR.h"
#include "Motor.h"
#include "ADC.h"
#include "Override.h"
#include "Alarm.h"


void getCommands();
void waterWait(int duration);
void wait(int duration);

static int waitTime = 60;

int main(void)
{
    initIR();
    initSeg();
    initLED();
    setupADC();
    initMotor();
    initOverride();
    initSpeaker();

    while(1)
    {
        wait(waitTime);

        if (!OVERRIDE)
        {
            resetOverride();
            while(!OVERRIDE);
            waterWait(1000);
            while(!OVERRIDE);
```

```
        }
        else
        {
            int duration = readADC();
            waterWait(duration);
        }
    }

}

void wait(int duration)
{
    ProcessTimer(duration);
    int c;
    int flag = 0;
    do
    {
        // Check for manual override
        if(!OVERRIDE)//getButtonPressed())
        {
            return;
        }

        if(getCommand() != -1)
        {
            getCommands();
        }
        else
        {
            c = getCount();
            toDisplay(c, readADC());
        }
    }while(c > 0);
}
void waterWait(int duration)
{
    openValve();
    ProcessTimer(duration);
    int c;
    do
    {
        // Check for manual override
        if(!OVERRIDE)
        {
            closeValve();
            return;
        }
        if(getCommand() != -1)
        {
            getCommands();
        }
        else
```

```
        {
            c = getCount();
            pourDisplay(c);
        }
    }while(c > 0);
    closeValve();
}

void getCommands()
{
    int commands[4];
    int i;
    int curTime = getCount();
    for(i = 0; i < 4; i++)
    {
        ProcessTimer(5);
        int c;
        while(getCommand() == -1)
        {
            irDisplay(commands, i);
            c = getCount();
            if(c <= 0)
            {
                ProcessTimer(curTime);
                return;
            }
        }
        commands[i] = getCommand();
        resetCommand();
    }
    irDisplay(commands,4);
    __delay_ms(500);
    waitTime = (commands[0]*10 + commands[1])*60 + (commands[2]*10 + commands[3]);
    if(waitTime >= 6000)
       waitTime = 5999;


    ProcessTimer(waitTime);
}
```