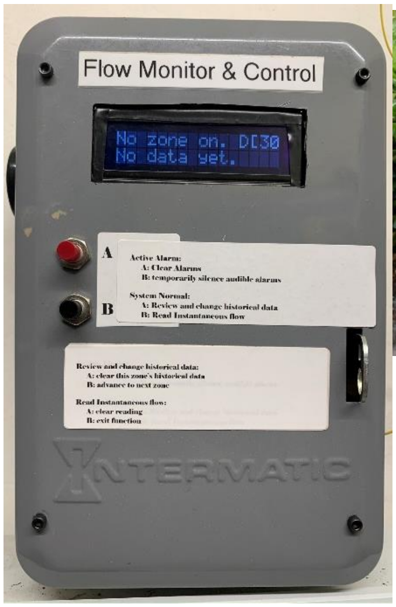
Irrigation Flow Monitor and Control System, Version 2.3.1

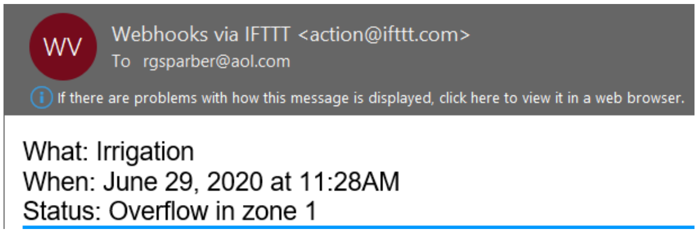
By R. G. Sparber

Conclusion



This package of hardware and software can stop water wasted due to irrigation leaks and the death of plants due to insufficient water.

Within one minute, it will detect and shut down any zone with a flow rate greater than 30% above the average. It will also alarm if the flow is less than 30% below the average.



Webhooks via IFTTT <action@ifttt.com>
To rgsparber@aol.com
If there are problems with how this message is displayed, click here to view it in a web browser.

What: Irrigation
When: June 29, 2020 at 11:28AM
Status: Overflow in zone 1

Besides sounding an audible alarm, it sends an email to the user. This feature would be particularly valuable for irrigation companies responsible for residential sites and for “snowbirds” that are far from their property being irrigated.

One blown or clogged line can cost more than the price of the Flow Monitor and Control.

I built this system for under $150. Approximately $80 of this was for commonly available parts. The challenge is to find an accurate flow meter at a reasonable cost. I used a modified second hand Badger Model 25 flow meter. This is fine for a hobbyist but not as part of a product.

My Goal

I want someone or some company to pick up this design and turn it into a product. Developing the hardware and software is 10% of the effort. Product development, packaging, marketing, distribution, and product support make up the other 90%. In exchange for this effort, they are free to keep all profits. I just want to see a significant reduction in wasted water.

Background

I own a very smart Irrigation Controller from RainBird. It is aware of the weather and adjust the run time of each of my 6 zones accordingly.

However, this controller is missing an important feature:

***the ability to monitor and act on flow rates*.**

If a valve sticks closed or open it can’t tell. If a pipe bursts, it can’t tell. If a single “spaghetti line” blows its emitter, can’t tell. Worse yet, I can’t quickly shut off the water.

All of that changed with my Flow Monitor and Control (FMC) installed.

See https://www.youtube.com/watch?v=fuf6UD6G-A8&t=94s and https://www.youtube.com/watch?v=oRK1sAZ7rfI&t=73s for videos of the system in action.

Up until now, I have compensated for the lack of automatic flow control by manually recording my water usage on my city water meter. Since most of my usage is due to irrigation, this does at least tells me something is wrong. I must then sequence through all 6 zones and inspect the area of the property until I find the one with the problem.

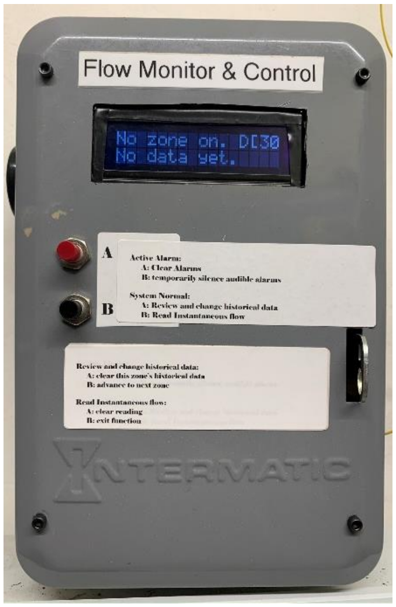
In one case, I had a significant leak that left no water on the surface. My monthly bill told me something was wrong when it indicated 20,000 gallons of unusual consumption. That $300 bill was a big motivator for finding an automatic method of dealing with flow problems.

RainBird does sell a flow monitoring system, but it is targeted for the commercial market and is priced accordingly.

This article presents the design of my system, including system block diagrams, schematics, flow charts, and software. It does ***not*** show the many dead ends and re-writes needed to get to what was finally built.

I have included sufficient detail, so someone with some hardware background could build the system. Contact me to receive the current software.

User’s Manual



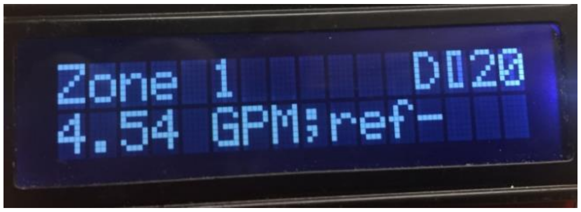
Under normal conditions, the user does nothing. If the beeper sounds, the screen will tell the user which zone has a problem and what is wrong.

If the Flow Monitor and Control was able to stop the flow of a faulted zone, the user should inspect the zone, correct the fault, and then clear the alarm.

If the flow cannot be stopped, the user should turn off the main water supply. The problem will be between the water meter’s output and the valve’s output. A broken pipe will be obvious. If a valve is unable to shut down, you will see water flowing in the zone when the controller says it is off.

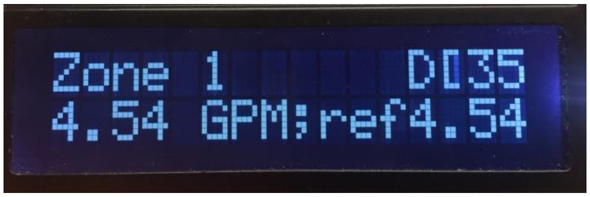
All alarms trigger an email to the user at the time of the fault. To verify the reporting path is functional, a heartbeat message is sent at sunrise each day. Power up and Maintenance operations also cause an email to be sent.

System Normal



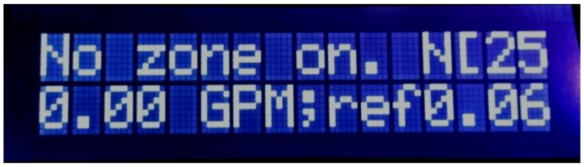
```
Zone 1      D[20
4.54 GPM:ref-
```

When The Flow Monitor and Control is first powered up, it records the average water use for each zone. Here you see Zone 1 is active, and the flow is 4.54 Gallons Per Minute (GPM). These measurements are saved as the first reference flow[2] when Zone 1 is turned off by the Irrigation Controller.



```
Zone 1      D[35
4.54 GPM:ref4.54
```

When a given zone is rerun, its flow rate is compared to the reference. In this case, the current flow matches the flow the last time this zone ran.

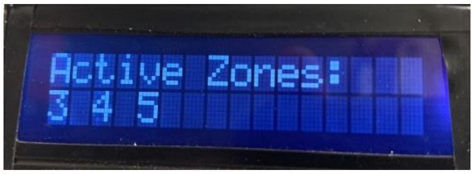


```
No zone on. N[25
0.00 GPM:ref0.06
```

When no zones are on, the flow is compared to a limit[3] of 0.06 GPM. Flow less than 0.06 GPM is considered acceptable.

[2] It is therefore essential that each zone is initially in proper working order. See page 9 for how to audit each zone.

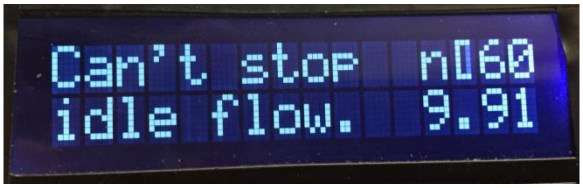
[3] All readings are ± 0.03 GPM so this is the smallest limit we can have and not get false alarms.



```
Active Zones:
3 4 5
```

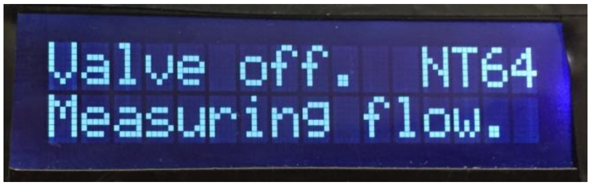
If more than one zone is on, no flow information is displayed. A short in the cable out to the valves would cause this condition. The RainBird controller does not run more than one zone at a time but it might be possible to do manually.

Fault Situations



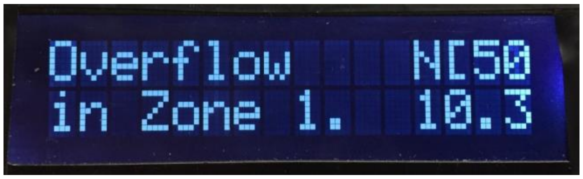
```
Can't stop   n▯60
idle flow.   9.91
```

When no zones are on, and the flow exceeds the limit of 0.06 GPM, we get an Overflow condition, and a Major Alarm is sounded both day and night. It will sound for 0.1 seconds and be quiet for 0.1 seconds. The number in the lower right-hand corner is the current flow. It is possible to see “Can’t stop idle flow.” with a current flow of 0. This means it ***was*** greater than 0.06 GPM but is ***now*** zero.



```
Valve off.   NT64
Measuring flow.
```

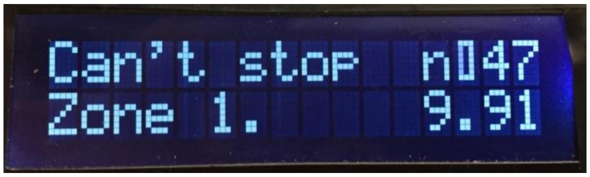
If the new flow is more than 30% higher than the reference, we get an Overflow condition. The system responds by turning off the valve and taking another flow measurement.



```
Overflow     N[50
in Zone 1.   10.3
```

If the flow is stopped, the screen indicates the problem. If it is daytime, the Flow Monitor and Control sounds a Minor Alarm, which is 0.3 seconds on and 2 seconds off. The flow that triggered the alarm is in the lower right corner.

Other zones can run, and they will be monitored. If the flow can’t be stopped, a Major Alarm will sound. Any time an Overflowed zone with Minor Alarm runs, its flow will be stopped.

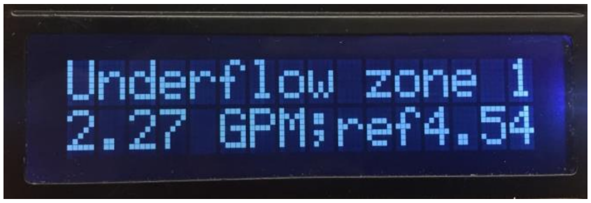


```
Can't stop   n▯47
Zone 1.      9.91
```

If the Flow Monitor and Control was unable to stop the flow[4], the screen tells the story, and a Major Alarm is sounded. The current flow is in the lower right corner.

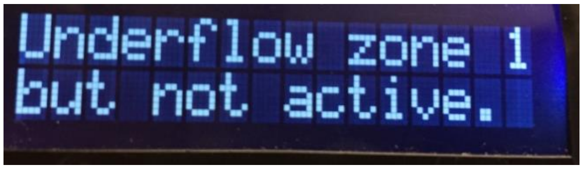
The Irrigation Controller can run other zones, but they will be blocked as the Flow Monitor and Control continues to try and stop the leak. I have seen this fault when a valve was stuck open.

[4] 6.80 GPM is 9,800 gallons per day!



```
Underflow zone 1
2.27 GPM;ref4.54
```

If the flow is more than 30% lower than the threshold, we get an Underflow condition. The second line shows the current flow and average flow (ref). If during the day, a Minor Alarm is sounded.

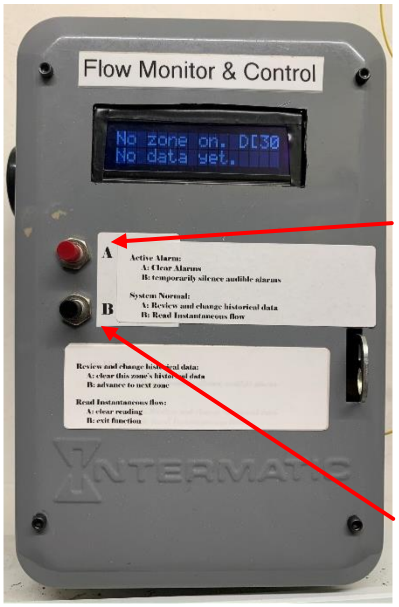


```
Underflow zone 1
but not active.
```

If the underflowed zone is not active, the system will monitor and control the other zones while displaying the error message for the faulted zone.

If you hear two chirps every two seconds plus “AkF”, “WfF”, or “MkF” displayed in the upper right corner of the screen, it means[5] the ability to send status and alarm emails has been lost at least in the recent past. Since there is no way to know, at a distance, if another alarm is active, these chirps will sound even when audible alarms were turned off. The “alarm clear” and “audible alarms temporarily silenced” functions described below do act on this failure. However, when you silence the alarm, it stays silent until the alarm is cleared. This is because it might be a long time before someone can deal with the problem. If the alarm is cleared but the failure is still there, the alarm will return within an hour.

Controlling Alarms



The following applies when there is an active alarm.

By pushing the “A” button, all alarms will be cleared. If the problem is still present, the alarm will come back after 70 seconds.

By pushing the “B” button, any audible alarm will be temporarily silenced. They will return if the problem has not

[5] AkF means the acknowledgement expected from the processor that sends out emails has doed. WfF means WiFi was down the last time an email was sent out. MkF means the “Maker” was down the last time an email was sent out. The Maker translates the message from the Flow Monitor and Control into an email and is a web based service.

been resolved and cleared in time. For a Major alarm, expect it to sound again in 15 minutes. Minor alarms will sound again after 24 hours. Fault information will remain on the screen. See page 8 for how to permanently silence the audible alarm.

Automatic Alarm Retirement

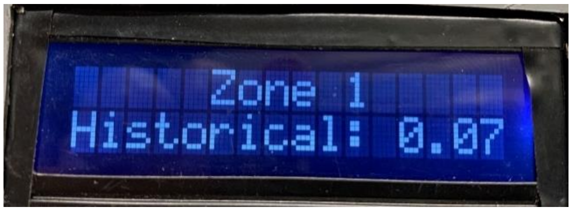
Since all alarms are registered with the owner via email, they are cleared each day six hours after sunrise with no loss of information.

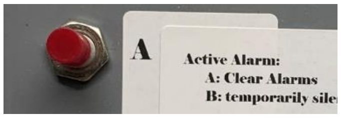
Automatically clearing alarms avoids the problem of zones going out of service due to large fluctuations in city water pressure.

Maintenance Function

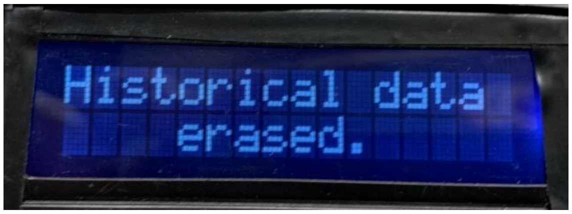
The following applies when there are no active alarms.

Review and Clearing Historical Flow Data

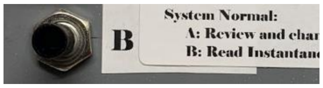

Zone 1
Historical: 0.07



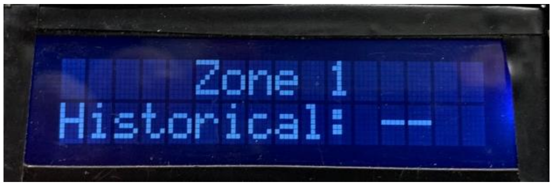
Pressing the A button, you will see the historical flow data for Zone 1.


Historical data
erased.

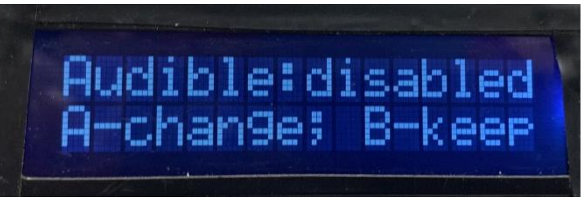
If you press the A button again, it will erase this zone’s data.



If you press the B button, you will advance to the next zone. Repeated pressing of the B button will sequence through all of the zones, and you will then return to the normal screen.


Zone 1
Historical: --

If the historical data was previously cleared and that zone has not yet run, you will see --

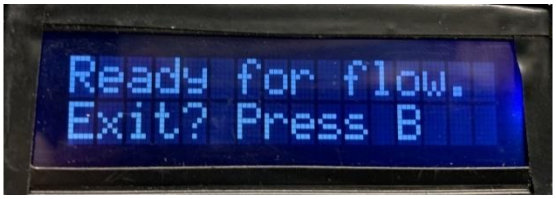

Audible:disabled
A-change; B-keep

After all zones have been displayed, you get the chance to enable or disable the audible alarm. This option is also presented when the system power us.

If there is no action within five minutes, the system returns to normal operation.

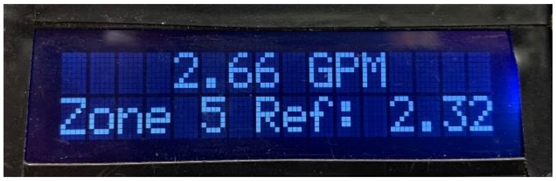
The following applies when there are no active alarms.

Measuring Instantaneous Flow



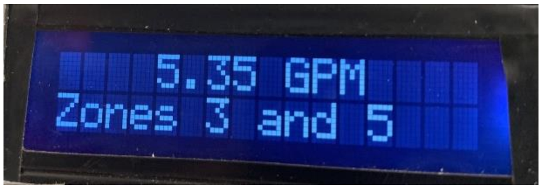
```
Ready for flow.
Exit? Press B
```

From the normal screen, pressing the B button enables you to see the current instantaneous[6] flow rate. Before any flow is detected, you get this screen. Press the A button at any time to return to this screen.



```
    2.66 GPM
Zone 5 Ref: 2.32
```

Once a non-zero flow rate is measured, the active zone and reference (historical) flow are also displayed. They will update every second. If the flow returns to 0, the last non-zero flow rate will remain on the screen. Pressing the A button sends you back to the Ready for Flow screen.



```
    5.35 GPM
Zones 3 and 5
```

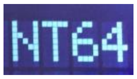
If two zones are active, the historical flow rate is replaced with the second zone's number.

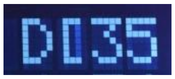
If more than two zones are active, you will see `>2 zones active`

If there is no action within five minutes, the system returns to normal operation.

[6] Normally, flow is measured over one minute intervals so small variations are averaged out. This flow measurement is instant. That is good for seeing what the flow is doing every second.

Status Indicators


NT64

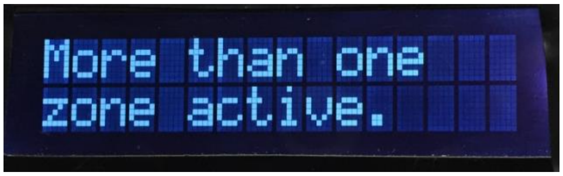

D[35

On most screens you will see one or two letters and one or two numbers. This is the status indicator. It is a sanity check to be sure all is working. The first letter can be **N** for nighttime with no audible alarms silenced. If **n**, it is nighttime and a silenced alarm is active. A **D** means daytime with no audible alarms silenced. A **d** is daytime and a silenced alarm is active.

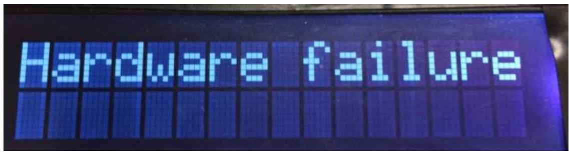
The next character will be a **T** when we are waiting for the flow to build up and stabilize. **[** means the flow meter has not indicated a unit of water[7] has passed. A **]** means the flow meter just saw a unit of water pass. The result is that you see **[** with the right side flickering into a rectangle when there is flow. The higher the flow rate, the closer you get to a constant rectangle.

D[35

The two numbers are a countdown until a new flow measurement is complete. If a zone just turned on, the count will go from 70 seconds down to 60 and the **T** will precede the count. At 60 you will see **[** and the count will decrement until it reaches 0. Then the flow information will update.


More than one
zone active.

If the user chooses to manually operate more than one zone at the same time, the flow is not monitored.

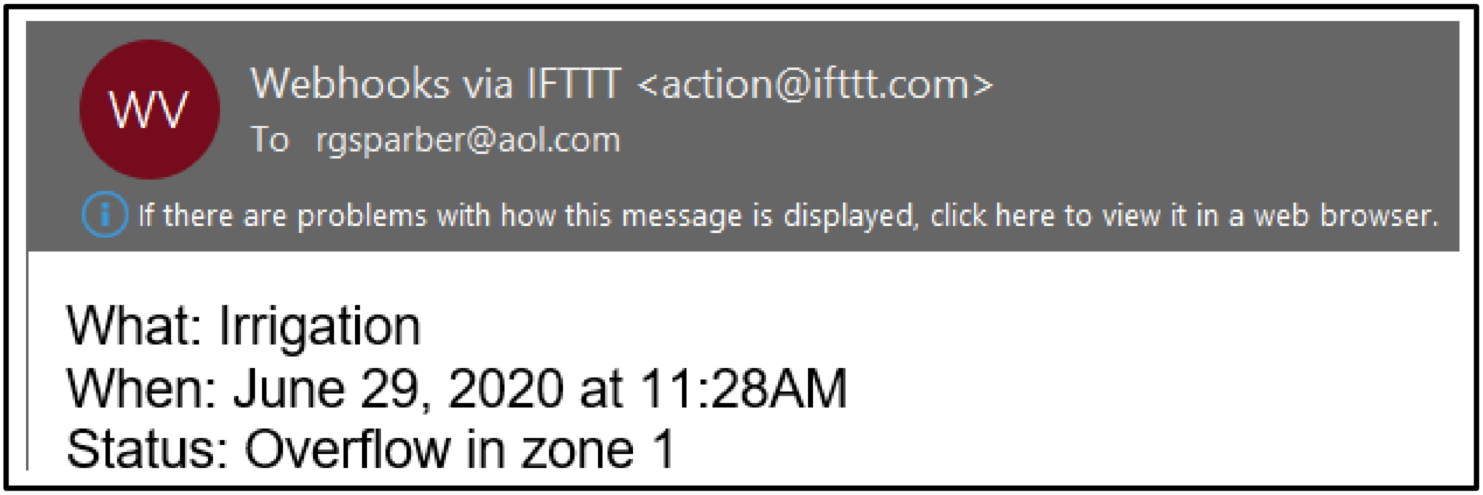

Hardware failure

There is an electromechanical relay inside the Flow Monitor And Control that can fail. As long as it is stuck in the open position, this message will display and a Minor alarm will sound. However, if the problem goes away, we will return to normal operation.

It is also possible that software errors will occur. This might be from real software bugs or could be due to electrical noise crashing the Flow Monitor And Control. If you get a software error message, please write it down. Then unplug power for 10 seconds and plug it back in. See if the problem returns. Any correlations with zones turning on or off would be helpful in debugging the problem.

[7] One unit of water is 3.6 ounces or 0.028 gallons.

Most of the above information is sent via email to the user.

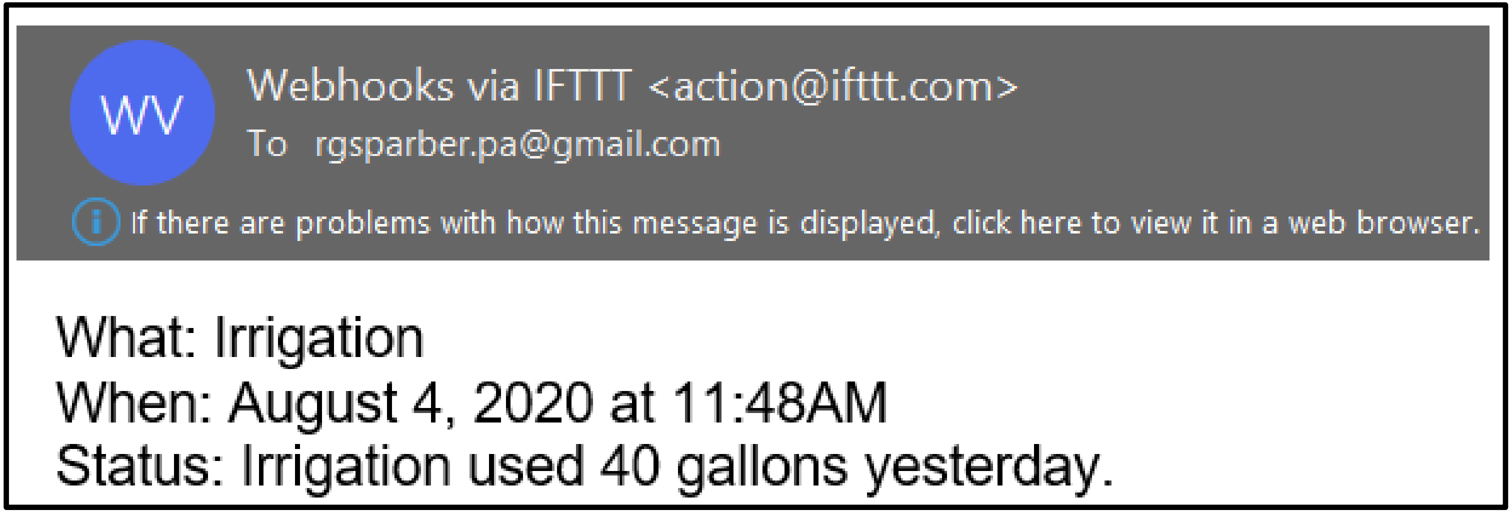


Webhooks via IFTTT <action@ifttt.com>
To rgsparber@aol.com

If there are problems with how this message is displayed, click here to view it in a web browser.

What: Irrigation
When: June 29, 2020 at 11:28AM
Status: Overflow in zone 1

Additionally, at dawn, they will receive a status message telling you how much water was used the previous day.



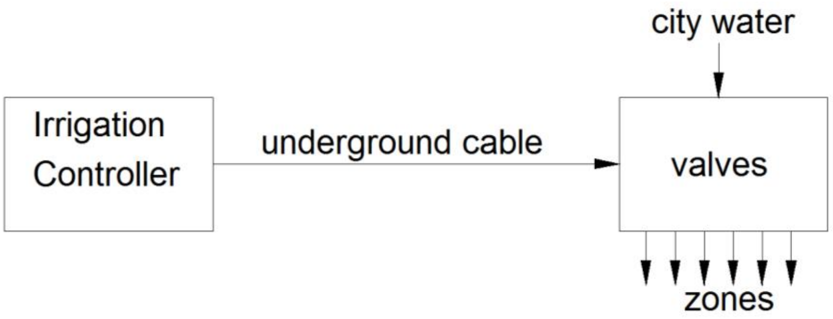
Webhooks via IFTTT <action@ifttt.com>
To rgsparber.pa@gmail.com

If there are problems with how this message is displayed, click here to view it in a web browser.

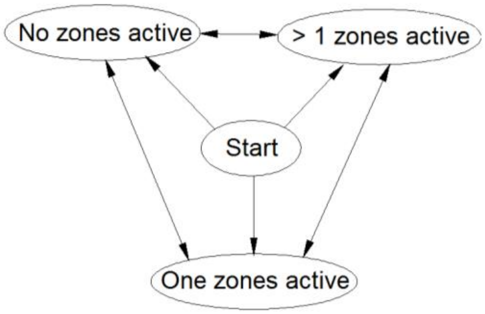
What: Irrigation
When: August 4, 2020 at 11:48AM
Status: Irrigation used 40 gallons yesterday.

Contents

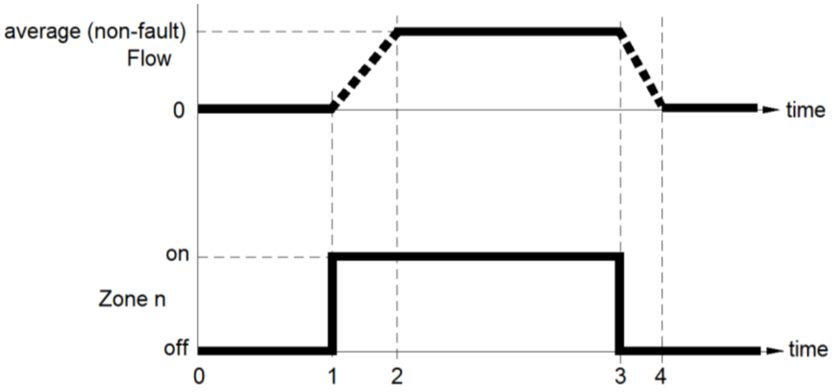
The Original Configuration



As intended by the manufacturer, the RainBird Irrigation Controller is connected to a number of electrically powered water valves via a cable. City water is applied to the input side of each valve. The controller applies power via the cable to one valve at a time to send water to the corresponding zone.



As viewed from a state diagram, the Irrigation Controller powers up in the "Start" state. From there we can move to one of three states. Either no zones are active, one zone is active, or more than one zone is active[8]. The irrigation controller can move between one zone active and no zones active as needed.

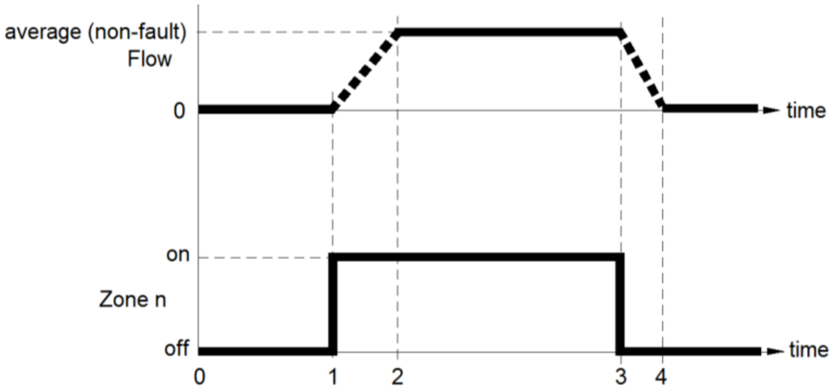


This behavior can be viewed on a time line. From start at time 0, we move to no zones active. Water flow is 0.

At time 1 Zone *n* turns on and flow starts to rise. It takes time for the pressure to build up in the pipes and for any pop up sprayers to rise. I call this the transient time and is less than 10 seconds for my small residential system.

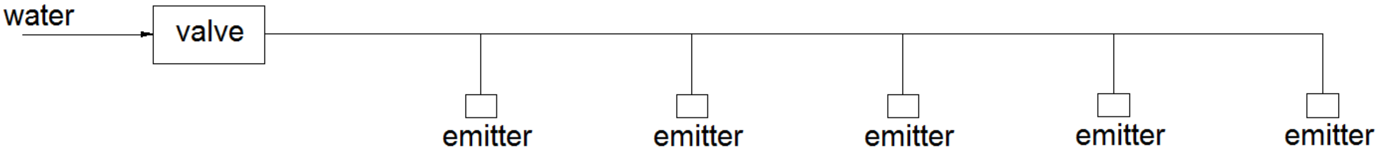
At time 2 the flow has stabilized and we are around the average flow rate. The rate could vary as a function of water pressure and sprinkler head variation but is expected to be within ±30% of the average. Between time 2 and 3 Zone *n* is running at full flow.

[8] This last state can only occur manually.

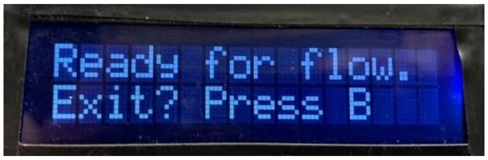


At time 3 the zone turns off and the flow starts to drop. At time 4 the flow is back to 0. This is the second transient time and is also less than 10 seconds. We just transitioned from one zone on to no zones on.

Auditing A Zone



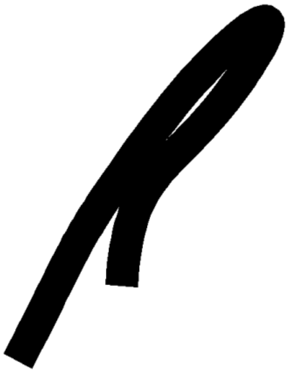
The FMC assumes that the first time it watches a zone run, all is well. It is therefore essential that the user verifies there are no faults.



```
Ready for flow.
Exit? Press B
```

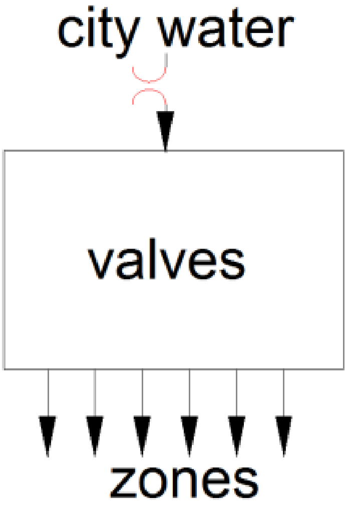
The FMC can be turned into a flow meter by pressing the B button when no alarms are active.

Turn the zone on manually. Then walk along the zone. If an emitter or sprinkler head should be putting out water and is not, time to repair it.

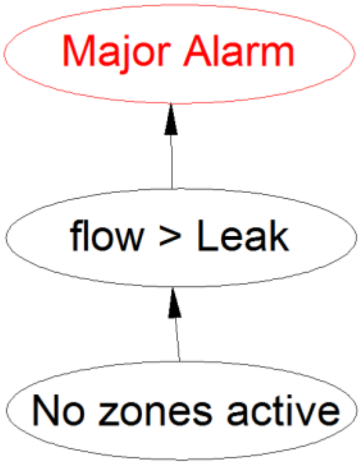


One way to detect leaks in a zone is to block all ¼ inch lines and see how much flow is left. This can be done by bending each ¼ inch line in half and securing it with a piece of wire or bag tie. No water should then come out of the emitter. With all ¼ inch lines blocked, the flow seen by the FMC will be from leaks.

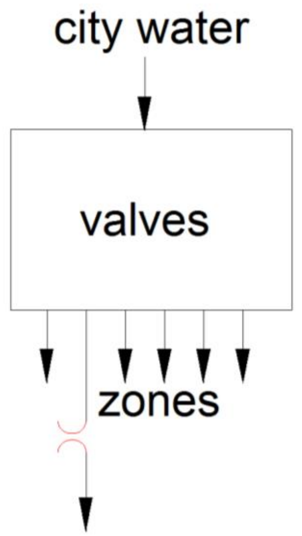
Fault Scenarios



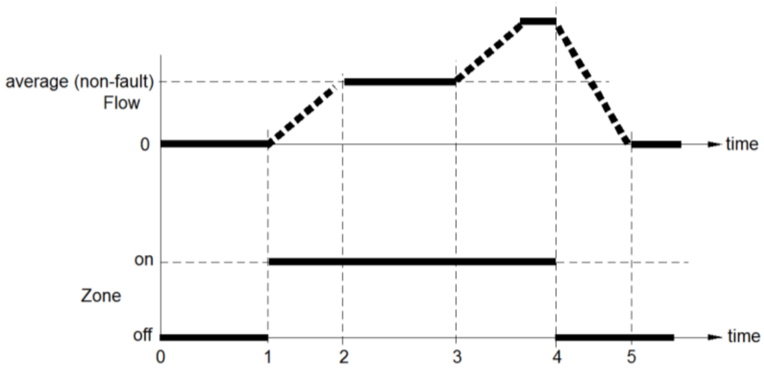
Of course, things do go wrong. A leak can develop between the city water supply and the valves. With no valve in the way, the leak cannot be stopped except by me turning off the city water.



We need to measure the flow, determine it is more than the Leakage limit, and sound a "Major Alarm". Of course, this assumes the leak occurs downstream of the flow measuring device.

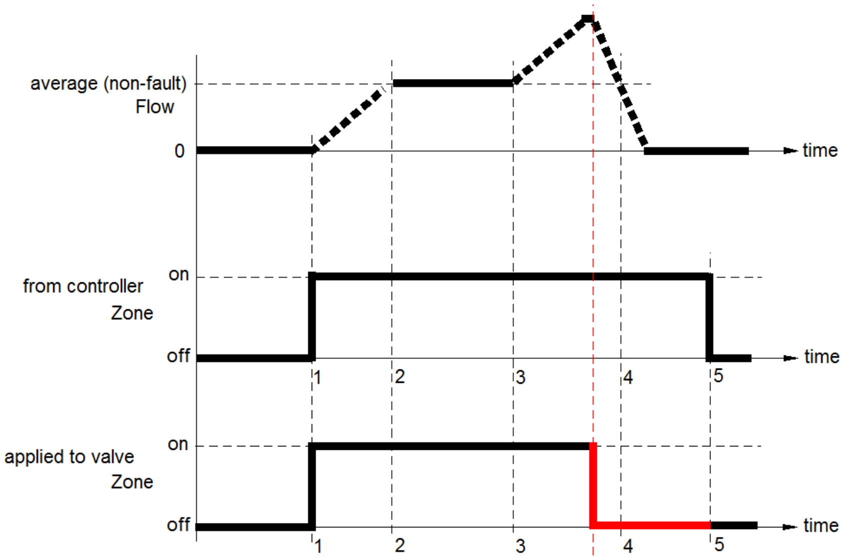


More likely is a leak in a zone downstream from the valve. As long as the zone is active, excess water will flow. But when we return to having all zones off, the leak stops.



In this time line we start out the same as the non-fault case. The zone is initially off and the flow is 0. At time 1 the zone turns on and the flow starts to build. At time 2 the flow is at the average. But at time 3 the flow builds up a second time. Maybe a pipe burst or a sprinkler head blew. We end up at a flow more than 30% above average. At time 4 the controller turns off the zone, not knowing that a fault has occurred. The flow then drops back to 0 at time 5.

Blocking the Faulted Flow



Now, what if we added a new function that was able to monitor the flow and the state of each zone. When it saw excessive flow, the new function would turn the zone off (red lines) and sound an alarm.

When the controller turned this zone off at time 5, the new function would stop interfering.

So what is "average (non-fault) flow"? In order to reduce the burden on the user, the system will assume that the first time a zone runs, it is normal. The software will record the average flow for this run and save it as normal flow. If the next time the zone runs it is within 30% of this historical data, all is fine. The new run becomes the historical data. If a fault is detected, the previous historical data is kept.

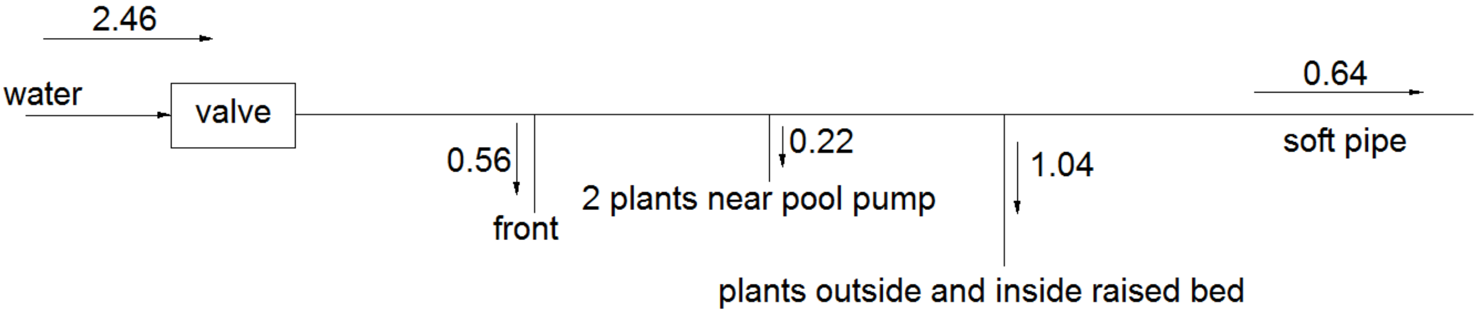
There is a blind spot here: if a zone is faulted while running for the first time, the flow will be assumed normal and become the historical data. The user can manually clear out all historical data so the system can learn anew.

Detecting A Fault

We do not want any false alarms but also do not want to miss faults. Experience has taught me that normal flow can vary ± 25%. To avoid false alarms, we want our limit to be greater than this variation. RainBird has their limit set at ±30% which is reasonable[9].

If a zone consists of sprinkler heads, the nominal flow will be much larger than with drip but a blown head will pass more water too. I don't have any problem detecting a blown head. However, a cracked riser could go undetected if it passes less than 30% of the total.

Here is a case study of a poorly designed zone.



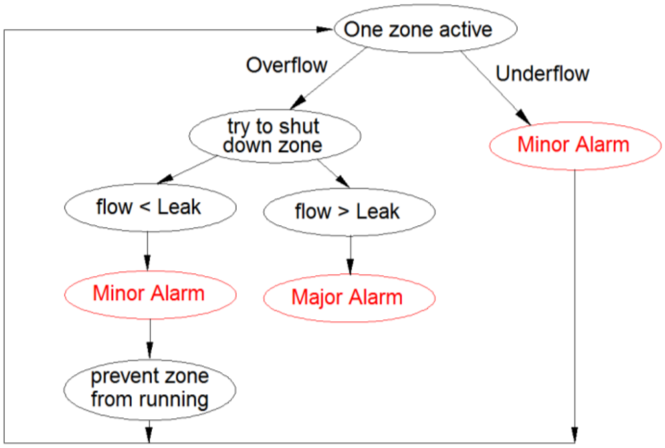
The FMC tells me this zone uses 2.46 GPM. I was able to sequentially close off flow and build this map. Additionally, I introduced two faults to see how the flow would change. Cutting a ¼ inch line near the raised bed caused an increase of 0.6 GPM. Far down the soft pipe run I only measured a 0.3 GPM rise due to a cut ¼ inch line.

30% of 2.46 GPM is 0.74 GPM. Therefore, neither fault will trigger an alarm. At the risk of causing a false alarm, I could reduce my alarm threshold to below $\frac{0.3\ GPM}{2.46\ GPM} \times 100\% = 12.2\%$. Alternately, I could just accept that this loss of detection is a consequence of an excessively large zone. Breaking this zone into thirds would solve the problem.

My really big leaks have come from split soft pipe that is buried deep enough in coarse crushed rock as to not show a puddle on the surface. Such a leak would likely increase flow more than 30% so would alarm. To directly address this failure mode, I replace all leaking soft pipe with ½ inch schedule 40 PVC.

[9] Any zone with an average flow rate of less than 0.04 GPM is given a larger limit.

Logic Overview



To address faults, we need to

1. monitor the flow on an active zone
2. determine if it is too much
3. try to shut down the valve
4. check the flow again
5. determine if it is now less than leakage
6. and sound an alarm

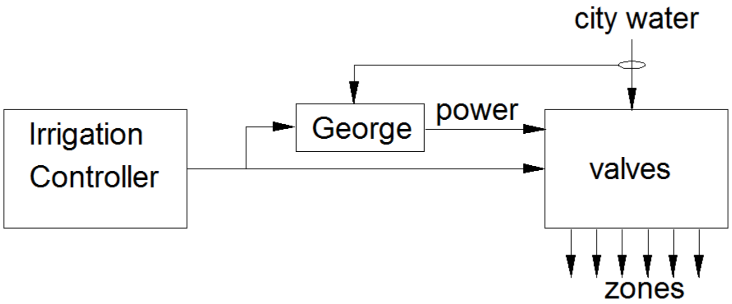
If closing the valve stopped the flow, we call it a Minor Alarm. Then we mark this zone so in the future it is prevented from running.

If, after we try to shut down the zone, the flow is more than leakage, we did not stop the flow. It is then marked as a Major Alarm.

If the flow is too small, we call it Underflow and declare a Minor Alarm. The plants are not getting enough water so shutting down the flow would only make matters worse.

System Overview

Flow Monitoring and Control is not a new invention. Companies like RainBird sell a similar function. They are used in application where a large amount of water normally flows and one leak can be costly. The price of such a system is over $1000 and I'm sure it pays for itself quickly. But in my little home irrigation system, this price tag is out of the question. Mine cost around $150.



I will call this new function "George". He sits between the Irrigation Controller and all of the valves.

George can see which valves have been operated by the Irrigation Controller and also how much water is flowing.

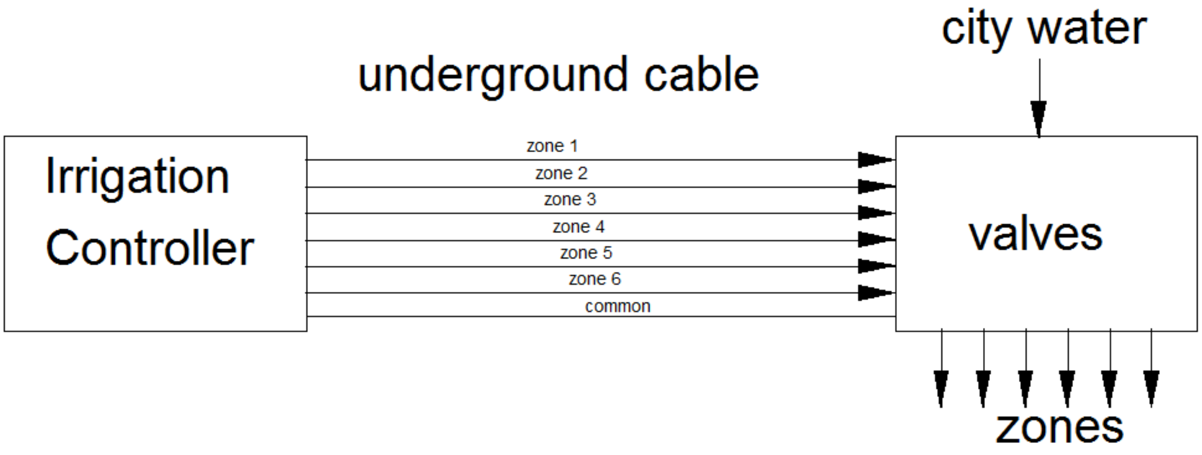
If the leak cannot be stopped, George must sound an alarm immediately. I must then manually turn off the city water.

If the leak is only flowing while zone ***n*** is active, George can turn off valve ***n*** every time the controller turns it on. Then he can sound an alarm during the day. No need to wake me up if it is night.

My RainBird Controller automatically turns on only one zone at a time. The user can manually turn on more than one zone. In this case, the user must deal with any excessive flow. Since George only looks at the total flow, he would not be able to figure out which zone was in trouble.

The user can also manually turn on a zone right on the valve. This will look like an uncontrolled leak and generate an Overflow Major Alarm. To avoid such excitement, it is best to manually turn on zones via the Irrigation Controller.

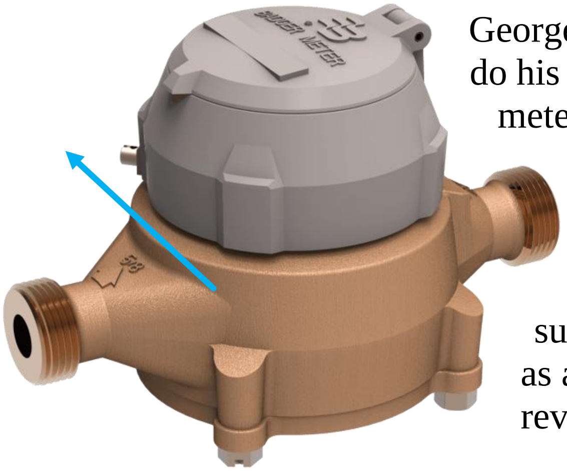
Details of the Existing System



A seven conductor cable carries control power from the controller to the valves. When the controller applies about 24 volts AC to a zone wire, the corresponding valve opens and water flows in the associated zone.

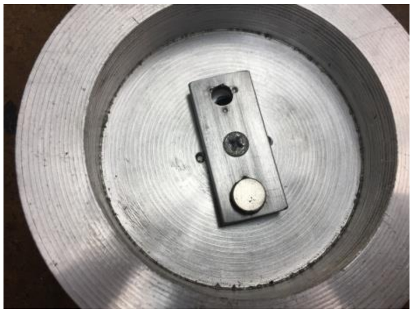
Introduction to the New Functionality

The Flow Measuring Device

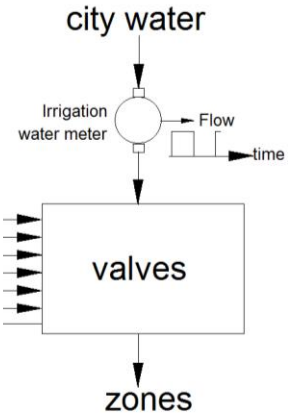


George depends on knowing the precise flow rate in order to do his job. Therefore, the first thing I bought was a water meter. It was on eBay for $27 including shipping. This meter is a Badger model 25 with a plastic body[10].

By removing a screw (red arrow), the readout head comes off. This provides access to a surface with a spinning steel bar under it. The bar turns as a function of flow through the meter. I see one revolution every 0.028 gallons of flow[11].



I sense the steel bar with a magnet attached to an arm. The arm is supported by a ball bearing so is free to spin. A magnetic sensor suspended over the path of the magnet reports each revolution. It outputs pulses that are read by George.



By measuring the flow over 1 minute, George calculates the Gallons Per Minute (GPM). The meter connects between the city water inlet and my valves and will provide water usage for my irrigation system. This is necessary but not sufficient for knowing the water usage per zone.

Logic is needed to look at which zone is active and how much water is flowing.

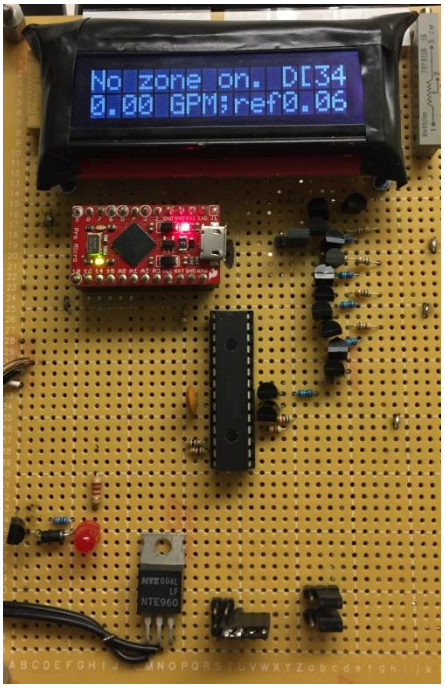
[10] The picture is of a bronze body meter. The plastic ones cost less and are black.
[11] I used the city's water meter while I counted revolutions to determine this conversion factor.

On 12/13/2019 I was searching for an off-the-shelf flow measuring device and came upon flows.com which is owned by Assured Automation. They are located in New Jersey. They sell the WM-PD-050 low volume flow measuring device. There is an option to add an electrical contact that closes each time 0.05 gallons passes through it. The price, including shipping/handling and tax is about $100.

This is a major find because it means that all components of the Irrigation Flow System can be bought, rather than fabricated.

The Flow Monitor and Control

The second piece of the puzzle is the Flow Monitor and Control that will read the flow signal from the water meter plus read and control the signals in the cable. This intelligence will have control of the valves including overriding the Irrigation Controller if a fault occurs in the valves or irrigation zones.



The FMC is built using discrete parts plus an Arduino system on a board. Since this is a prototype, I used a board much larger than ultimately needed. These things do have a tendency to grow over time.

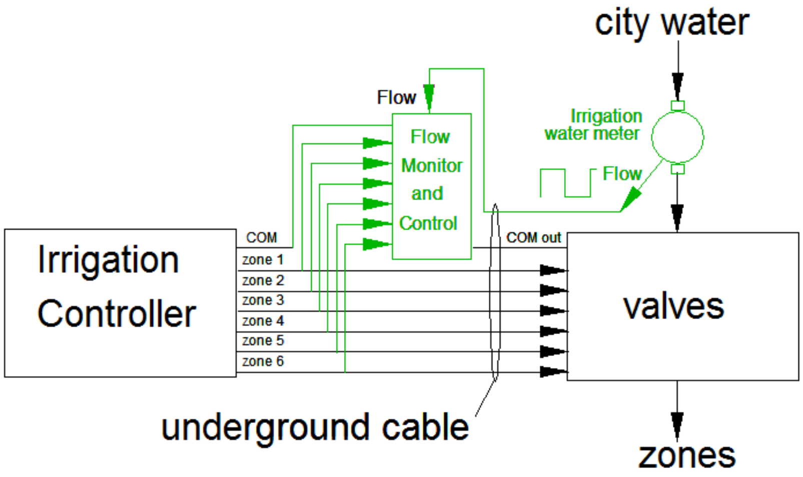
If turned into a real product, I expect the electronics to be all Surface Mount Technology. It likely could be slightly larger than the display.

Presenting the Design

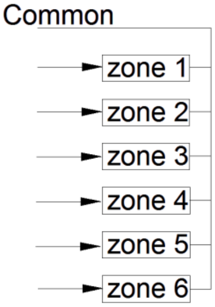
I have no interest boring you with ***how*** I arrived at the following design. Instead I will present the illusion that it was a simple task with no dead ends or false starts.

The hardware will be presented first starting with a high level view and ending with circuits. Then the software will be presented starting with a high level view and ending in code.

Level One System Block Diagram



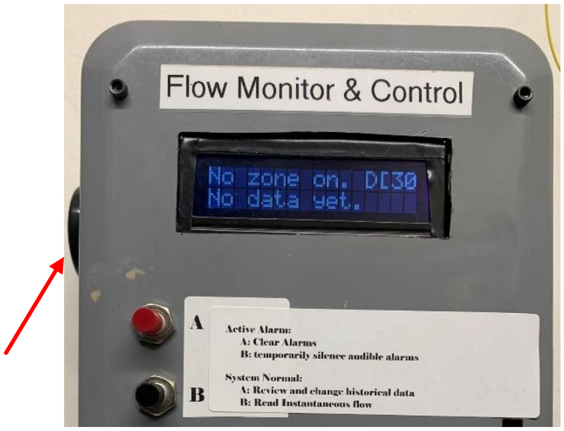
The Flow Monitor and Control taps into each zone output of the Irrigation Controller. It also connects to the Common (COM) wire. Under non-fault conditions, "Common out" (COM out) connects to COM.



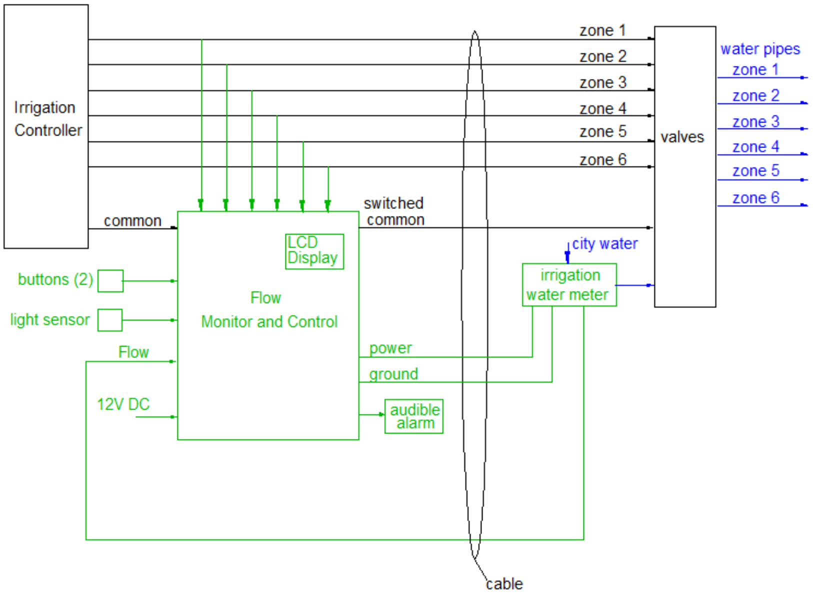
All of the valves have two wires. One wire connects to a zone wire. The other wire is joined with all other valves into a single conductor called Common. Apply power between any zone wire and Common and that valves will operate.

The figure shows the flow signal connected to wires in the underground cable. In my case, the original cable had no spare wires so I had to run another cable.

Level Two System Block Diagram

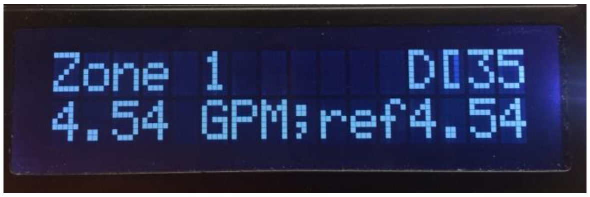

Flow Monitor & Control
A
B
Active Alarm:
A: Clear Alarms
B: temporarily silence audible alarms
System Normal:
A: Review and change historical data
B: Read Instantaneous flow

```
No zone on. D[30
No data yet.
```



The FMC's Human/Machine Interface consists of a display, two buttons, and a piezoelectric beeper. The beeper can be seen on the left side of the box (red arrow).

On many screens you will see "D[xx" where "xx" is a number between 0 and 70. This is a countdown timer that tells me how long until the next update. If during the day, you will see "D". At night, it will be "N". Remember that Minor Alarms sound only during the day. Major Alarms sound when they occur.



```
Zone 1      D[35
4.54 GPM:ref4.54
```

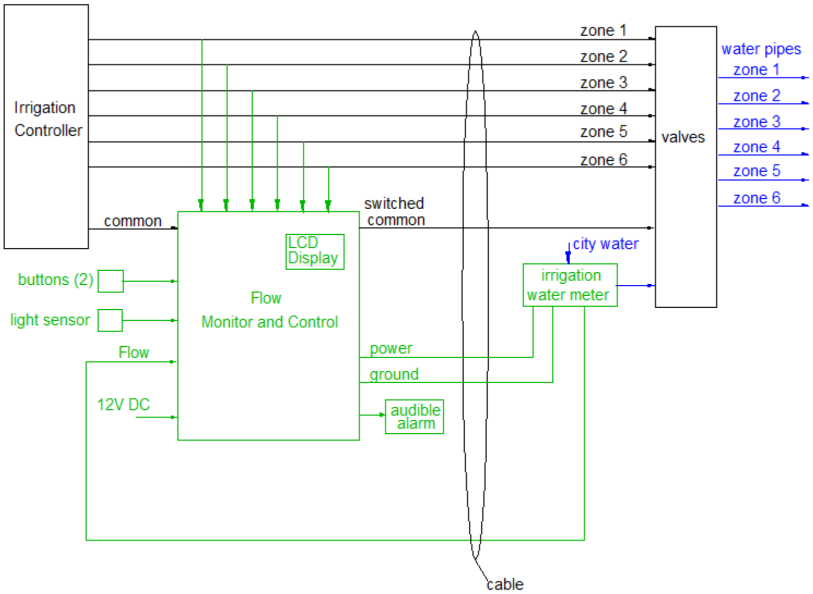
It is also common to see the flow rate on the second line. Here you have 4.54 GPM. It is being compared to the reference which is also 4.54. If there was no reference yet, you would have seen "ref-". After the zone has run once, the "-" is replaced by the reference flow.

If any alarms are active, the top button (A), clears them. If the problem has not been corrected, the alarm will reappear after 70 seconds. The bottom button (B) will temporarily silence the audible alarm. A Minor Alarm will stay quiet for 24 hours. A Major Alarm will return in 15 minutes.

The audible alarms emanate from the piezoelectric beeper. Intentionally, it is extremely annoying. A Minor Alarm will be on for 0.3 seconds and off for 2 seconds. A Major Alarm will be on for 0.1 seconds and off for 0.1 seconds. These patterns were chosen so they would not sound like a smoke alarm.

Flow Monitor and Control Inputs

The FMC will monitor all outputs of the Irrigation Controller so it knows what zone is supposed to be active. Under automatic control, there should be only one zone running at a time.



The FMC will also accept the common wire. This is the return path for all valves. If the FMC detects excessive flow through a valve, it will be able to disconnect the common wire and remove power to all valves.

The light sensor is located outside and outputs a logic 1 when the sun is out. It tells the system when there is daylight so safe to sound minor audible alarms.

This approach lets me avoid the cost and complexity of including a real time clock. I do not need to know the exact time, just that it is a reasonable hour to sound a non-urgent alarm.

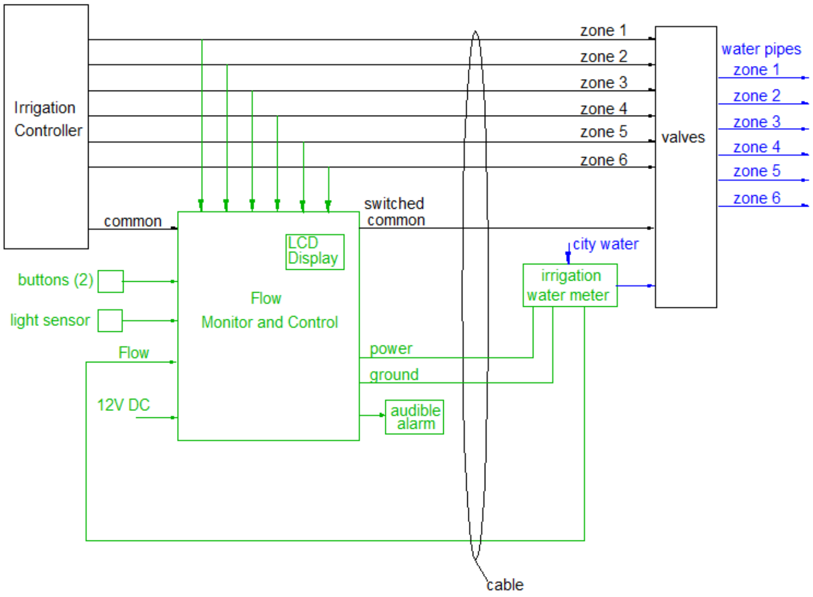
The irrigation water meter has a Hall Effect magnetic sensor attached to it. For every 0.028 gallons of water passed through the meter, the sensor will generate a pulse on the Flow line (red arrow). The FMC counts these pulses over time to determine flow rate. The maximum flow I have seen is 15 GPM. This means a pulse every

$$\frac{15\ Gallon}{minute} \times \frac{1\ pulse}{0.028\ gallons} = \frac{1\ pulse}{1.9\ milliseconds}.$$

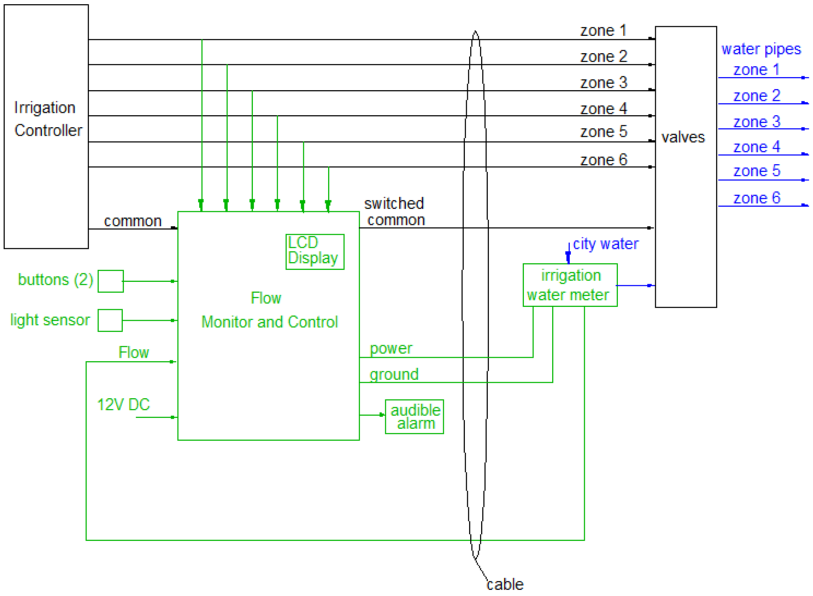
The software has been designed to keep up with this pulse rate.

The FMC receives unregulated 12V DC.

It is used directly to run the relay that switches common and also to power the piezoelectric beeper. Most of this power feeds into a 5V regulator which supplies the rest of the board.



Flow Monitor and Control Outputs



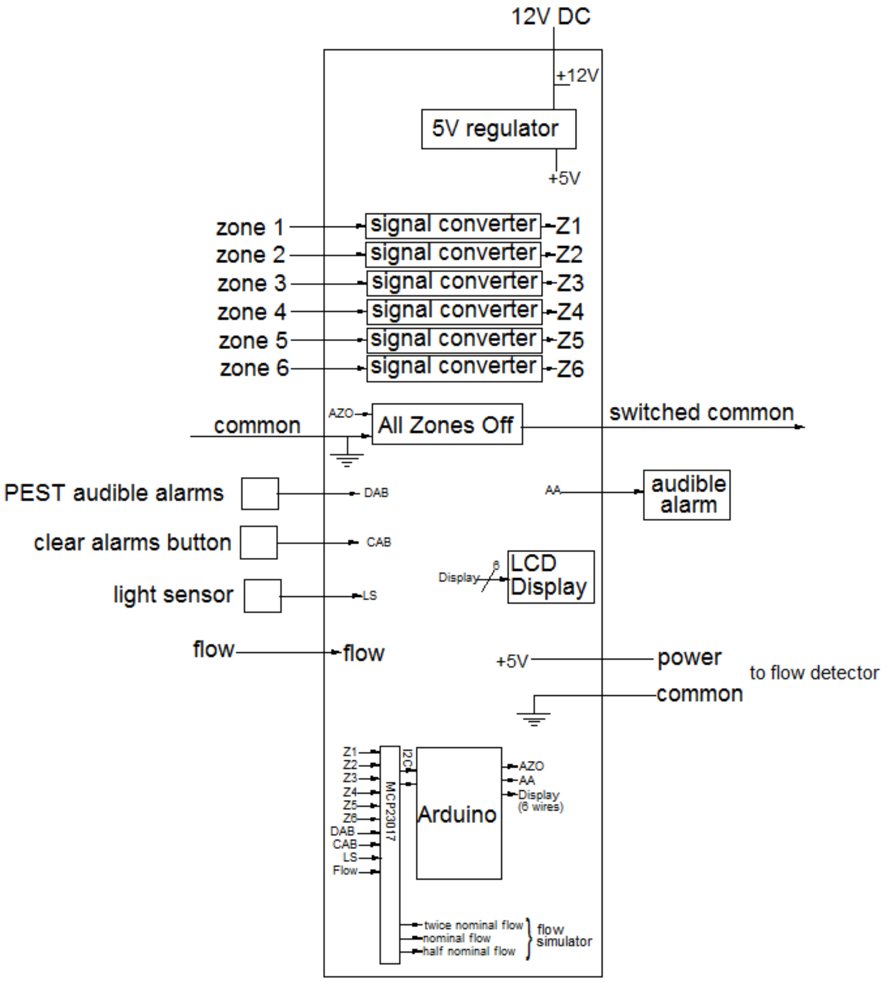
The FMC switches the ***switched common*** line between the ***common*** line at its input and an open circuit.

The LCD display tell the user status.

Power is supplied to the magnetic sensor on the water meter.

The audible alarm is sounded by the software to alert the user of any fault conditions.

Level Three System Block Diagram

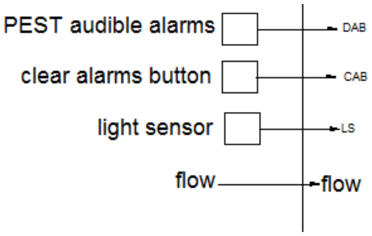


The FMC is controlled by an Arduino Pro Micro 5V/16 MHz. In runs on 5V and has a 16 MHz clock. The Arduino is paired with a MCP23017 port expander to provide sufficient digital input and output pins.

Starting at the top, we have unregulated 12V DC coming from a wall wart. This power supply can deliver up to 500 mA. The voltage is fed to the piezoelectric beeper and to the relay that switches the common wire. A 5V regulator brings the 12V down to a level used by the rest of the electronics.

Below the 5V regulator are our signal converters. They sense 24VAC on the zone leads and convert them to 5V logic levels.

Next we have the All Zones Off function. It contains a normally closed relay under software control. When no power is applied to this relay, the “common” wire is connected to the “switched common” wire. When needed, the Arduino will send an active AZO signal to the All Zones Off box which will energize the relay and disconnect the switched common from the common wire. This will cut power to all valves. Note that if the FMC loses power, it will become transparent to the RainBird Irrigation Controller.

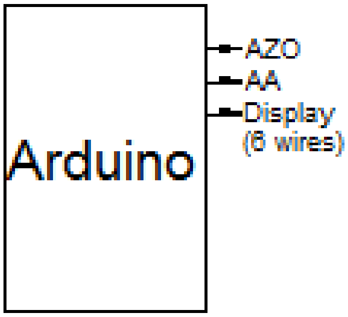


The “B” button used to be called PEST audible alarms, and the “A” button used to be called clear alarm button. They feed directly into the port expander which has been configured to connect 20K pull up resistors.

The light sensor generates a digital signal compatible with the port expander. The sensor was placed where it can detect daylight. If an irrigation fault is detected and controlled, the resulting alarm won’t sound until the sun comes out. No need to wake me.

The flow signal from the irrigation water meter feeds directly into the port expander.

Software is used to remove any mechanical bounce in the buttons.



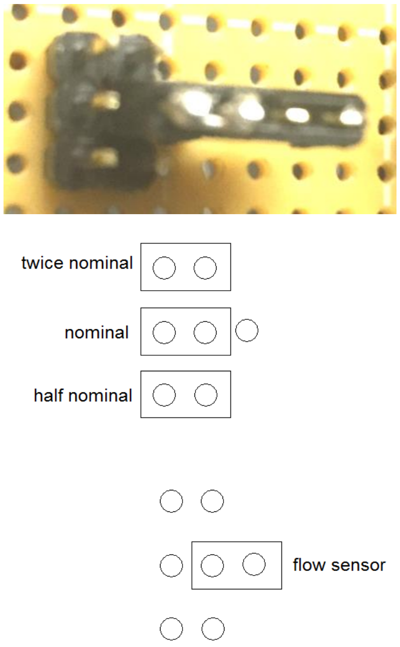
The Arduino has three outputs.

During an irrigation fault condition, it drives the AZO signal which cuts power to all valves and hopefully stops water flow.

It drives an audible alarm, AA, to notify the user when a problem has been detected.

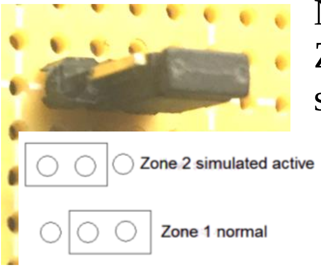
And finally, The Arduino feeds the Liquid Crystal Display to provide system status. While the other three outputs are single pins, the display takes 6.

Built In Testing Features



In order to facilitate testing, a flow simulator has been built into the software and hardware. The port expander can output a pulse stream equivalent to a flow of 4.5 GPM. It can also output 9 GPM and 2.25 GPM. These signals are used to test nominal, Overflow, and Underflow states. By moving a jumper I can select the flow sensor or simulated flow. The jumper is shown in the flow sensor position.

My jumper block is made from a length of connector to make it easier to grasp around all of those pins.



Not shown in the diagram are two jumpers that can simulate Zone 1 and Zone 2 being active. Here you see both jumpers set to normal operation.

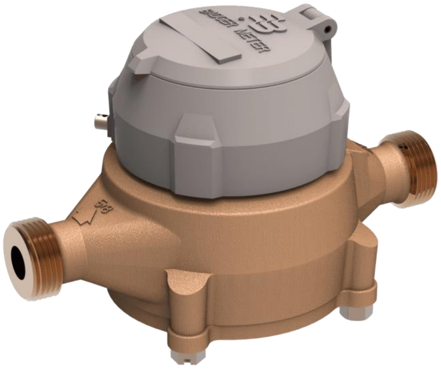
An LED with limiting resistor connects across the relay coils. This makes it easier to see when the software has disconnected the switched common wire from the common wire.

Additionally, there is a software switch called SpeedUp. When turned on, the software only measures flow for 10 seconds rather than a full minute. It also shortens the PEST intervals to 100 seconds. This makes testing the software quicker.

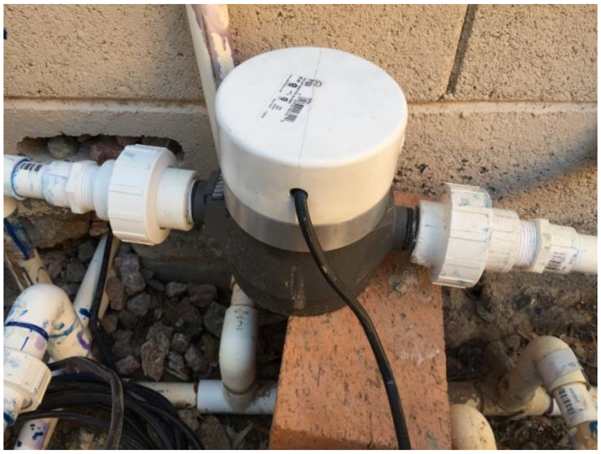
Level Four System Block Diagram

At this level of detail, each functional block is described.

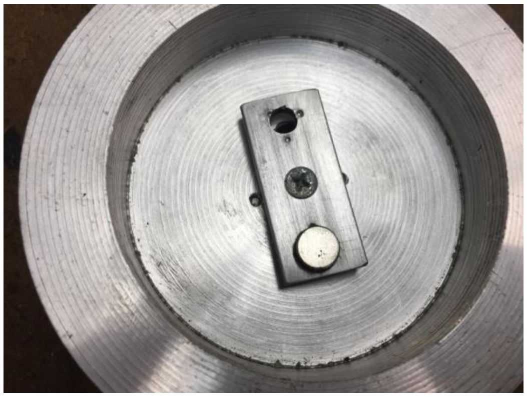
Flow Sensor Mechanism



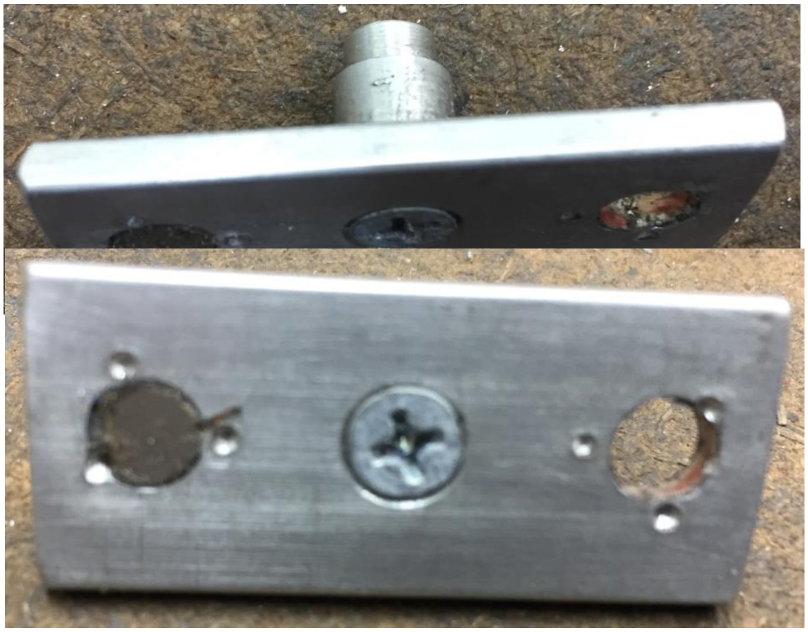
I started with a Badger model 25 flow meter. The readout mechanism was lifted off



and my interface hardware dropped in place.

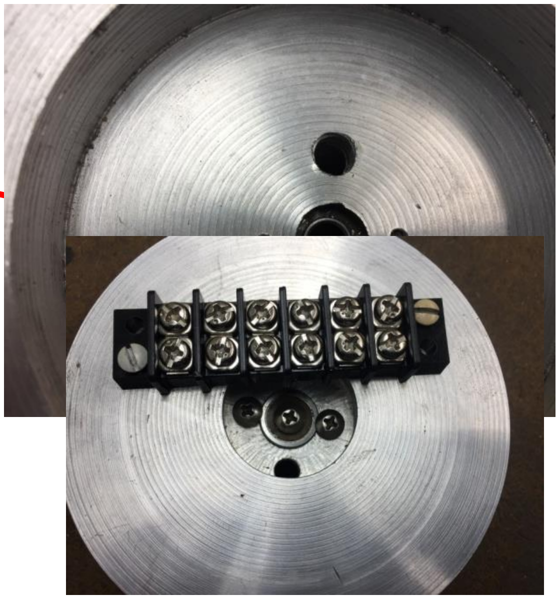


The interface hardware was an unfortunate addition. Originally I thought that there was a spinning magnet inside the water meter. This could be directly sensed by a Hall Effect device. But upon investigation, I discovered that a steel bar was in there. The interface hardware senses the bar and generates a spinning magnetic field. Then my Hall Effect device can do its job and convert revolutions to pulses. By using the City’s water meter as my reference, I found that one revolution equals 0.028 gallons of flow.



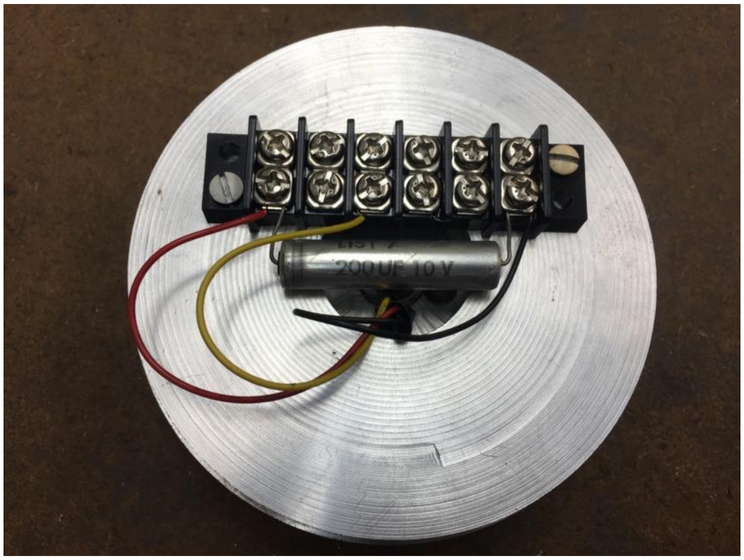
The bar has a spigot on the back that is a close fit to the ID of a ball bearing. The bar has a ¼ inch diameter hole in it that accepts a neodymium magnet 1/8 inch thick. I locked it in place with 3 punch marks. At first I had two magnets but found that this prevented movement.

I later found that I needed to move the bar closer to the top of the meter in order to insure reliable coupling. Placing a 3/8 inch diameter by 1/8 inch thick magnet on top did the trick.



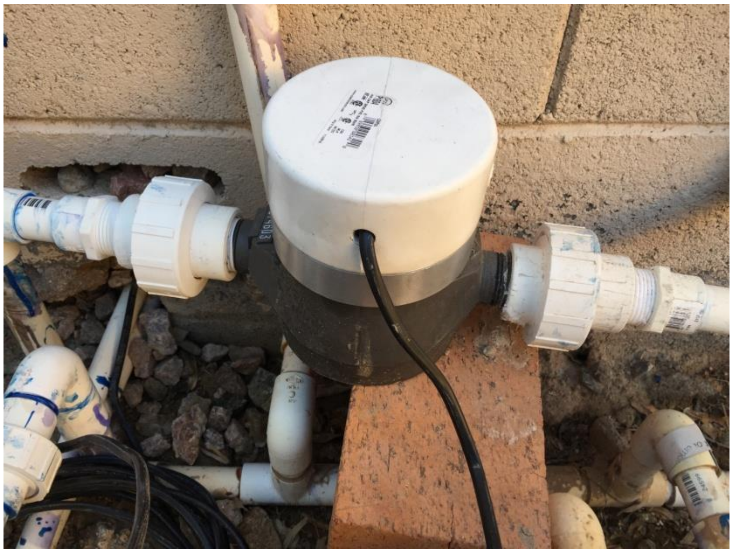
Looking at the underside of the enclosure, you can see the ball bearing’s inside diameter. The spigot slides through and is secured with a screw on the other side.

This hole accepts the Hall Effect sensor.



With the Hall Effect sensor installed, the power, ground, and output leads are routed to a terminal block. The electrolytic capacitor completes the assembly.

A 4 inch diameter PVC pipe end is used as a weather tight enclosure. The cable running back to the FMC passes through this cover and terminates on the terminal block.



The resulting assembly came out nice but in the future I hope to be able to simplify the interface hardware and go with a proximity switch that can sense the spinning steel bar. The proximity switch I tried did not work.

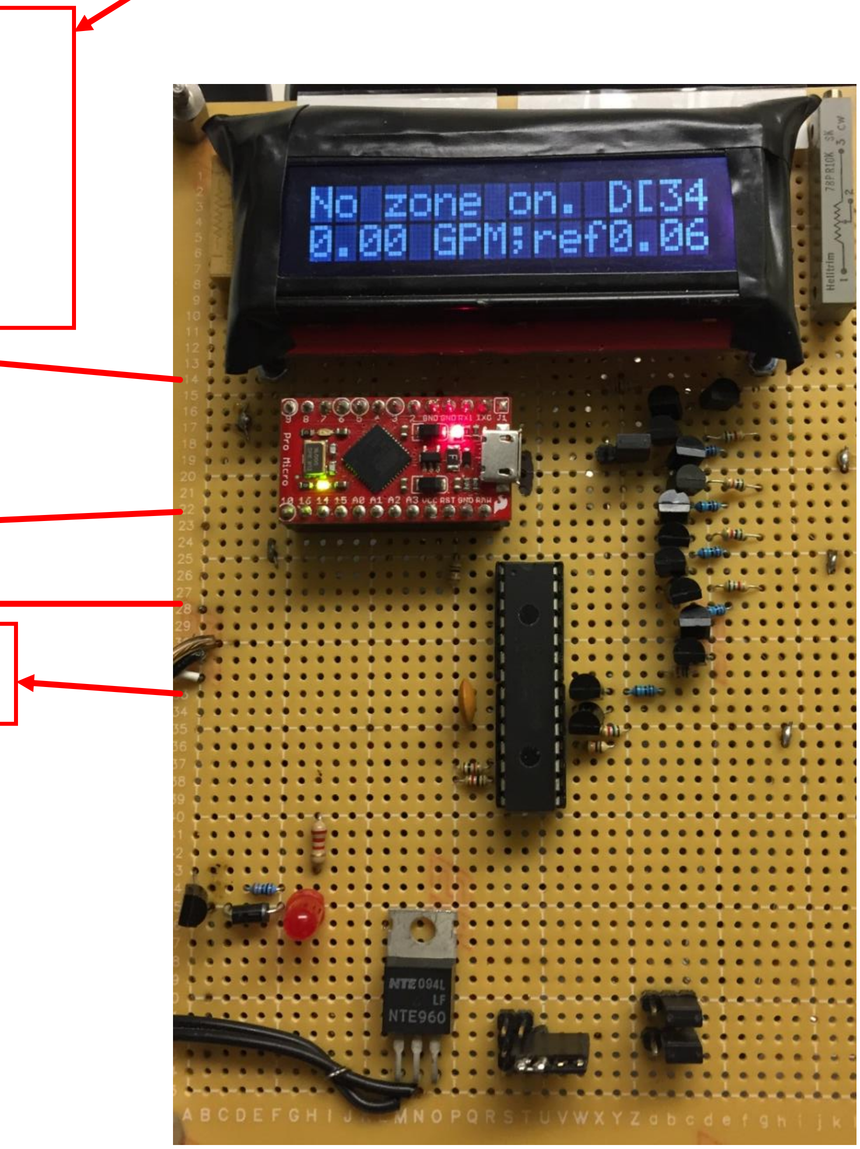
I did spend a lot of time sitting with the installed flow meter trying to get it to work. This was not without excitement. That is a Western Diamond Back rattlesnake. My wrist was about one inch from its head when I realized it was there. Fortunately, it was early morning and cold out. The snake did not perceive me to be a threat and was also rather slow. I was able to convince it to move on by filling the pit with some water. I watched as it went back through a hole in the block wall. Closing that hole worked. No more company while I work in the valve pit. Yes, I do look carefully before climbing in.

Flow Monitor and Control

Here is an overview of the subsystems on the perf board.

2 row by 16 character LCD

Arduino Pro Mini



Signal Converter Subsystem

Port expander

Relay driver and status LED

5V regulator

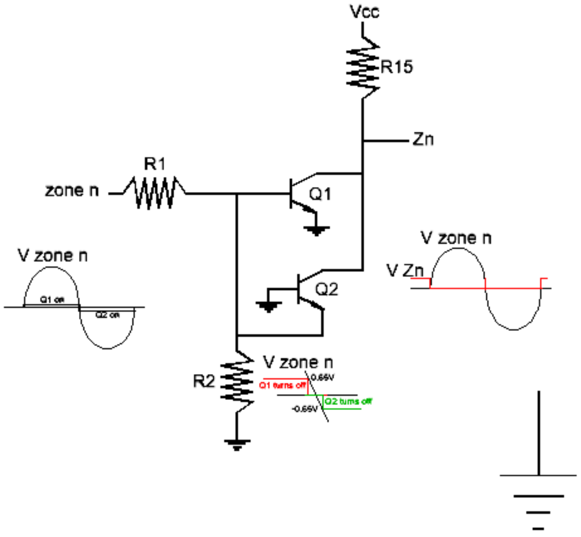
Testing jumpers

The relay is mounted on the back side.

Signal Converter Subsystem

zone 1
signal converter
Z1

The zone wires will either have about 24V RMS on them or be open circuits. The signal converter circuit monitors this AC voltage and puts the state in a format readable by the Arduino.



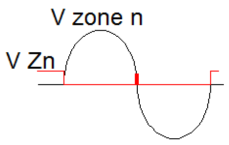
The signal converter circuit consists of 3 resistors and 2 NPN transistors. The input is the voltage $V_{\text{zone } n}$ and the output is V_{Zn}.

When $V_{\text{zone } n}$ is above about 2V, Q_1 turns on. This pulls down on the Z_n node and presents a logic 0 to the port expander.

While Q_1 is on, Q_2 has about -0.65V across its base emitter junction. This keeps Q_2 off but does no damage.

When $V_{\text{zone } n}$ is below about -2V, Q_2 turns on. This pulls down on the Z_n node and presents a logic 0 to the port expander.

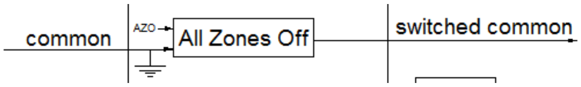
When $V_{\text{zone } n}$ is less than 0.6V but greater than -0.6V, both transistors will be off. The output then gets pulled up to V_{cc} (+5V) by R_{15}.



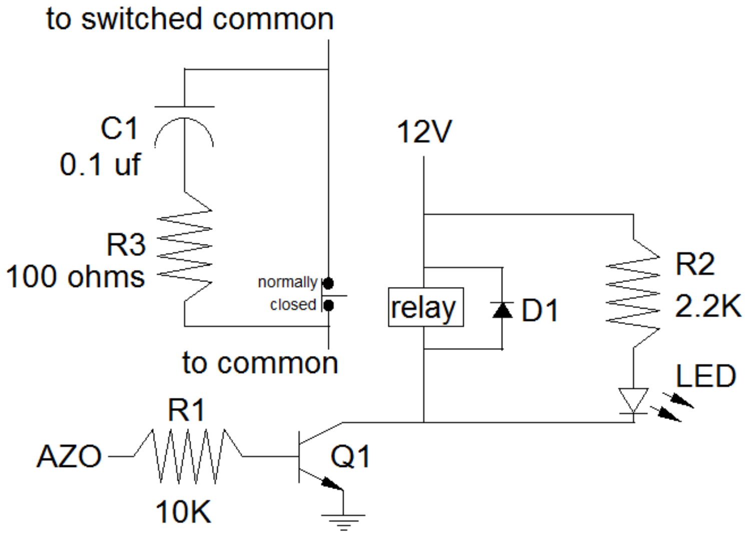
Note the red vertical line on the V_{Zn} trace as $V_{\text{zone n}}$ changes from positive to negative. This results from $V_{\text{zone n}}$ being less than 2V in magnitude. The spike is treated the same as bounce on a mechanical switch.

See Appendix 1 on page 73 for details on this narrow pulse which is about 320 microseconds wide.

All Zones Off

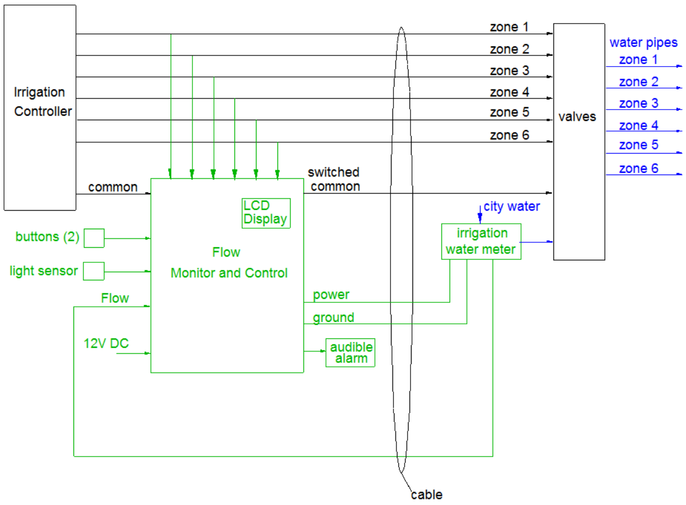


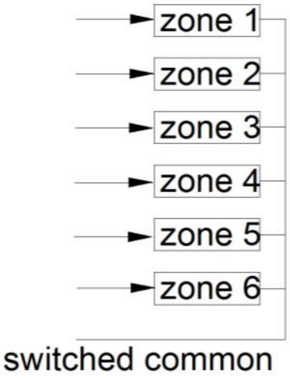
This functional block is an electromagnetic relay with contacts that are closed when power is removed. When the FMC loses power or when all is idle, AZO is near 0 volts. This turns off Q_1 and no power is sent to the relay. The common wire is therefore connected to the switched common wire.



When AZO becomes active (around 5V), Q1 turns on. The relay powers up along with the optional indicator LED next to it.

When the relay is operated, the normally closed contacts open. The current being broken is AC and passes through the valves which are inductive. This means that arcing can occur on these contacts. The snubber circuit, C1 and R3, provide an alternate path for the transient current. This minimizes the arcing which can erode the contacts and cause the Arduino to go off the rails.





When the relay is operated, the switched common lead is disconnected from common. This means the signal converters will be connected to a long cable that floats at the far end. Electrical noise can be picked up and cause zones to randomly look active when they are not.

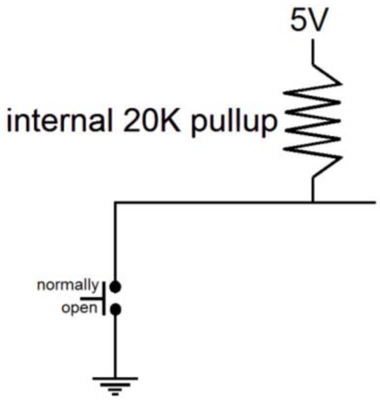
Even worse, if one zone is active so 24V AC is applied to one zone wire and the switched common lead is floating, all zones will be at this voltage. This say they are all active.

The problem is solved in software. When at least one zone is active and the relay is then operated, we just wait for all zones to turn off. That will occur when the one zone goes inactive. When all zones turn off, we again monitor zone states.

If the contacts on the relay fail to conduct due to built up oxides caused by arcing, the software will see all 6 zones turn on when any one of them turns on.

Any time all 6 zones are seen in the on state while the relay is not operated, we assume the relay is the problem. By turning the relay on and off quickly 20 times, we attempt to break up the oxide layer. If that works, we return to normal operation. If we still see all 6 zones on, we monitor the state and only return to normal operation if the fault goes away. The display will say “hardware failure” as long as the fault is detected or until the clear all alarms button is pushed.

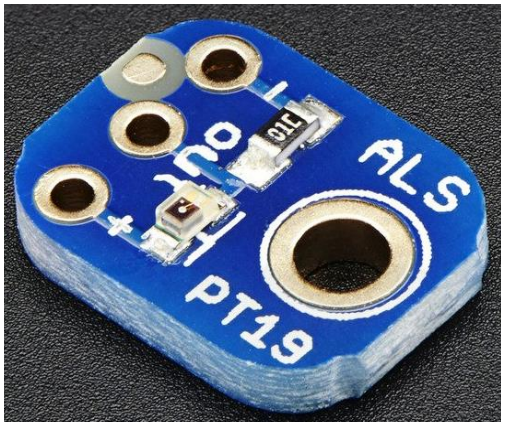
Pushbuttons

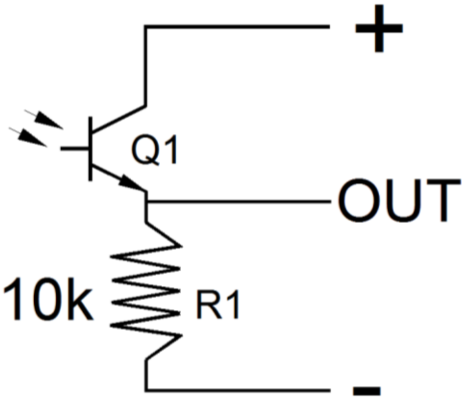


When either of the pushbuttons is depressed, a contact closes and the port expander sees a logic 0. When released, the internal pull up resistor generates a logic 1.

This closing action may cause the contacts to bounce. Software is used to filter that out. We take 10 readings and average the result. If it is less than ½, we call a zero. Greater than or equal to ½ is a one.

Light Sensor

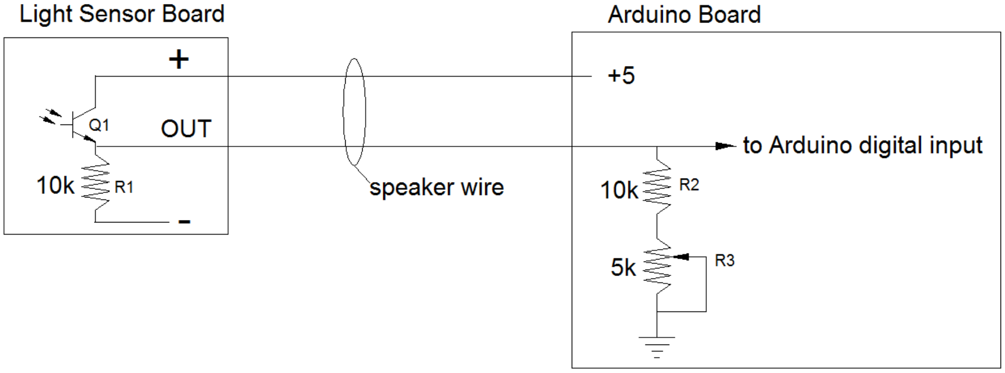




I am using an ALS-PT19 light sensor from Adafruit Industries. When light shines on the sensor, the phototransistor, Q1 conducts. This moves the output voltage towards the voltage on the positive terminal. Without light, Q1 is off and the output voltage moves towards the voltage on the negative terminal.

My light sensor is mounted inside a glass bottle that is a tight fit into a PVC 45° coupler. I happen to have two wire speaker cable. Since the ALS-PT19 has three terminals, I would need to run two of these speaker cables and have half of one cable unused.

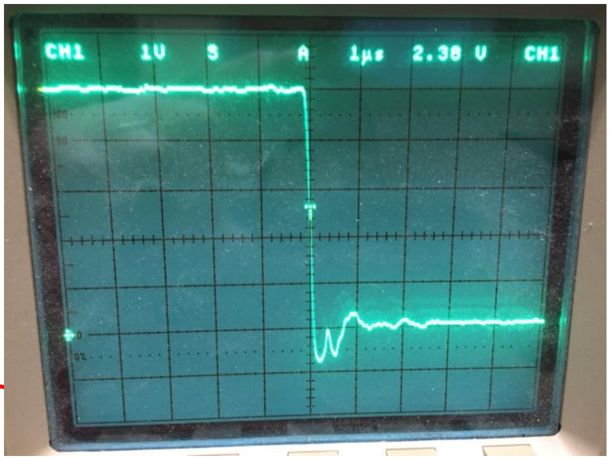
Instead, I chose to not use R1. Then I was able to use a single cable between sensor and Arduino board. Over at the Arduino I duplicated R1 plus put a 5K variable resistor, R3, from the output signal to ground. It was then possible to adjust the variable resistor so daytime generated a logic 1 and nighttime produced a logic 0.



This scheme lets me avoid the cost and complexity of having a battery backed up real time clock.

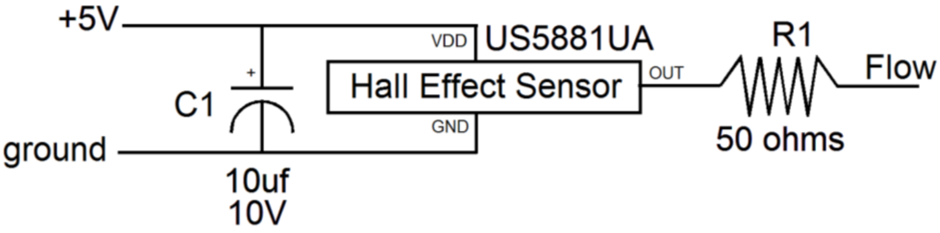
Flow Interface

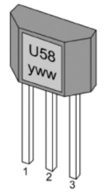
The Hall Effect Sensor pulls the Flow lead down to ground very quickly to generate a logic 0. With a logic 1 it just lets go. Rise time is defined by the capacitance of the cable[12] and a pull up resistor. This means that the fall time is extremely small compared to the rise time. Both are much smaller than the minimum period of the pulse stream.



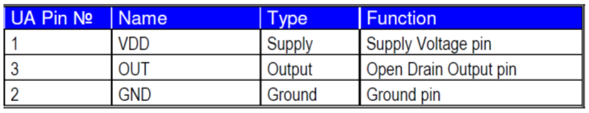
The fast fall time is not a good thing. It causes undershoot at the port expander that is greater than what the device can safely handle. I see about -0.8V which is beyond the -0.5V spec.

By placing a 50 ohm resistor in series with the Hall Effect device's output, I was able to slow the fall time and therefore reduce undershoot. Now the undershoot is about -0.2V so all is well. It still takes only about 200 ns to go from +5V down to near 0.



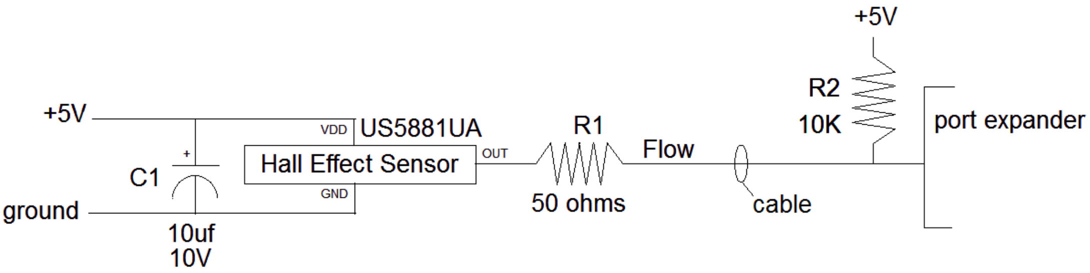


UA package

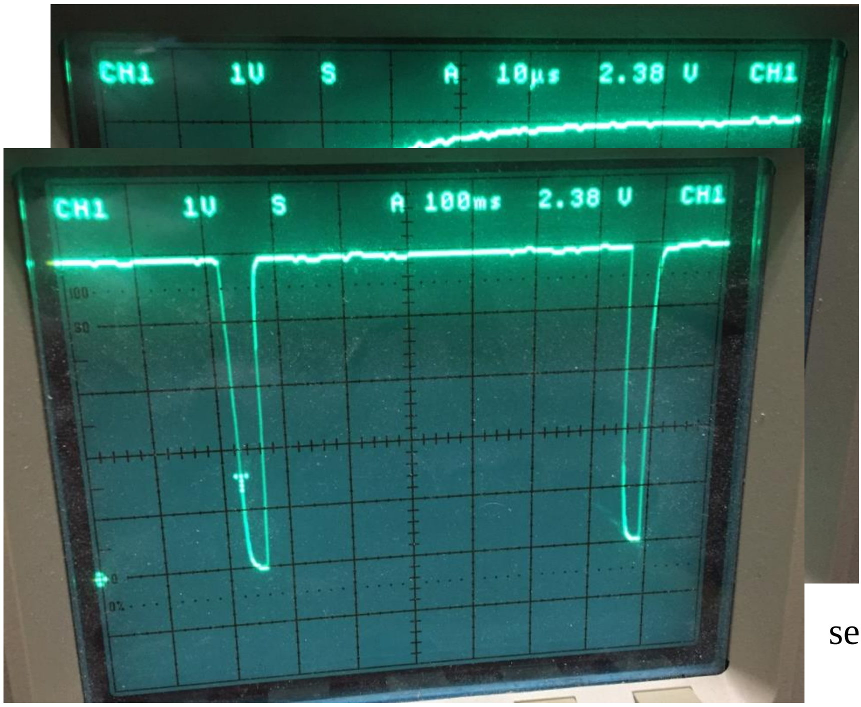


UA Pin №	Name	Type	Function
1	VDD	Supply	Supply Voltage pin
3	OUT	Output	Open Drain Output pin
2	GND	Ground	Ground pin

[12] My cable is 50 feet long.



There is a 10K pull up resistor at the port expander which assists the internal "weak" pull up. The result is a rise time of about 40 μs which is fine given that the minimum period for Flow pulses is 2 ms.



Each time the water meter's nutating disk[13] moves, my spinning magnet[14] follows it. The Hall effect sensor sees this magnetic field and pulls low. The software responds to this high to low transition. To be consistent with other inputs, I use the debounce software to read the Flow signal. Here you see a period of 640 ms.

Give a period of 640 ms we can verify the conversion process works. Each pulse represents 0.028 gallons. Therefore

[13] Here is a good explanation: https://www.youtube.com/watch?v=hxuFuT-RQyI

[14] Refer back to page 25.

$$\frac{1\,pulse}{640\,ms} \times \frac{60{,}000\,ms}{1\,minute} \times \frac{0.028\,gallons}{pulses} = 2.63\,GPM$$

The display showed 2.69 GPM which is a difference of 0.03 GPM or 1 pulse. All readings are ±1 pulse so this is reasonable.

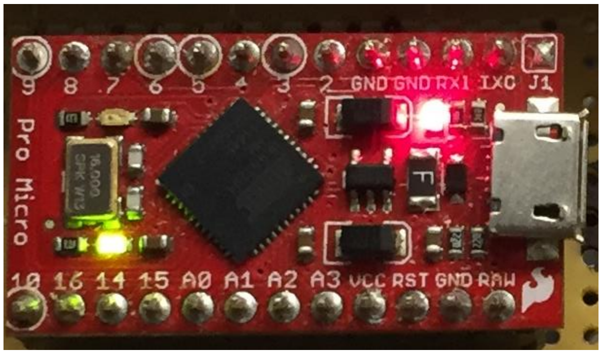
The maximum flow rate that I have directly measured is around 15 GPM. The software can handle up to 63 GPM.

Power

A wall wart supplies unregulated 12V DC at up to 500 mA. It is used directly to power the relay and the piezoelectric beeper. It also feeds a 3 terminal regulator that supplies 5V to the rest of the FMC.

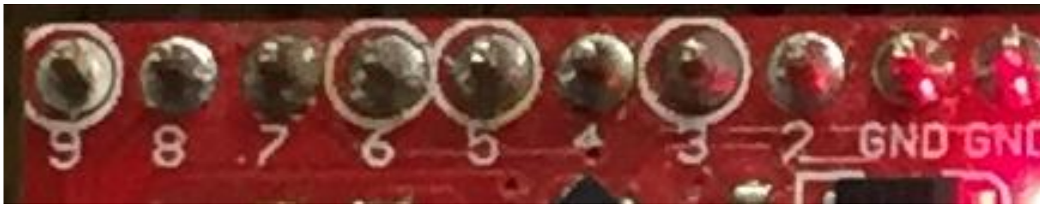
The Arduino

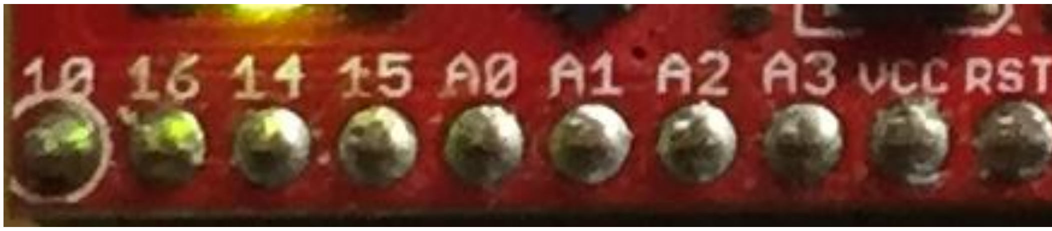
I am using a Pro Micro 16 MHz 5 volt device from Sparkfun.com. For about $20 (2017 price), you get an insane amount of functionality. This device is far more than just a computer. Much has been written about it so I won’t duplicate that effort.


9 8 7 6 5 4 3 2 GND GND RX1 TXO J1
Pro Micro
10 16 14 15 A0 A1 A2 A3 VCC RST GND RAW

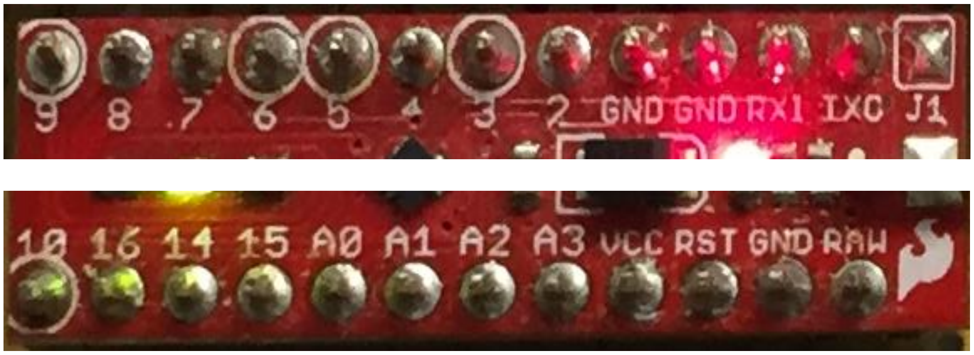
It is important to understand the two basic ways a pin is named: physically and logically.

Physical: its physical location. When looking at the socket that will accept the Pro Micro, pin are numbered starting at 1 and sequentially numbered in a counter clockwise fashion when looking down on the socket from the top. Pin 1 is in the upper right corner.

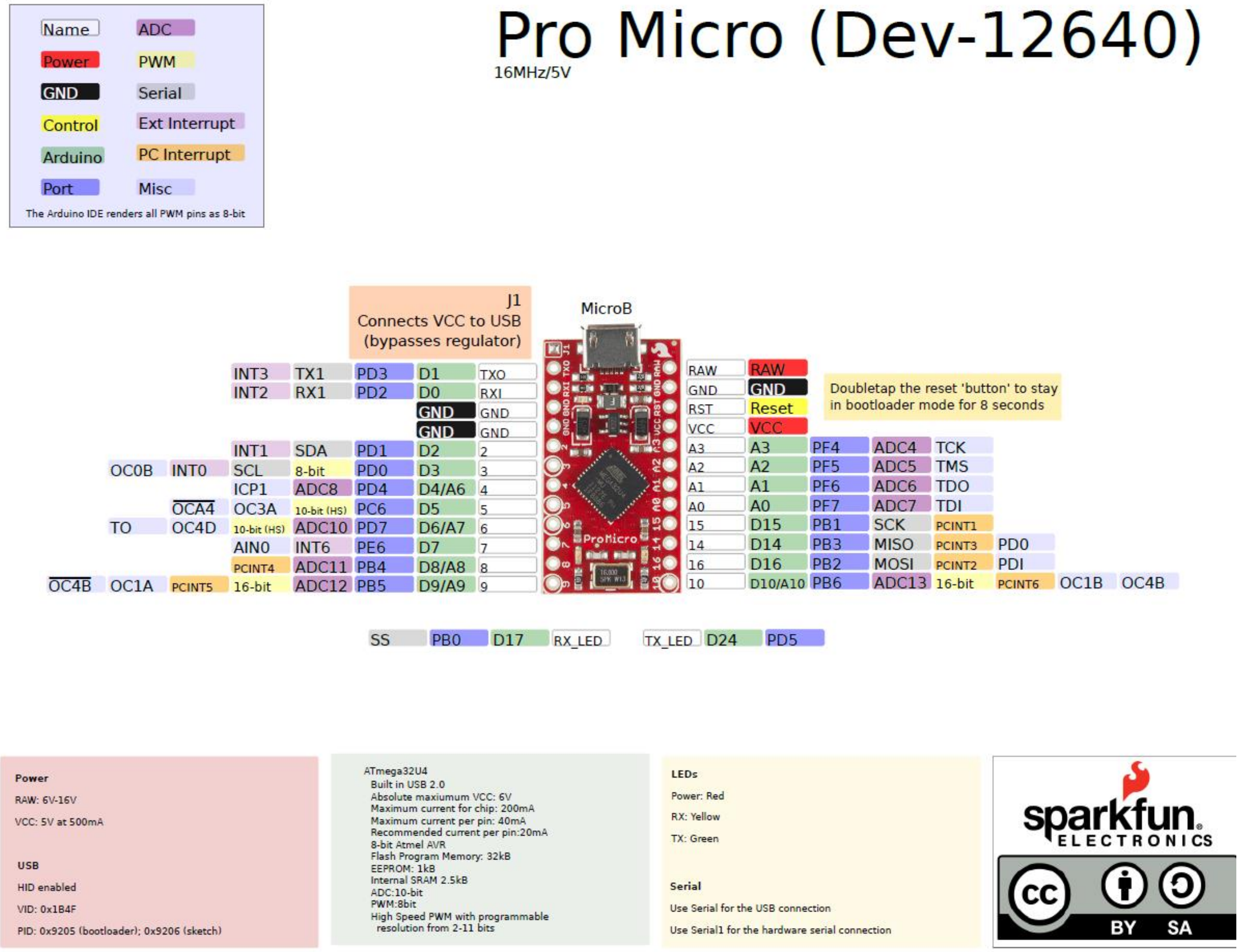




Logical: the logical names of the pins. Some of these names are letters, some are numbers, and the rest are a mix of letters and numbers. For example, physical pin 1 has the logical name TXO. It can get rather confusing at times. Consider physical pin 5 which is logical pin 2. On top of this, you can configure each of these logical pins to be any one of a number of types. This further changes the name. For example, logical pin 2 can be configured to be a Serial Data port (SDA) or to be Interrupt 1 (INT1).



You really need a score card to keep it all straight! Fortunately, Sparkfun has done a masterful job of providing such a card as can be seen on the next page.



Power

RAW: 6V-16V

VCC: 5V at 500mA

USB

HID enabled

VID: 0x1B4F

PID: 0x9205 (bootloader); 0x9206 (sketch)

ATmega32U4
- Built in USB 2.0
- Absolute maxiumum VCC: 6V
- Maximum current for chip: 200mA
- Maximum current per pin: 40mA
- Recommended current per pin:20mA
- 8-bit Atmel AVR
- Flash Program Memory: 32kB
- EEPROM: 1kB
- Internal SRAM 2.5kB
- ADC:10-bit
- PWM:8bit
- High Speed PWM with programmable resolution from 2-11 bits

LEDs

Power: Red

RX: Yellow

TX: Green

Serial

Use Serial for the USB connection

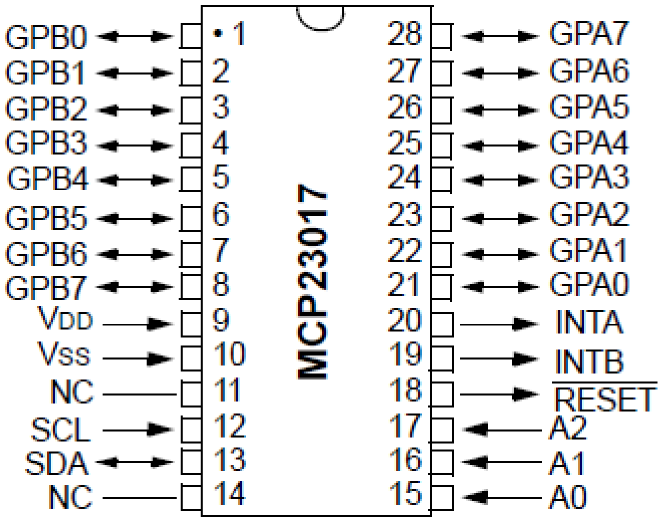
Use Serial1 for the hardware serial connection

The D# values are the logical pins recognized by the software. The physical pins are shown in the above color graphic. I^{2}C uses pin 3 for Serial Clock so we can't use D3. Serial Data is on pin 2 so we can't use D2. There is also a 10K pull up on RESET.

Arduino D# Outputs	**Signal Destination**
0	AZO
4	AA
5-10	Display[15]

[15] See page 37 for details.

One limitation of the Pro Micro is the number of input/output pins. This is easily solved by adding a MCP23017 Port Expander. The device is controlled via I^{2}C using the **S**erial **CL**ock and **S**erial **DA**ta pins.

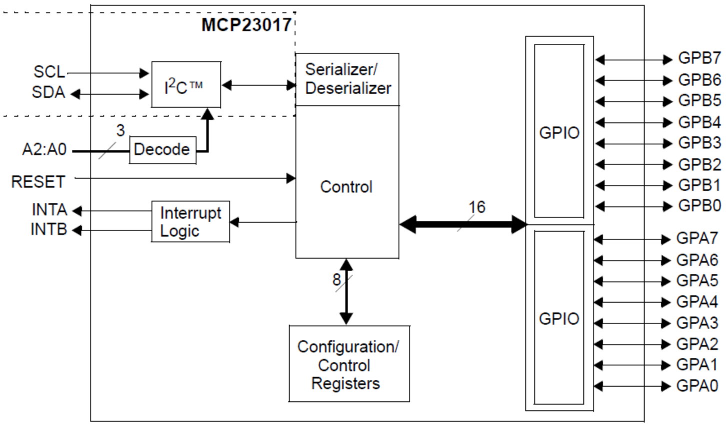


Connect pin #12 (SCL) of the expander to the Arduino SCL pin plus add a 1K pull-up resistor to 5V.
Connect pin #13 (SDA) of the expander to Arduino SDA pin plus add a 1K pull-up resistor to 5V.
Connect pins #15, 16 and 17 of the expander to ground (address selection).
Connect pin #9 (V_{DD}) of the expander to 5V.
Connect pin #10 (V_{ss}) of the expander to ground.
Connect pin #18 (RESET) through a 10K ohm resistor to 5V (reset pin, active low).

From Adafruit: https://cdn-shop.adafruit.com/datasheets/mcp23017.pdf

GPA0-7 and GPB0-7 give a total of 16 input/output pins.

An example of how to use this device can be found at https://github.com/adafruit/Adafruit-MCP23017-Arduino-Library/blob/master/examples/button/button.ino#L1



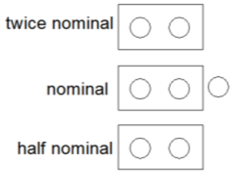
We must include two header files available from Adafruit: wire.h and Adafruit_MCP23017.h. The file “wires.h” sets up the I^{2}C interface.

Port Expander Input Name	**Virtual pin#**	**Physical pin #**	**Signal Source**
GPA0	0	21	Zone 1
GPA1	1	22	Zone 2
GPA2	2	23	Zone 3
GPA3	3	24	Zone 4
GPA4	4	25	Zone 5
GPA5	5	26	Zone 6
GPA6	6	27	PAB
GPA7	7	28	CAB
GPB0	8	1	LS
GPB1	9	2	Flow

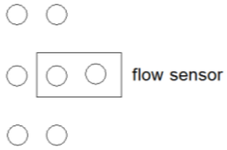
This leaves 6 spare virtual I/O pins (10 - 15) on the port expander. I use three of them as outputs of a flow simulator:

Port Expander Input Name	**Virtual pin#**	**Physical pin #**	**Description**
GPB2	10	3	Half of nominal
GPB3	11	4	nominal
GPB4	12	5	Twice nominal

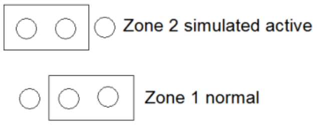
This flow simulator will generate a square wave of 50% duty cycle and output the above 3 signals as long as the program is running. The cycle rate of the loop is divided by the FlowSimulatorDivider value. Additionally, jumpers exist that can pull Zone 1’s and/or Zone 2’s input to ground to simulate it/them going active.



The rectangles represent the position of the jumper blocks. I can select twice nominal flow, nominal, or half.



When not testing, the jumper connect to the flow sensor.

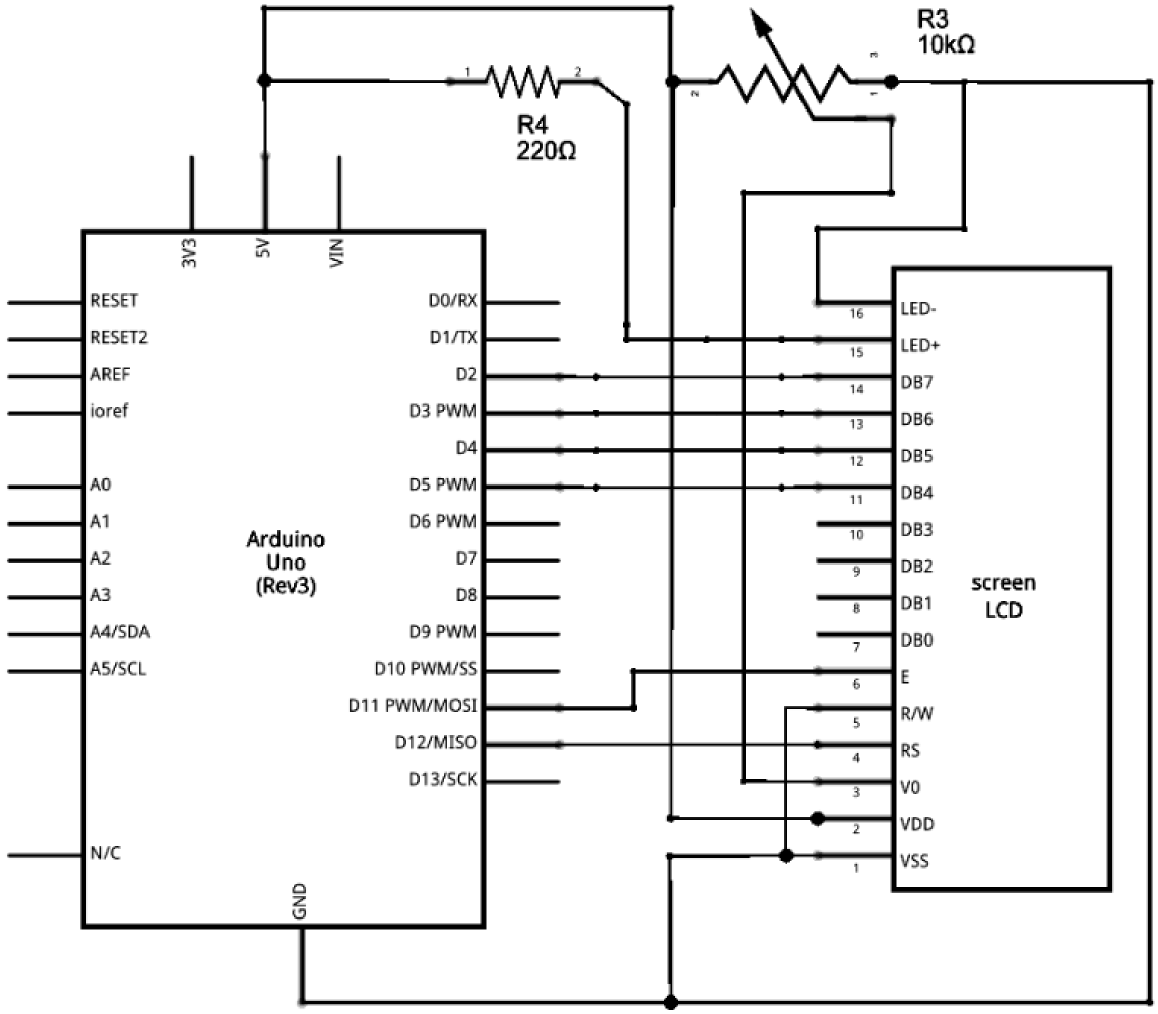


I can also switch between simulate active and normal for zones 1 and 2. As shown, zone 1 is in normal operational mode while zone 2 is simulated active.

This simulation subsystem was invaluable for debugging the code.

The Liquid Crystal Display

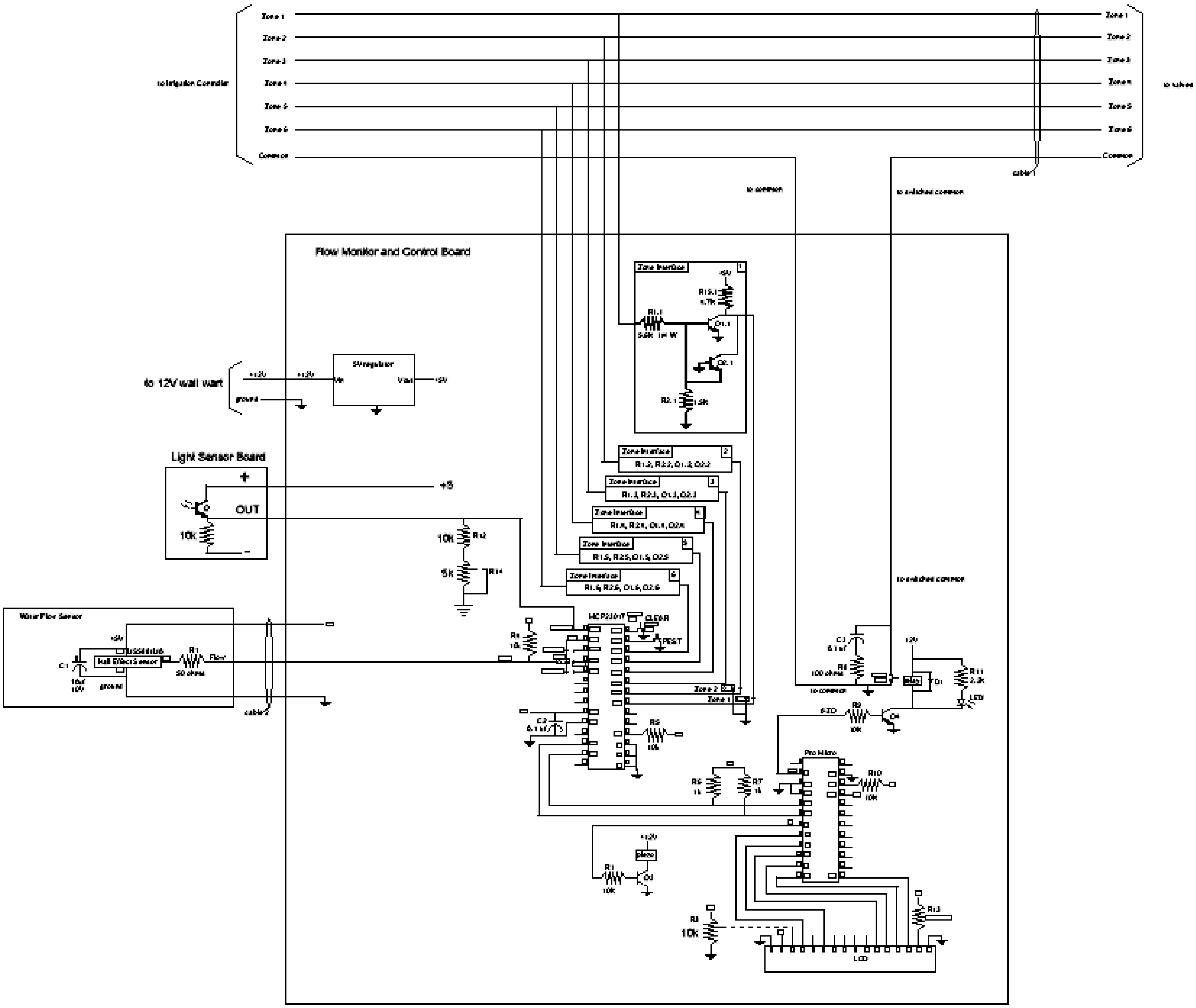
I am using a Sparkfun Basic 16x2 Character LCD - Black on Green 5V for $14. It can be controlled with 6 pins as shown in https://www.arduino.cc/en/Tutorial/HelloWorld?from=Tutorial.LiquidCrystal Support circuits include a 10K ohm contrast control potentiometer and 220 ohm backlight power resistor: https://www.arduino.cc/en/uploads/Tutorial/LCD_Base_bb_Schem.png

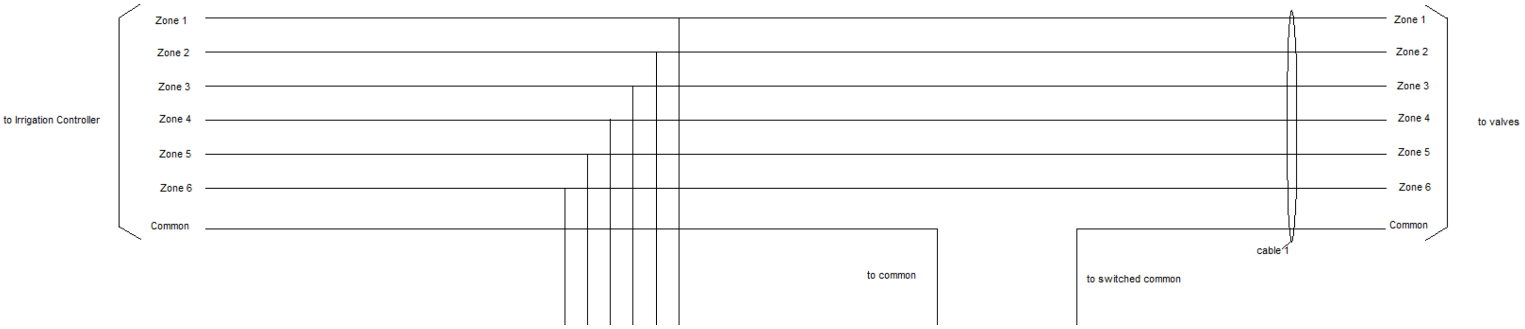


LCD Inputs	LCD pin	Pro Micro Outputs, D#
RS	4	5
EN	6	6
DB4	11	7
DB5	12	8
DB6	13	9
DB7	14	10

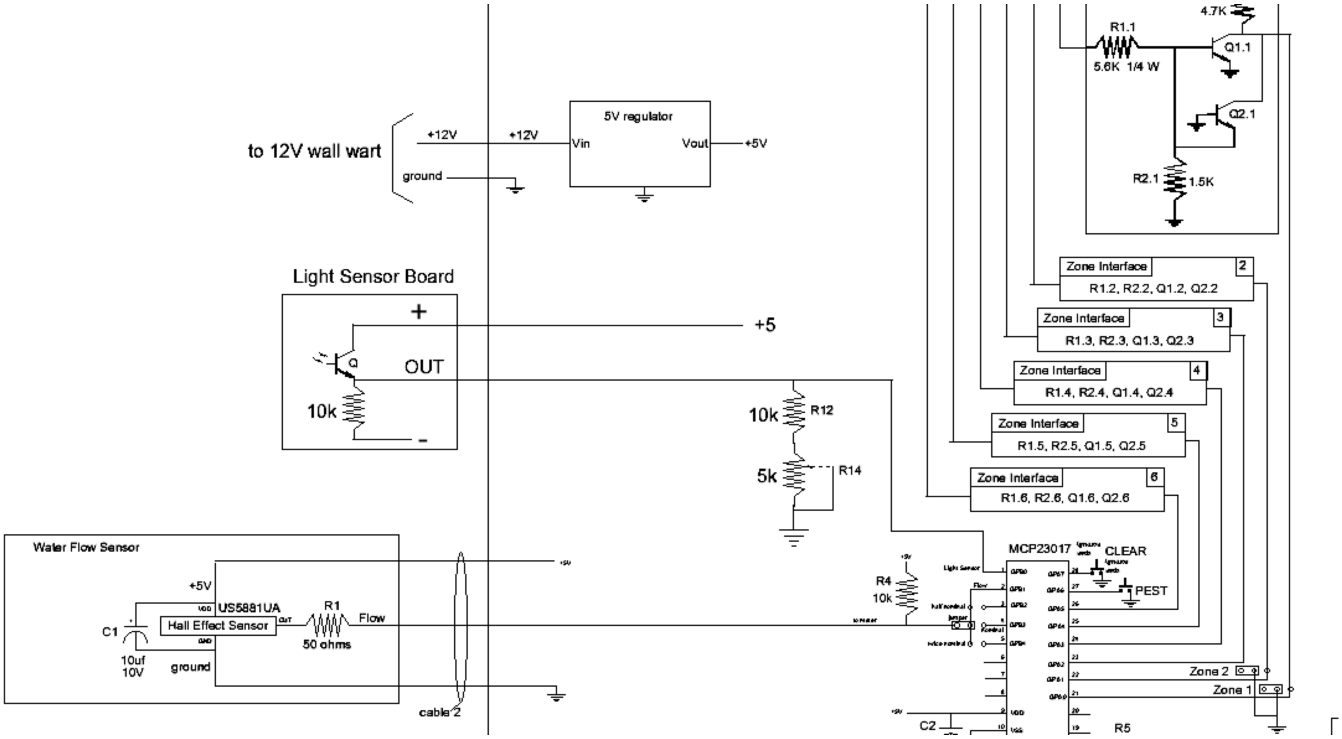
The Full Schematic

Here is an overview.



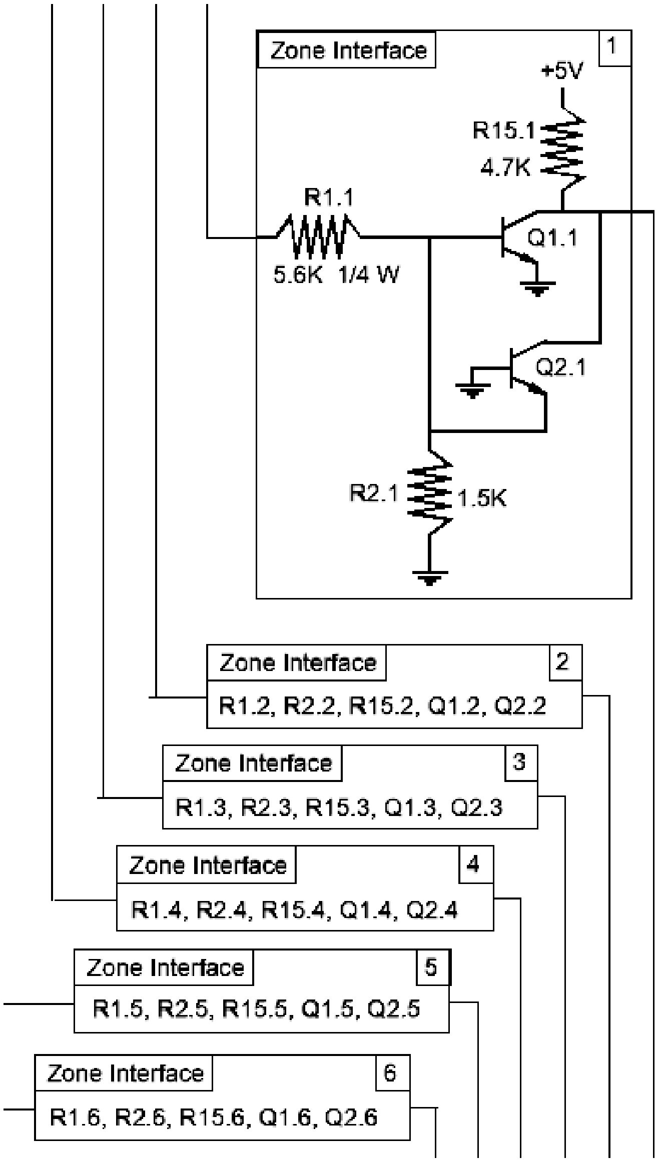


The zone wires from the Irrigation Controller pass through but are monitored. The Common wire goes into the FMC and back out. All of these wires go into Cable 1 that runs out to the zone valves.

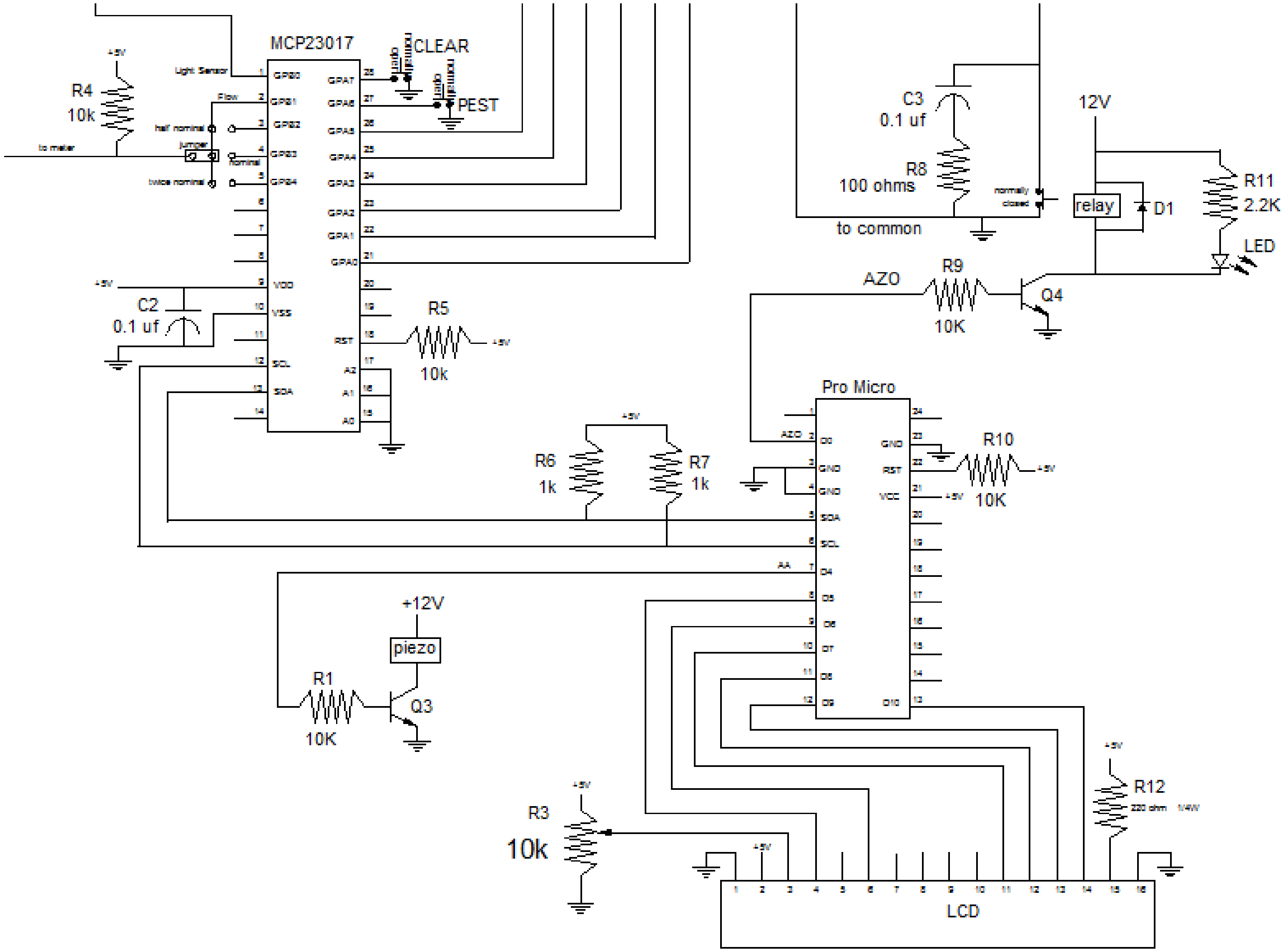


Power from the wall wart feeds the 5 volt regulator along with the relay and piezoelectric beeper, not shown here.

The Light Senor runs through a two conductor cable a short distance to where it can see daylight.


Zone Interface
1
+5V
R15.1
4.7K
R1.1
5.6K 1/4 W
Q1.1
Q2.1
R2.1
1.5K
Zone Interface
2
R1.2, R2.2, R15.2, Q1.2, Q2.2
Zone Interface
3
R1.3, R2.3, R15.3, Q1.3, Q2.3
Zone Interface
4
R1.4, R2.4, R15.4, Q1.4, Q2.4
Zone Interface
5
R1.5, R2.5, R15.5, Q1.5, Q2.5
Zone Interface
6
R1.6, R2.6, R15.6, Q1.6, Q2.6

The six zone interface circuits are identical. Note that zone interface 1 shows all detail and the rest are identical except for the names of the components. When 24 VAC is seen on a zone line, the interface sends a logic 0 to the port expander.



The port expander, MCP23017, outputs test flow signals that are selectable via optional jumpers. It accepts inputs from the clear all alarms and PEST audible alarms buttons.

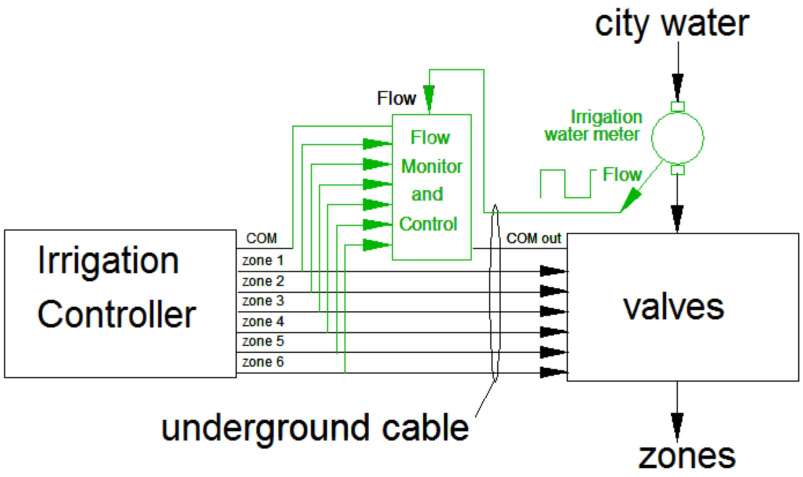
The Pro Micro Arduino controls the port expander via the I^{2}C which consists of the Serial Data (SDA) and Serial Clock (SCL) lines. It drives the piezoelectric beeper via R1 and Q3 plus the relay via R9 and Q4. And finally, it drives the Liquid Crystal Display.

Bill Of Materials

Estimated cost is $80 plus the cost of the Badger flow meter, cable, and enclosure.

Name	**Quantity**	**Description**	**Notes**
R1	1	50 ohms 1/8W	Mouser.com
R1.1 - R1.6	6	5.6K 1/4W	Mouser.com
R2.1 - R2.6	6	1.5K 1/8W	Mouser.com
R3	1	10K 1/8W pot	Mouser.com
R4, R5, R9, R10, R12	5	10K 1/8W	Mouser.com
R6, R7	2	1K 1/8W	Mouser.com
R13	1	220 ohms	Mouser.com
R8	1	100 ohms 1/4W	Mouser.com
R11	1	2.2K 1/8W	Optional
R14	1	5K 1/8W pot	Mouser.com
R15.1-R15.6	6	4.7K 1/8W	Mouser.com
C1	1	10 uf 10V electrolytic	Mouser.com
C2, C3	2	0.1 uf	Mouser.com
Q1.1 - Q1.6, Q2.1 - Q2.6, Q3, Q4	14	BC550B or any general purpose NPN	Mouser.com
D1	1	any general purpose diode	Mouser.com
LED	1	any general purpose LED	optional
RELAY	1	OUAZ-SS112D,900	Mouser.com
LIGHT SENSOR	1	ALS-PT19	Adafruit.com
HALL EFFECT DEVICE	1	US5881UA	Mouser.com
PRO MICRO	1	Arduino	Sparkfun.com
PORT EXPANDER	1	MCP23017	Adafruit.com
PIEZO	1	254-20C6-ROX	Mouser.com
LCD	1	LCD 00709	Sparkfun.com
PUSH BUTTON 1 AND 2	2	any general purpose low power	
PERFERATED CIRCUIT BOARD	2	As needed	
CABLE 1, CABLE 2		As needed	
FLOW METER	1	BADGER MODEL 25	eBay.com
FLOW SENSOR ENCLOSURE	1	See page 34	
FLOW MONITOR AND CONTROL ENCLOSURE	1	Your choice	
5V Regulator	1	833-MC7805CT-BP	Mouser.com
12VDC at 500 mA unregulated power supply	1	Your choice	
jumper	3	Two terminal blocks	optional

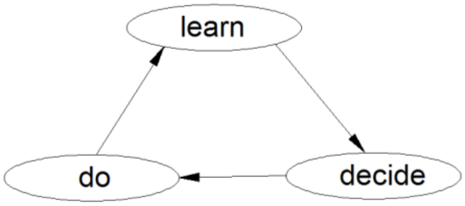
Software



We have 6 zones. Normally no more than one zone is on at a time. Any zone can have a flow that is too low or too high. Depending on the fault, we might be able to stop excessive flow. At any time there can be a change in active zone or in fault state.

The result is similar to a pinball machine with many possible states. For example, zone 1 might be running normally while zone 4 is in underflow and zone 5 is in overflow but controlled. The software must deal with all of these situations. Appendix 5 contains a list of my test cases. All of these cases passed but that does not mean the software is free of bugs.

Overall Software Strategy

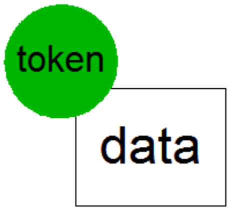


Control constantly circulates between learning what is going on, determining what to do, and acting. This is a “polling” strategy which I prefer over an interrupt driven scheme. With polling, the software’s behavior is known at all times. This makes debugging easier.

An essential requirement is that each cycle through the loop take place faster than the time variant inputs change. Otherwise the real world’s behavior will be missed by the software.

You have seen a bowl of cooked spaghetti. The strands are easily identified but not the ends. Of course, there is no “logic” to it. As my software was evolving, it was starting to look like spaghetti. Debugging and even comprehension quickly became too difficult for me.

Life got better when I re-architected the code into functions. For example, Timer Control contains all timers. If another function wants to start, stop, or read a timer, it must send a flag. In this way, I always know to look at Timer Control if there is a problem with timing.



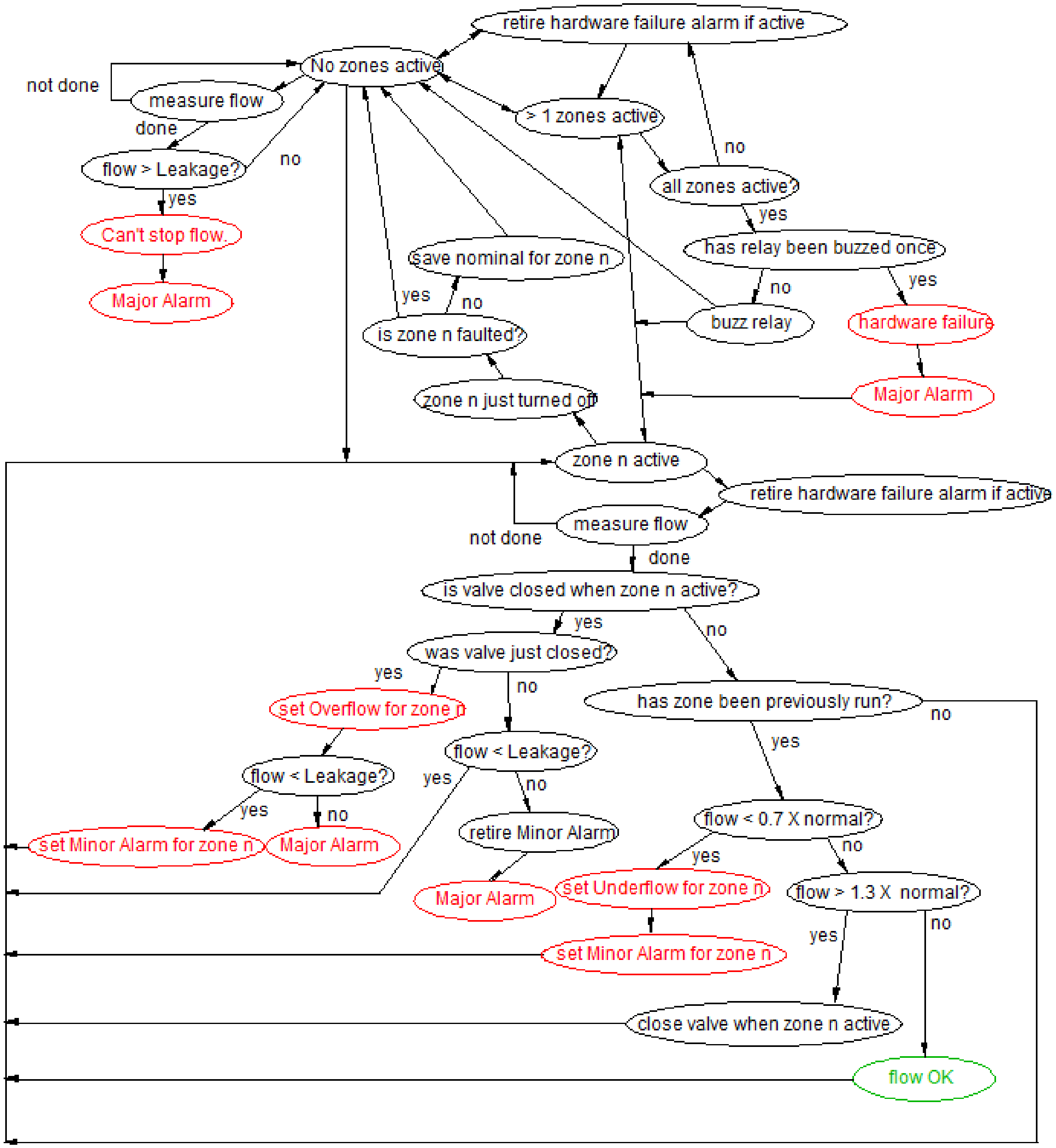
When a flow measurement is taken, I define a “token”. This token says the measurement is valid. If the token is used to inform the user via the LCD, it is still valid. But when the data is used to make a state change, the token becomes invalid and the data is ignored. This scheme prevents old data from being reevaluated while we collect new data.

Many timers are used. In some cases I just need to know that a given timer is running. In other cases I need to know when it starts and when it ends. Flags are defined in the Timer Control subroutine and used by the rest of the code.

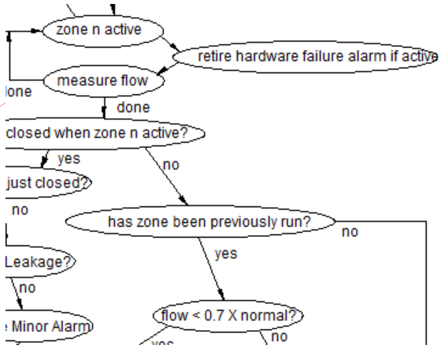
If you make it all the way down to the code, you will see that I have taken full advantage of variables and subroutine naming freedom. These tags have been selected to help me remember what is going on. For example, The subroutine used to get zones states is called GetZoneStates( ). My flag that tells me I just set the inhibit flag active is called JustSetInhibitActive. These long names are removed during compilation so do not take up space in the Arduino’s memory. By using copy and paste, I avoid typing them more than once.

Level 1 Flowchart

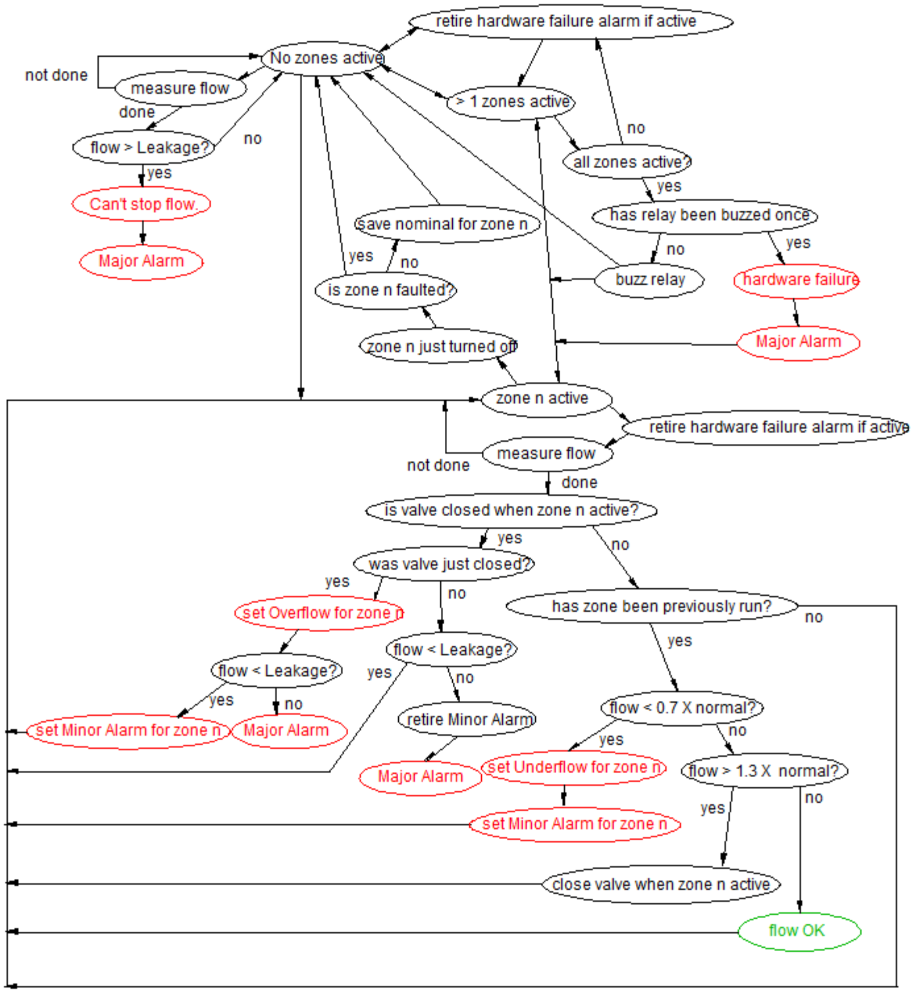
When viewed for the first time, I'm sure there is an urge to just turn the page. If you just want to run the code, there is no need to go further. But if you want to understand the code, here is the first step.



Not shown is display, and audible alarm logic.



When the program starts for the first time, it has no flow data on each zone. After it has run each zone, that data exists.



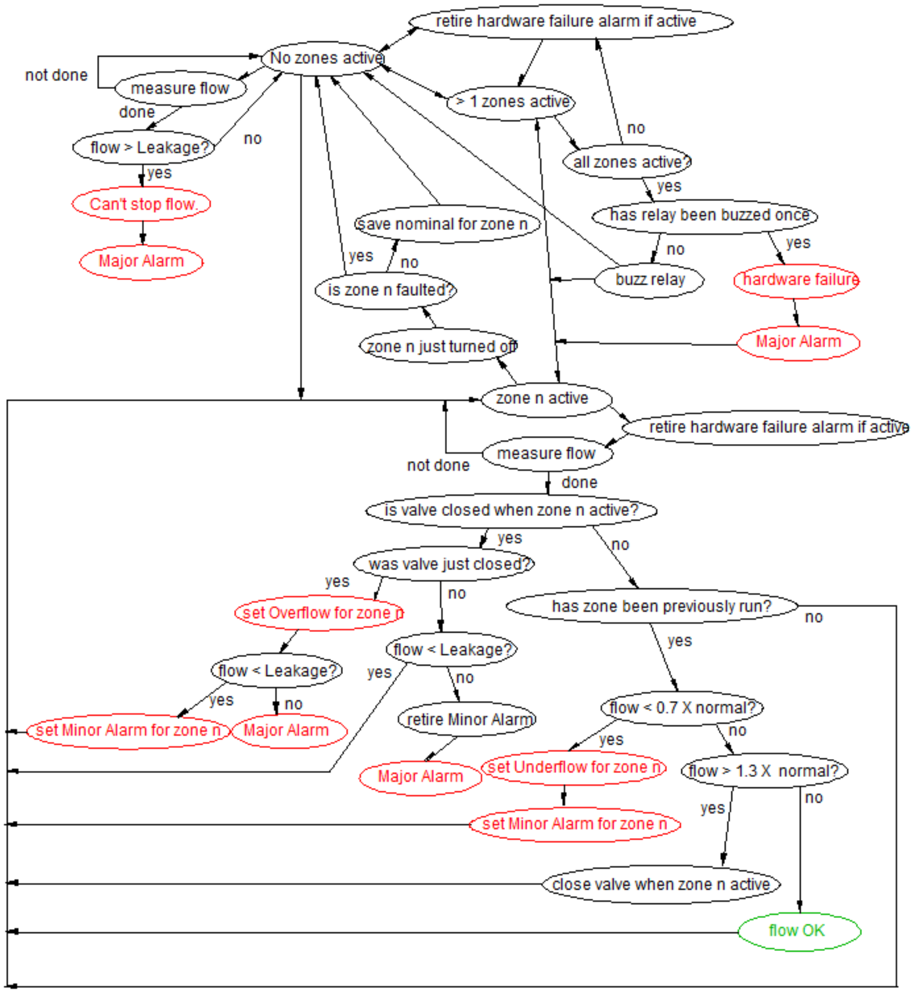
Not shown is display, and audible alarm logic.

Starting at “No zones active”, we go to “zone n active”. The flow is measured for 60 seconds. When done, we see if the zone’s valve was closed due to a previous fault. If it was not, we see if this zone has ever run before. If this is the first time, we have no historical flow data so cannot judge if there is a problem. If it has run before, we know what “nominal[16]” means so start judging the flow. If the flow is below 70% of nominal, we have Underflow and set Minor Alarm for this zone. Then we start the cycle over.

If Above nominal by more than 30%, we close the valve every time this zone is active. Then we start the cycle over.

If the flow is within 30% of nominal, it is ok and we start the cycle over.

[16] "nominal" uses too many characters on my LCD so I use "ref" there.



Not shown is display, and audible alarm logic.

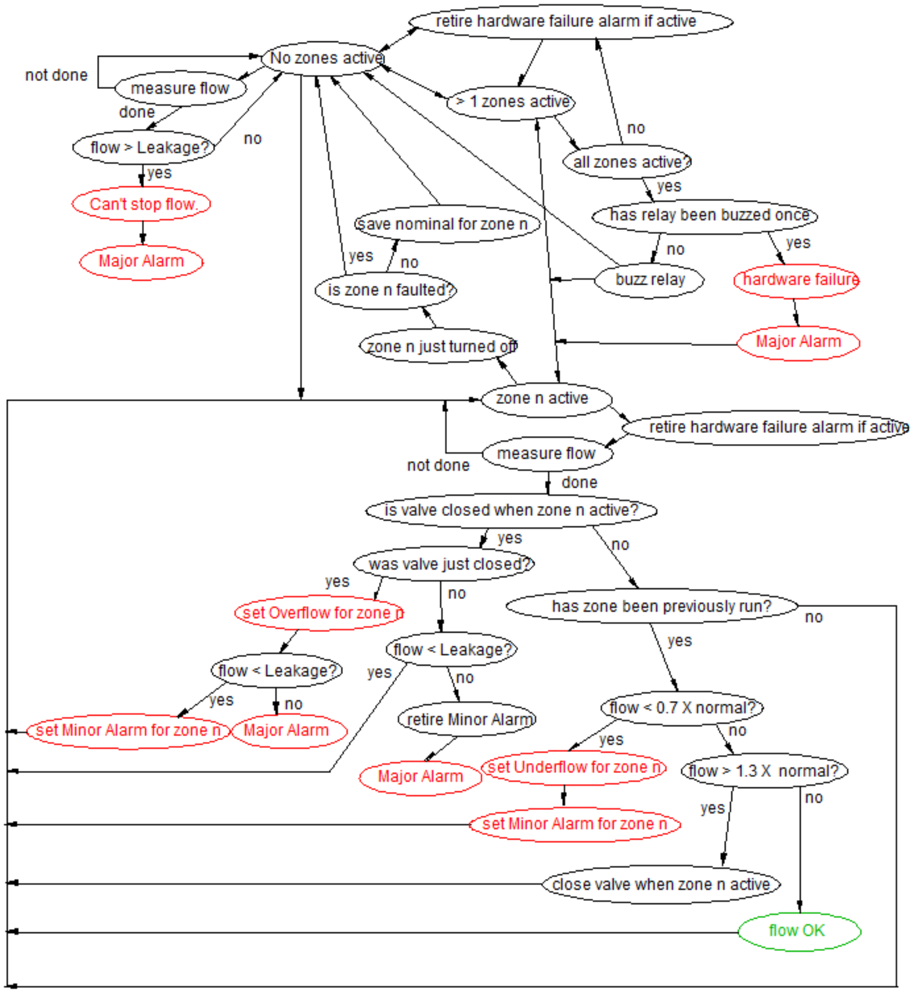
If the valve has been closed when this zone is active, we check if it was just closed. If it was, we say the zone is in Overflow and then see if the flow is below the leakage limit.

If closing the valve did not turn off the flow, we go to Major Alarm and wait for human intervention. We have an uncontrolled flow.

If closing the valve turned off the flow, we call out a Minor Alarm for this zone. As we repeat the cycle, this zone is defined as in Overflow with Minor Alarm.

On the next pass, we again measure the flow and see that the valve has been closed. But now we see that the valve has not ***just*** been closed. If the flow is still less than Leakage, the cycle repeats. If the flow is now greater than Leakage, we retire the Minor Alarm and escalate to a Major Alarm and wait for help to arrive.

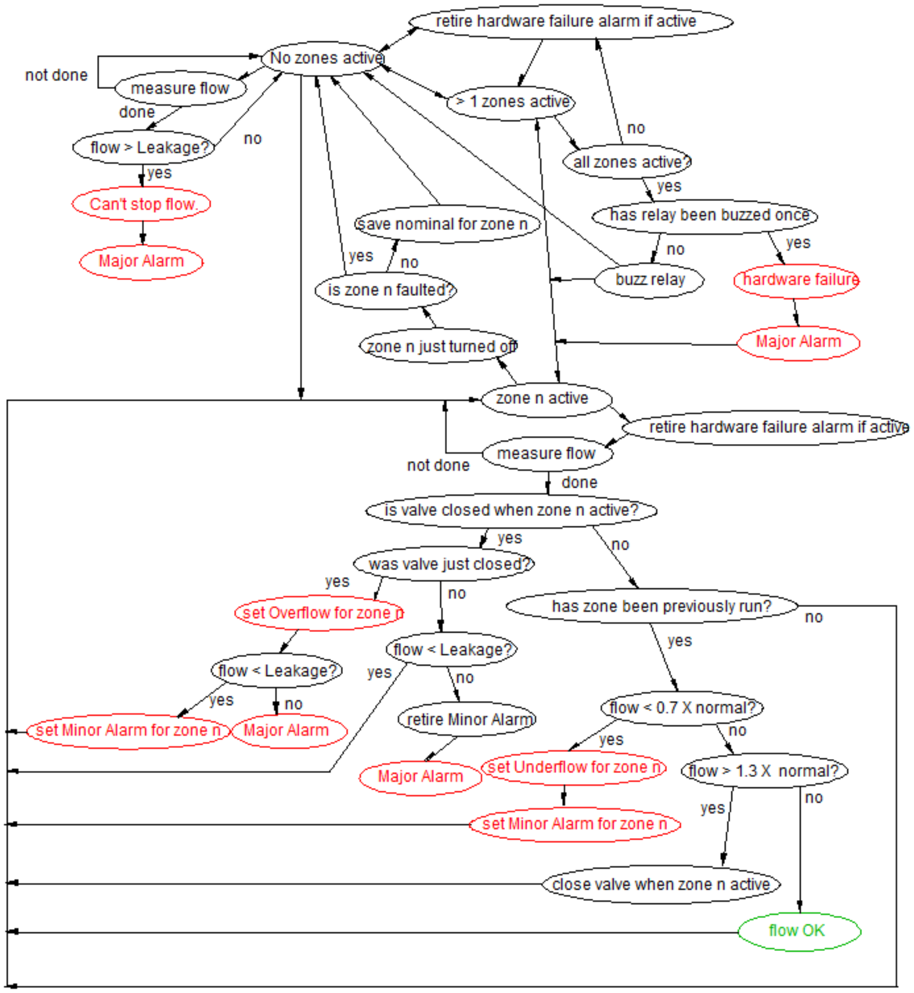
If a zone is in Overflow Minor alarm, every time it runs the FMC blocks the flow.



Not shown is display, and audible alarm logic.

When a zone turns off, we first check if it is in an alarm state. If not, we save the flow data collected while it was running and store that as the new nominal for this zone. If there is an alarm state for this zone, we do not save the flow data.

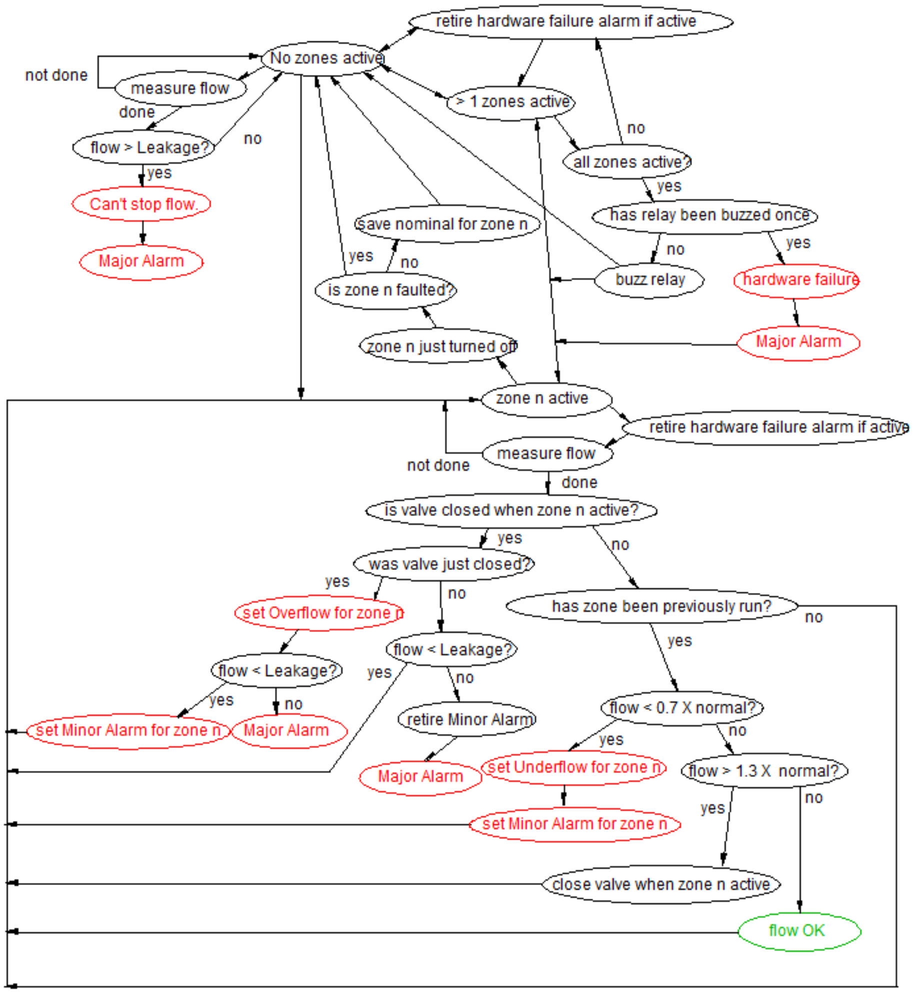
When no zones are active, we constantly check that the flow is less than the leakage limit (green arrow). If excessive leakage is detected, we go to Major Alarm and wait for help.



Not shown is display, and audible alarm logic.

The automatic operation of the RainBird Irrigation Controller only turns on one zone at a time. However, it is possible for a user to turn on more than one zone. In this case, which I call MultiZone, we just wait until less than two zones are on.

If all zones are active at the same time, we probably have a problem with our relay. If so, we turn it on and off quickly 20 times and check again. If we are then seeing less than all zones active, we return to normal scanning. If we still see all zones active, we go to hardware fault Major Alarm. Then we return to scanning just in case the relay corrects itself. If so, the hardware fault Major Alarm is retired.



Not shown is display, and audible alarm logic.

Four secondary functions are not shown in this flowchart.

The first is the display function. See the User's Manual, starting on page 14 for what is shows. See also Appendix 4 which explains how I prevent flickering of the display.

The second secondary function is audible alarms. A Minor Audible alarm sounds for 0.3 seconds and is off for 2 seconds. A Major Audible alarm sounds for 0.1 seconds and is off for 0.1 seconds. At any time, the user can push the PEST audible alarms button and it will bring silence.

The third secondary function clears all alarms. All variables are initialized except for historical flow data. If a fault condition still exists, it will return in 70 seconds.

The forth secondary function clears all alarms and historical flow data. This would be used if a change was made to the flow rate that was more than 30% from nominal. The FMC is powered down, the clear alarm button held down, and power is brought back up. Then the clear alarm button is released.

Major Subroutines

Here you see the main subroutines that are in the loop and the subroutines called from them.

```
GetZoneStates()
    InitializeZone()
    PopulateZones()
    FindActiveZone()
ZonePower()
TimerControl()
    TransientTimeElapseTimer ()
    OneMinuteElapseTimer ()
    MinorPestElapseTimer ()
    MajorPestElapseTimer ()
    MinorCadence()
    MajorCadence()
    AntiFlickerCadence()
ZoneTransitionQ()
    SaveOldZoneFlowMeasurement()
    PrepareForNewZone()
ProcessFlowInfo()
    GetFlowRate()
    DisplayFlowRate()
        SuspectRelay
        MoreThanOne()
        PossibleFault()
        NoZoneOnAndNoDataYet()
        NoZoneOnWithData()
        ActiveZoneNoDataYet()
        ZoneActive()
    JudgeFlowRate()
        MajorAlarm
        MultiZoneQ
        NoFlowDataQ()
        NoHistoryQ()
        SmallFlowQ()
        UnderflowQ()
        OverflowBeforeInhibitQ()
        ImmediatelyAfterInhibitFaultQ()
        AfterInhibitStableFaultQ()
        LeakageNoZonesActiveQ()
    RelayErrorControl()
    AlarmControl()
        CleanUpPrematureAlarmExitQ()
        DisplayFaultDetails()
            RelayErrorQ()
            CABpressed()
            SoftwareError()
            CannotStop()
            Trying()
            HaveMinor()
        ManuallyClearAlarmQ()
        PestActiveAlarmQ()
        AudibleAlarmProcessing()
```

GetZoneStates() looks at all of the zones and reports back which ones are active. To do this it calls three subroutines:

InitializeZone()
PopulateZones()
FindActiveZone()

ZonePower() turns the relay on and off under the control of an array of Inhibit flags. If the Inhibit flag for the active zone is true, then the relay is powered up and power is removed from this zone's valve.

TimerControl() handles all timers including the transient timer, the one minute timer, PEST timers, and audible alarm cadence timers. Timers that only run when needed are initiated by telling them to start. When they start, the clear the start flag and raise a running flag. When timed out, the running flag is lowered.

ZoneTransitionQ() looks at the previous active zone and the current active zone to see if there has been a zone transition. If so, it looks to see if the previous active zone had a fault. If not, it saves the flow data as the new historical flow. If there was a fault, it does not save the flow data. Then it prepares for the new zone being active.

ProcessFlowInfo() performs three major tasks. First it measures the flow rate, then it displays the flow rate, and finally, it judges the flow rate to see if it is normal or faulted. The subroutine SmallFlowQ() checks the historical flow for the active zone in order to prevent a division by 0.

RelayErrorControl() displays a message if the relay is stuck.

AlarmControl() handles all alarms associated with flow. This includes displaying fault details, responding to a manual clear alarms request, responding to a PEST audible alarm request, and controlling the audible alarm.

Software Structure

Although it is possible to have multiple files that together hold the Arduino software, I have chosen to put it all in one file to minimize confusion and error. Separate files must be loaded is a specific order to link correctly.

The following is an outline of how I ordered the segments of the program. Note that variables and constants defined outside of functions are accessible by all functions which is why you see so much going on before we reach Setup.

- Program name and version number - they are displayed at start up on the LCD
- #include - pulls in libraries that define hardware. In my case I have EEPROM, an LCD display, and a port expander chip that uses I^2C.
- Constants and variables
    - MCP23017 port expander inputs
    - Arduino Outputs
    - States of the hardware and software
    - Timers
    - Rate of change of flow parameters
    - Flow level parameters
    - Input states like "Day" versus night and a button being pushed or not
    - Faults that are recorded and processed
    - Software error descriptions
    - LCD flicker reduction parameters
    - Diagnostic and built in simulator parameters
- Hardware Initialization - port expander and LCD
- Void Setup - this is where code is placed that only executes once. All of the input and output pins in the hardware are defined here. We also look at the pushbuttons to determine if the user wants all historical data erased or if they want flow measuring mode enabled

- Void Loop - a collection of high level subroutines that hold the major functional blocks resides here. These subroutines are made up of lower level subroutines defined next.
- Lower level subroutines - these are built from subroutines that are at the lowest level.
- Lowest level subroutines - narrowly defined functions that act as my custom made program language.
- End of file

An important fine point:

Looking at the code, you may spot something odd in my print commands to the LCD. The standard command would be

```
lcd.print (“All alarms”);
```

but I wrote

```
lcd.print (F("All alarms"));
```

You are looking at the F() subroutine which I found on an Adafruit forum[17]. F() tells the Arduino compiler to keep the string “All alarms” in program store rather than duplicating it in Static Random Access Memory (SRAM). Without this subroutine, the program runs out of memory used to store variables and strange things happen.

For example, I was checking a small block of code because it did not seem to execute correctly. A series of Serial.println statements told me what was going on. I could see command 1 being reached followed by command 2. But rather than command 3 I saw that a subroutine was called. That code did not call this subroutine. It was corrupted memory caused by the SRAM being corrupted.

[17] First try the URL: https://learn.adafruit.com/memories-of-an-arduino/optimizing-sram but if this link is broken, search using " Adafruit managing sram F()"

Acknowledgements

Very special thanks go to my wife, Donna, for putting up with me constantly at my computer writing and debugging the software.

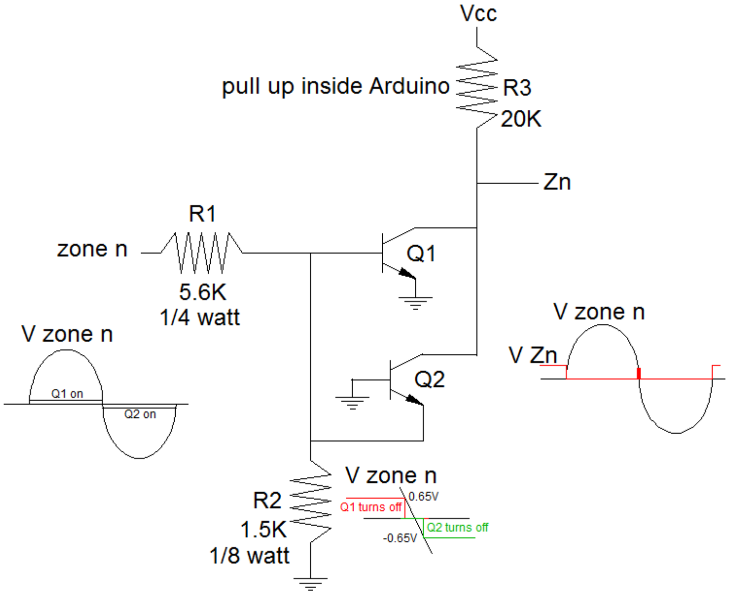
Thanks to Misol from Instructables for the invaluable software plus instructions on how to set up IFTTT.

I welcome your comments and questions.

If you wish to be contacted each time I publish an article, email me with just "Article Alias" in the subject line.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org

Appendix 1: Zero Crossing Pulse Width



The input voltage is specified at 24V RMS but I measured 27V RMS. Assuming 24V RMS, that is a peak of $\sqrt{2}\ \times 24V\ RMS = 34V_{peak}$. This means that $V_{zone\ n}\ = 34 \sin \omega t$ where $\omega = 2\pi f\ = 2\ \times 3.14\ \times 60\ Hz = 377\frac{radians}{second}$. Assuming Q_1 turns on when $V_{\text{zone n}}$ is more than 2V, we can solve for t and see the width of the spike. Be sure your calculator is set to radians and not degrees for the following calculations.

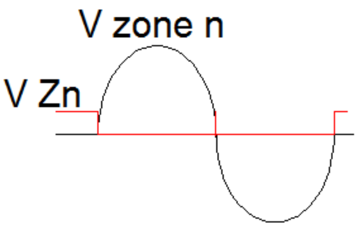
$$2V\ = 34V\ \ \sin 377t$$

$$\frac{2}{34}\ = 0.06 =\ \sin 377t$$

$$377t =\ \sin^{-1} 0.06$$

$$377t = .06$$

$$t = 160 \text{ microseconds.}$$



AT t = 0, $V_{zone\ n}\ = 0$. At t = 160 microseconds, it is at +2V and Q_1 turns on.

The 60 Hz sine wave has a period of 16.67 milliseconds. This means it goes through half of its cycle in 8.33 milliseconds. 160 microseconds before 8.33 milliseconds, Q_1 has turned off. Then 160 microseconds after 8.33 milliseconds Q_2 turns on. This says I will have both transistors off for about (2 × 160 =) 320 microseconds every 8.33 milliseconds. With both transistors off, the digital output goes high. So if the software sees a logic 1, it just has to wait more than 320 microseconds and take another reading to be sure it didn't hit this crossover zone.

Appendix 2: EEPROM Map

The Electrically Erasable Programmable Read-Only Memory (EEPROM) present inside the Arduino device will hold two seven member arrays of data.

The first array is ZoneNeverRun. It is initialized to all elements true. As each zone is run for the first time and historical flow data collected, that element will be set to false. Each element is defined as boolean so takes one byte.

We have:

ZoneNeverRun[i] with i equal any integer from 0 to 6. It occupies EEPROM at memory locations 0 through 6.

To write to this array, we will use the subroutine WriteZoneNeverRun( ). To read this array, we will use the subroutine ReadZoneNeverRun( ).

The second array is HistoricalFlow[i]. It is not initialized. Instead, we only look at a given element when ZoneNeverRun[i] says HistoricalFlow[i] has valid data. As each zone is successfully run, the measured data is written into this array. Each element is defined as float so takes four bytes.

We have:

HistoricalFlow[i] with i equal any integer from 0 to 6. It will occupy EEPROM at memory locations 7 through 28.

To write to this array, we will use the subroutine WriteHistoricalFlow( ). To read this array, we will use the subroutine ReadHistoricalFlow( ).

Appendix 3: Arduino Compilation Error Experiences

It has been a while since I programmed an Arduino so ran into many “newbie” problems. The most confusing were the error messages when my code was not in the sketch book folder. I saw many errors due to files I just included and even one I didn’t include. They went away once I moved my code to the sketch book folder.

After working through way too many typos in my program, I ran into two compilation errors related to code supplied by Sparkfun and Adafruit.

The first error was

```
“main.cpp:43 undefined reference to ‘setup’
Collect2.exe: error: ld returned
1 exit status
```

A search of the web turned up others with this problem plus a solution that worked for them. Going to the file main.cpp line 43 I found

```
setup( );
```

I was instructed to add “void”

```
void setup( );
```

After switching to Administrator mode, I was able to save the change.

The second error was

```
EEPROM.h:43:30: warning: type qualifiers ignored on function return type [-Wignored-qualifiers]
operator const uint8_t( ) const       { return **this; }
```

I learned that EEPROM.h had two bugs in it. On line 43 was

```
operator const uint8_t( )
```

The expert said to remove “const”.

I then was instructed to go to line 92

```
operator const int( ) const
```

and remove the first “const”. The modified file was then saved. The compilation was then error free.

Appendix 4: Anti-flicker

The software is constantly running in a loop. If a given LCD screen is written, it will be re-written on each pass. This is both bad and good. It is bad because doing many updates each second causes the LCD to flicker annoyingly. However, it is good because any corruption of the LCD's displayed data is quickly corrected. The trick is to do updates often, but not too often. I have found that an LCD refresh every 10 seconds is about right.

The strategy I chose was to maintain a unique name for each LCD screen plus have a global flag that presents the opportunity to do an update only every 10 seconds. This flag is part of the TimerControl() subroutine.

We have two types of LCD screens. The simpler one is just text. I can describe it with a constant. For example:

```
const int TryingToClose = 25;
```

TryingToClose has been assigned an arbitrary but unique value of 25.

I also have screens that display text that doesn't change mixed with numbers that do change. In these cases, I assign them an arbitrary but unique value that has the changing number added to it. For example, NoZoneOnWithGPMConstant is assigned a unique number to represent the No Zone On screen. I generate the complete screen name by adding the displayed GPM with the result saved as NoZoneOnWithGPMC.

I keep track of the current LCD screen with the variable lcdNowDisplaying.

When the software is about to print a screen, it checks if the new screen differs from the old screen or if it is time for an update. If either is true, the LCD screen is refreshed. The one exception is when the software has closed the valve and is waiting to see if the flow is now near 0. While displaying the "Trying to close valve" screen, I block all non-alarm screens. Otherwise, I would get alternating screens which is unreadable.

Appendix 5: Test Cases

The following Cases are designed to fully test the software so are detailed. There were developed in parallel with creating the flow charts and thinking about how my irrigation runs and fails.

“Start” means all variables have been initialized. This is done by powering off, holding down the clear alarms button, powering on, and then releasing the button.

When a zone is first turned on, the software waits 10 seconds for the flow to stabilize. It then takes 60 seconds to measure the flow. I am using the built in flow simulator which generates half nominal, nominal, and twice nominal flow rates.

In all cases, m does not equal n.

Non-fault Cases

1. Start to no zones on for 65 seconds.
2. Start to zone n on
3. Zone n on with nominal flow. Zone n off. Zone n on with nominal flow. See that nominal flow becomes reference. Turn off power. Hold down Clear all alarms button. Power up. See that historical data removed. Zone n on with nominal flow and see that ref is - .
4. Zone n on. Nominal flow for 1.5 minutes. Zone n off. Zone m on. Nominal flow for 1.5 minutes. Zone m off. Zone n on for 75 seconds. Zone n off. Zone m on for 75 seconds. Zone m off.
5. MultiZone (more than one zone on)
    a. Before start have zones n and m on. Start.
    b. Zone n on. Zone m on. Zone m off.
    c. Zone n on. Zone m on. Zone n off.
6. No zones on to more than one zone on
7. No zones on to zone n on for first time with nominal flow. See that ref is - .
8. No zones on to zone n on not first time. See that ref has nominal after it.

Fault Cases

A. Single fault
   1. No zones on and then flow > leakage so go to “Can’t stop flow.” Major. Press Clear All Alarms.
   2. Zone n on with underflow. Then pest.
   3. Zone n on with underflow. Zone m on normal flow.
   4. overflow
      a. Zone n on with overflow minor .Then pest. Zone n off. Zone n.
      b. Zone n on with overflow minor .Then pest. Zone m on with nominal flow for 75 seconds. Zone m off. Zone n on.
      c. Zone n on with overflow major. Then pest. Zone m on. Zone m off.

B. Sequential Double fault on same zone
   a. underflow
      1. Zone n on with underflow. Then pest. Then overflow minor and should hear minor audible alarm. Then pest. Then clear all alarms.
      2. Zone n on with underflow. Hear minor. Then pest. Then overflow major. Hear major audible alarm. Then pest. Then clear all alarms.
      3. Zone n in underflow. Hear minor. Then pest. Then zone n off; zone m runs normally.
      4. Zone n in underflow. Hear minor. Then pest. Then zone n off; zone m goes into underflow. Hear minor. Then pest.
   b. Zone n on with overflow minor. Hear Minor audible alarm. Then pest. Then overflow major. Hear major audible alarm. Then pest.

C. First zone faults and then turns off. Second zone on and faults.
   1. Underflow
      a. Zone n in underflow and then zone n off; zone m goes into overflow minor
      b. Zone n in underflow and then zone n off; zone m goes into overflow major
   2. Overflow
      a. Zone n in overflow minor and then zone n off; zone m runs normally
      b. Zone n in overflow minor and then zone n off; zone m goes into underflow
      c. Zone n in overflow minor and then zone n off; zone m goes into overflow minor
      d. Zone n in overflow minor and then zone n off; zone m goes into overflow Major

   e. Zone n in Overflow Major. Turn off zone n. Turn on zone m nominal flow. Wait 1.5 minutes. Zone m off. Should stay in Overflow Major.

3. Interrupted flow measurement
   a. Zone n on. Flow < 0.7 X nominal. While measuring flow, turn on zone m. Then turn zone m off. Zone n should return to start of measuring.
   b. Zone n on. Flow = nominal. While measuring, turn on zone m. Then turn zone m off. Zone n should return to start of measuring.
   c. Zone n on. Flow > 1.3 X nominal. While measuring, turn on zone m. Then turn zone m off. Zone n should return to start of measuring.
4. Interrupted Inhibit processing
   a. Zone n on. Flow > 1.3 X nominal. When valve first closed, zone n off. Flow = nominal. Zone m on.
   b. Zone n on. Flow > 1.3 X nominal. When valve first closed, Zone m on. Zone m off. Zone n should go to Overflow Minor.

Appendix 6: Sending Emails

After the system had been in service for a few years, I decided to add the ability to send emails. This was accomplished with new hardware and new software.

The Initial Design

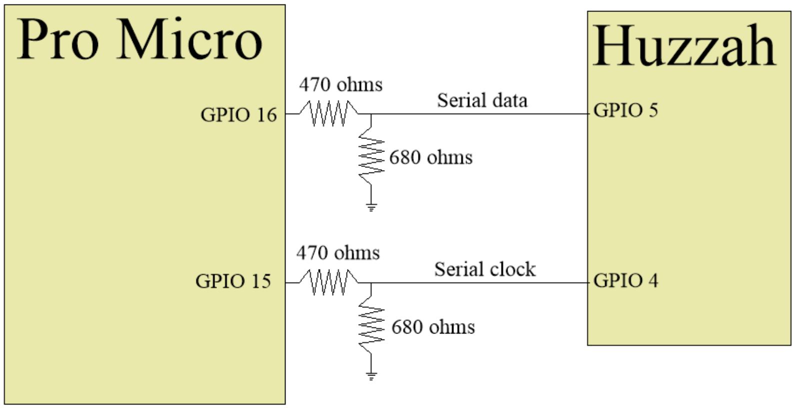
The hardware consists of an Adafruit Huzzah that contains an ESP8266. It costs about $10.

A small amount of new software has been added to the Pro Micro. New software runs on the ESP.

A critical part of this software came from an Instructable. I do not understand all of this code but the author, Misol, made it clear what to change to adapt it to my needs. As of 7/4/2020 https://www.instructables.com/id/ESP8266-to-IFTTT-Using-Arduino-IDE/#discuss worked. Thank you Misol!

The email is generated by the company called IFTTT (If This Then That). Since only I am receiving this email, there is no charge.

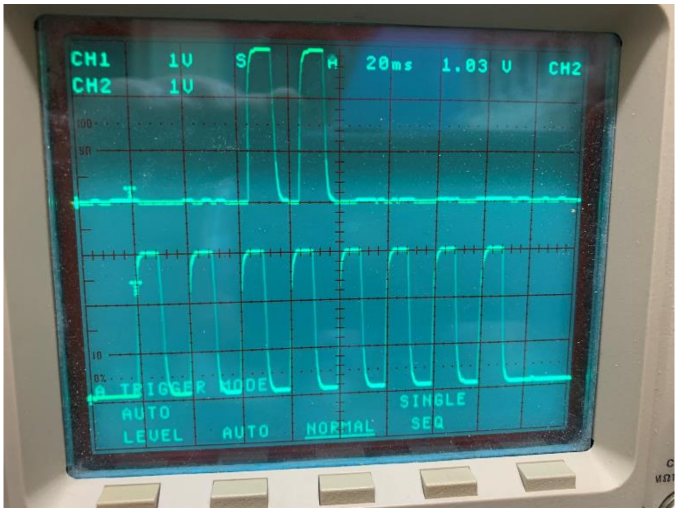
The Pro Micro outputs data via a homemade serial link to the Huzzah. GPIO 15 is my serial clock and 16 is my serial data. The Pro Micro runs at 5V so these signals are bounded by 0 and 5V.



The Huzzah runs on 5V[18] but is not 5V logic tolerate. I therefore had to add a voltage divider on each signal path. It accepts the serial clock on GPIO 4 and serial data on GPIO 5. Each divider consists of a 470 ohm resistor from a Pro Micro pin to an Huzzah pin. A 680 ohm resistor connects from each Huzzah pin to ground. When the Pro Micro puts out 5V, the Huzzah will see, nominally, 3V. These low values of resistance where chosen to minimize the effects of input capacitance which will slow the falling edge of the clock.

[18] The Huzzah has an on board voltage regulator that accepts 4 to 6V and outputs 3.3V to run the board.

Only one byte must be sent from the Pro Micro to the ESP. I chose to write subroutines to support this function because they are simple and, in this way, fully understood by me. That is invaluable during debugging. The data rate can be very slow given that, typically, only 8 bits will be sent per day.



Notice that my clock has a period of 40 ms. It takes 8 x 40 ms = 320 ms to send one byte. That translates to a blindingly fast 0.000003 megabytes per second.

This delay is of no consequence given that it triggers an email to the user who, at most, will place a phone call. Even a few minutes of delay would not matter. This slow rate means that no special realtime coding is needed to handle the clock and data.

The byte is defined in the Pro Micro, sent to the ESP, which sends corresponding text to IFTTT. An "applet" in my IFTTT account sends this text to the defined email address.

The byte is broken down into fields:

State	**Sub-state**	**zone**
System normal [00]	heartbeat [00]	Null [0000]
System normal [00]	Just powered up [01]	Null [0000]
System normal [00]	All alarms cleared [10]	Null [0000]
System normal [00]	Alarm Silenced [11]	Null [0000]
Minor alarm [01]	Underflow [01]	1-6
Minor alarm [01]	System failure [10]	Hardware [0000] or software [0001]
Minor alarm [01]	Overflow [11]	1-6
Major alarm [10]	Can't stop flow [00]	No zones active [0000]
Major alarm [10]	Can't stop flow [00]	1-6
Maintenance [11]	Historical data[00]	[0000]

A Minor alarm triggered by Overflow in zone 3 is 01110011.

Correcting Deficiencies

Murphy's Law is my constant companion. A few months ago I was out of town for a week. Upon arrival, I realized that I was not receiving my morning status emails. I then realized that it had been a few days before leaving that the emails had stopped. To address this second problem, I generated an automatic reminder on my phone to verify I received this email each day.

Once I determined I had a failure, I needed to add fault recovery to the system. This development did not start with a full set of requirements. I developed what I thought I needed, found it lacking, and then added more code. The key is that I held to the software architecture already defined. My final design has these feature:

- Any function can send a byte that will be translated into predefined text by the ESP. Under non-fault conditions, this is once per day. A fault occurs every few months.
- Once an hour, the Pro Micro pings the ESP and gets an acknowledgement (ACK). It indicates if the ESP is down, WiFi ***was*** down while sending the last message, or Maker ***was*** down while sending the last message. The Maker is IFTTT.
- When WiFi or Maker is down, we send recovery emails until ACK comes back without a failure.
    - A recovery email states that the failed element is now back up. If the email makes it to the user, the failed element must be back up. If the email doesn't arrive, the user is none the wiser.
    - When the fault clears, the last email that encountered a fault is resent.

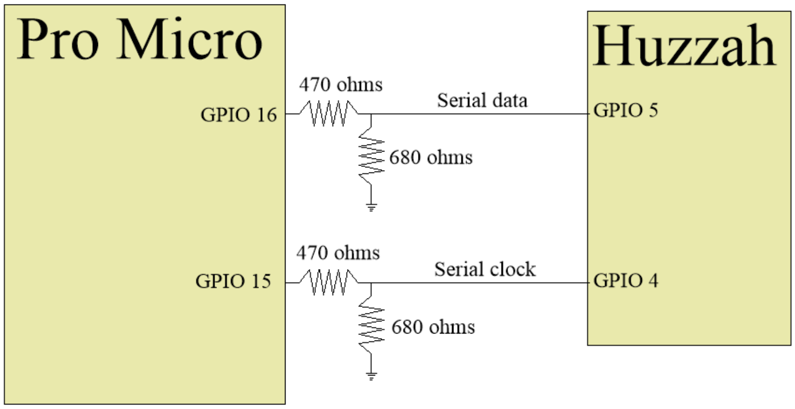
- The FMC's LCD displays three letters in the upper right corner indicating
    - an Acknowledgement Failure (AkF) – the ESP is down
    - WiFi Failure (WfF)
    - Maker failure (MkF).

  These indicators persist even when the fault goes away. This permits me to see intermittent problems. The last fault is the one displayed.

- Any of these failures also generates an audible alarm. The audible alarm consists of two chirps within 0.3 seconds followed by 2 seconds of silence. I chose two chirps so it would not sound like a smoke alarm battery going dead. As with all audible alarms, they can be silenced by the user. Unlike other audible alarms, it remains silenced until the alarm manually is cleared.

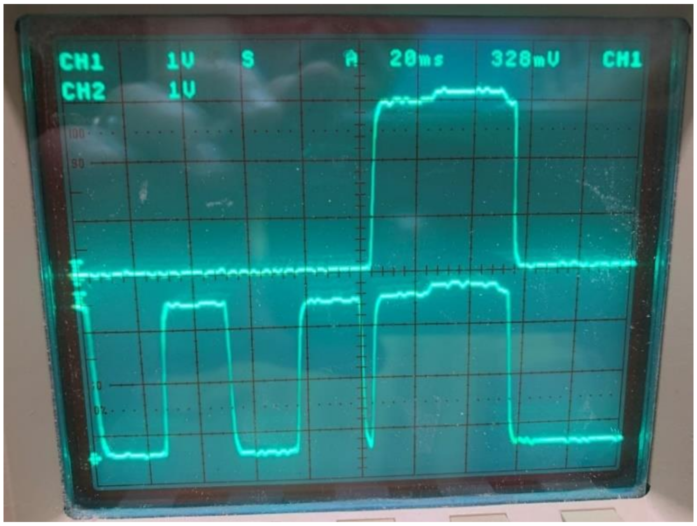
No Hardware Additions

The first challenge was that I didn’t want to add any hardware. I had the Serial Clock and Serial Data interface shown on page 81. The Pro Micro’s pins were set to outputs and the Huzzah’s were inputs. My solution was to have the Pro Micro send the byte and then change its Serial Data and Serial Clock pins from output to input. The Huzzah would receive the byte and change its inputs to outputs. Then it would drive these pin high or low for 50 ms. After this time, the Huzzah would switch back to being an input. The Pro Micro would see this as a logic zero due to the 680 ohms pull down resistors. I empirically found that the Pro Micro could not see a pulse less than about 20 ms so I made the pulse 50 ms wide. This delay, encountered once per hour, has minimal effect on the system’s functionality.



As previously mentioned, the Pro Micro uses 5V logic and the Huzzah uses 3V logic. These two voltage dividers translate 5V logic levels to 3V logic levels as the Pro Micro outputs clock and data to the Huzzah.

The challenge was to get the Huzzah to talk back to the Pro Micro without adding hardware. It turns out that Pro Micro inputs are compatible with 3V logic. So when I change GPIO 5 and 4 from inputs to outputs, The Pro Micro can switch from outputs to inputs and receive these state.



This ‘scope picture shows the tail end of the command from the Pro Micro to the Huzzah. The top trace is serial data and the bottom is serial clock. At the center of the screen, the last clock pulse goes low. The Huzzah decodes the command and generates the ACK. Since all is well, it sends back “11” which means the Huzzah is up plus WiFi and the Maker were up the last time we sent a message to the Maker. These pulses stay high for 50 ms which is enough time for the Pro Micro to read them. When the Huzzah is done sending these pulses, it reconfigures back to being inputs. This is the same as sending logic 0 as far as the Pro Micro is concerned. After the Pro Micro sends the last bit of the command, it changes to inputs and waits 25 ms. Then it reads Serial Data (SDA) and Serial Clock (SCL). After waiting 30 more ms, it reverts to outputs on these pins.One minor mystery is that the logic ones from the Huzzah start at 3.0v and rise to about 3.2V after 20 ms. It is harmless but I can't explain this behavior.

SDA	SCL	
0	0	ESP is down (i.e. no response from the ESP)
0	1	ESP is up but WiFi is down. Therefore, Maker can't be reached
1	0	ESP is up, WiFi is up, but Maker is down
1	1	ESP is up, WiFi is up, Maker is up

I have chosen the numerical values of each state so the Pro Micro can read SDA and SCL and then directly convert them to their value:

```
ackStatusByte = (digitalRead(IoT_SDA)*2)+ digitalRead(IoT_SCL)
```

Fault Recovery Overview

The Pro Micro will send a ping to the Huzzah once per hour. When there are no irrigation faults, a message is sent to the Huzzah each dawn. If, in the previous hour, there was a fault in the WiFi or Maker, the dawn email will be told of the problem. If the Huzzah just went down, the Pro Micro will know as soon as it sends the Huzzah a byte.

This fault coverage is good enough. There is no need to continuously report back the current state of the WiFi or Maker. I just need to know it had a problem recently. When emails can be successfully sent again, the last email that was lost is resent so it is not lost. If the ability to send emails never comes back, I detect this by the loss of my dawn email.

Pro Micro Function Descriptions

```
loadQueue(byte messageByte)
```

All messages generated in the Pro Micro, the master, are put on this queue if it is not full. If the queue is full, which is very unlikely, the newest messages are tossed.

If the message is ESPpingHeartbeat, it is put in the queue and we return.

If the message is not ESPpingHeartbeat, it is put in the queue followed by a messageDisposition. This scheme lets us know of any faults in the email generating message.

When a messageByte is received, it is placed in the next available position. When the queue is unloaded, the message position 1 is returned. This means that the first in is the first out.

Input: messageByte
Output: the array messageQueue[ ] is populated

```
sendMessageToESPandGetACK_Q()
```

Each time this function is called, it can send one message to the ESP. Messages will be sent every IoTIntervalMsUlong until the queue is empty.

The message just sent is saved in messageJustSentByte and is used if it encountered a fault to indicate which faults are timely. When the fault clears, lastStatusOrAlarmEmailSentByte will be put back on the queue followed by a messageDisposition.

State	**Sub-state**	**zone**	
System normal [00]	heartbeat [00]	yesterday’s water usage/10 [gggg] at dawn	
System normal [00]	Just powered up [01]	[0000]	
System normal [00]	All alarms cleared [10]	via button push	[0000]
System normal [00]	All alarms cleared [10]	via leaving the instant flow function	[0001]
System normal [00]	All alarms cleared [10]	automatically at midday	[0101]
System normal [00]	recovery email [10]	ESP came back up	[0010]
System normal [00]	recovery email [10]	WiFi came back up	[0011]
System normal [00]	recovery email [10]	Maker came back up	[0100]
System normal [00]	Alarm Silenced [11]	[0000]	
Minor alarm [01]	Underflow [01]	1-6	[0001 through 0111]
Minor alarm [01]	System failure [10]	Hardware [0000], software [0001]	
Minor alarm [01]	Overflow [11]	1-6	[0001 through 0111]
Major alarm [10]	Can’t stop flow [00]	No zones active	[0000]
Major alarm [10]	Can’t stop flow [00]	1-6	[0001 through 0111]
Maintenance message [11]	Historical flow data reviewed [00]		[0000]
Maintenance message [11]	Local heartbeat	[01]	[0000] Pro Micro pings ESP
Maintenance message [11]	Local heartbeat	[01]	[0001] messageDisposition
message queue empty [11]		[11]	[1110] not sent
message queue full [11]		[11]	[1111] not sent

In all cases where a message is actually sent, the ack will be evaluated:

ackStatusByte will have one of these values

espDownByte

espUp_WiFiDownByte
espUp_WiFiUp_MakerDownByte
espUp_WiFiUp_MakerUp_Byte

```
void detectESP_ACK()
```

SDA and SCL are used to send the ESP a message. To perform an ACK, both are changed to outputs at the ESP end for 50 ms and then changed back to inputs.

SDA	SCL	
0	0	ESP is down (i.e. no response from the ESP)
0	1	ESP is up but WiFi is down. Therefore, Maker can't be reached
1	0	ESP is up, WiFi is up, but Maker is down
1	1	ESP is up, WiFi is up, Maker is up

The result of the ACK is put into ackStatusByte.

When ackStatusByte transitioned from faulted to ATP, emailJustCameBackBool is set true.

previousAckStatusByte is compared to the current ackStatusByte to set emailJustCameBackBool

```
DisplayFaultDetails()
```

The newest ACK failure is displayed. ackStatusByte can change after each ACK but my flags are only cleared manually.

AkF	ESP was not responding
WfF	WiFi was down
MkF	Maker was not responding

These three letter alerts are placed in the upper right corner of the LCD.

```
processACKfaultsAndTranslateIntoArecoveryMessage()
```

messageJustSentByte tells us what triggered this ACK. Ping does not send an email so is only trusted to report ESP down. messageDisposition tells us the about WiFi and Maker faults.

If the ACK says the ESP is down, it is the current state so we act on it imediately.

If the ACK says WiFi down or Maker down, it relates to the previous message that caused an email. Each message that causes an email is followed by a messageDisposition. It returns the status generated by the email message so is the one we process into a recovery message.

If the recovery message fails as detected by its companion messageDisposition, we repeat the cycle which is set by the ping timer using pingIntervalMsUlong.

```
void FaultRecovery()
```

When ACK transitions from faulted to ATP, lastStatusOrAlarmEmailSentByte with its messageDisposition are put back on the queue because it didn’t make it to the user the last time it was sent.

ESP Function Descriptions

`acquiringMessageFromProMicro()`

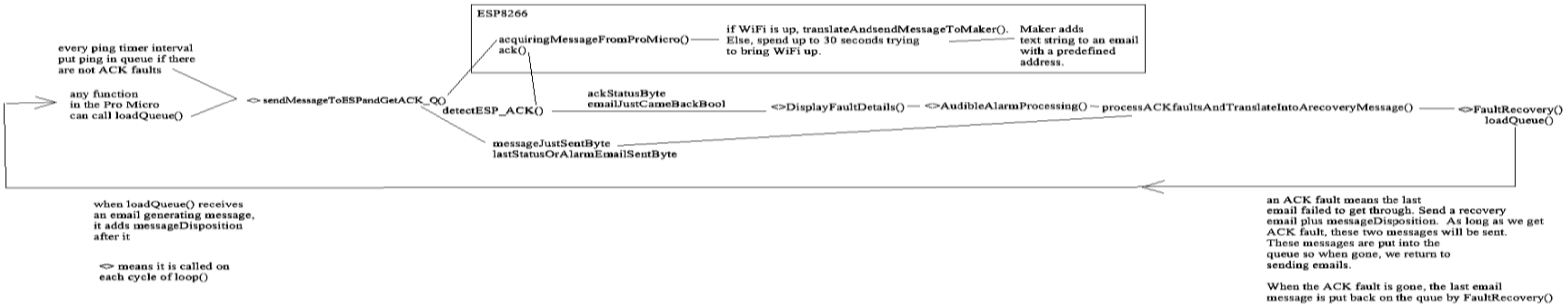
Read in byte from the Pro Micro and translate it to a text string. It only returns when it has a new message.

`translateAndsendMessageToMaker()`

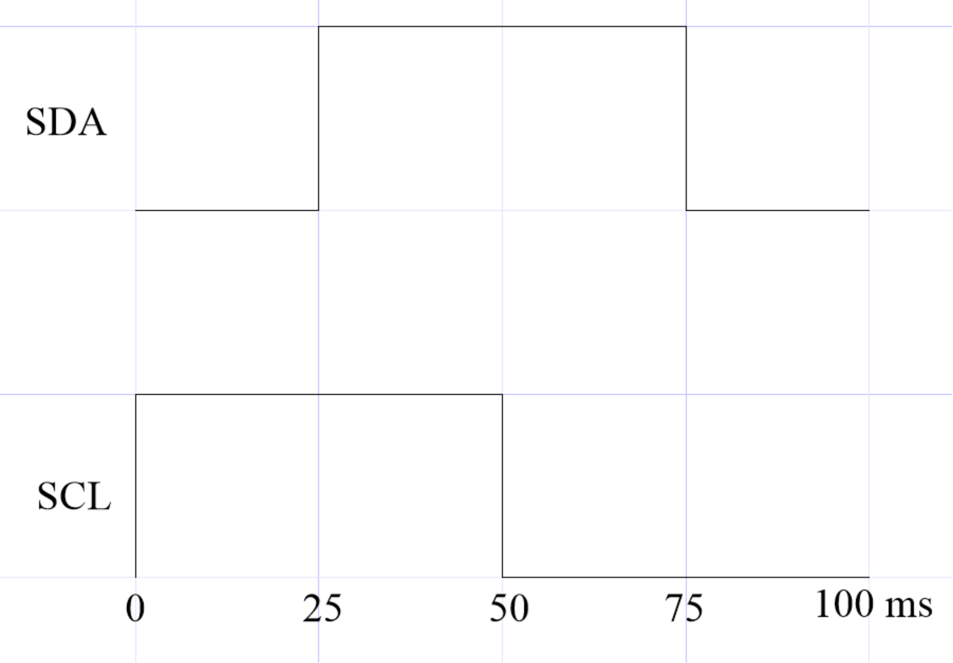
If data from the Pro Micro is a ping or messageDisposition, we just return.

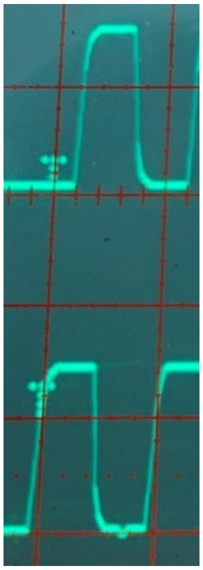
In the process of sending a message to the Maker, we control the fault flag makerFailedBool.

Here is how the functions relate to each other.


ESP8266
acquiringMessageFromProMicro()
ack()
if WiFi is up, translateAndsendMessageToMaker(). Else, spend up to 30 seconds trying to bring WiFi up.
Maker adds text string to an email with a predefined address.
every ping timer interval put ping in queue if there are not ACK faults
any function in the Pro Micro can call loadQueue()
<> sendMessageToESPandGetACK_Q()
detectESP_ACK()
ackStatusByte
emailJustCameBackBool
<>DisplayFaultDetails()
<>AudibleAlarmProcessing()
processACKfaultsAndTranslateIntoArecoveryMessage()
<>FaultRecovery()
loadQueue()
messageJustSentByte
lastStatusOrAlarmEmailSentByte
when loadQueue() receives an email generating message, it adds messageDisposition after it
<> means it is called on each cycle of loop()
an ACK fault means the last email failed to get through. Send a recovery email plus messageDisposition. As long as we get ACK fault, these two messages will be sent. These messages are put into the queue so when gone, we return to sending emails.
When the ACK fault is gone, the last email message is put back on the quue by FaultRecovery()

Details Of the FMC to ECP Interface





The code in the FMC is generating the waveshapes on the left. On the right is the measured signal. In this case, I have changed the clock period from 50 ms to 100 ms. Due to the voltage dividers, the FMC puts out 5V pulses, and the ESP sees 3V.

The ECP looks for the falling edge of SCL and then reads SDA.

New Key and Its Consequences

In early 2024, IFTTT told me I had to pay for the applet used in the FMC. By mistake, I started to pay on an account that did not hold this applet. This meant I needed to change msy key in the ESP software. The change was trivial but also the start of an almost four-month odyssey.

The first challenge was that the library I used to compile the code no longer worked. I found sample code at Adafruit.com and tried it.

It wasn't clear from their website, but the HUZZAH is the same board type as the Adafruit "feather HUZZAH ESP8266".

This sample code caused rolling stack dumps that included hardware resets. Guidance from "`Adafruit_support_mike`", an Adafruit employee on their forum, suggested I find newer libraries, but I still saw the stack dumps. Then Mike suggested that I look at power to the ESP since noise can cause this behavior because glitches in power cause the ESP's memory to become unreliable. I soldered a 100 u*f* electrolytic across power and ground on the device, and the stack dumps vanished. Why wasn't this a problem before I changed the key? I don't know.

Next, I modified my code to include parts of the sample code. I was back to stack dumps. I knew that the OS would trigger stack dumps if I didn't give it given enough realtime. I used the `yield()` function to hand control to the OS as often as possible.

I was able to remove enough of my code to make this work. As I added lines back in, it became clear that the `delay(<time in ms>)` function withheld control from the OS and caused stack dumps. I wrote a new function, `waitMs(<time in ms>)`, that looped on yield() for the specified time. That solved the problem.

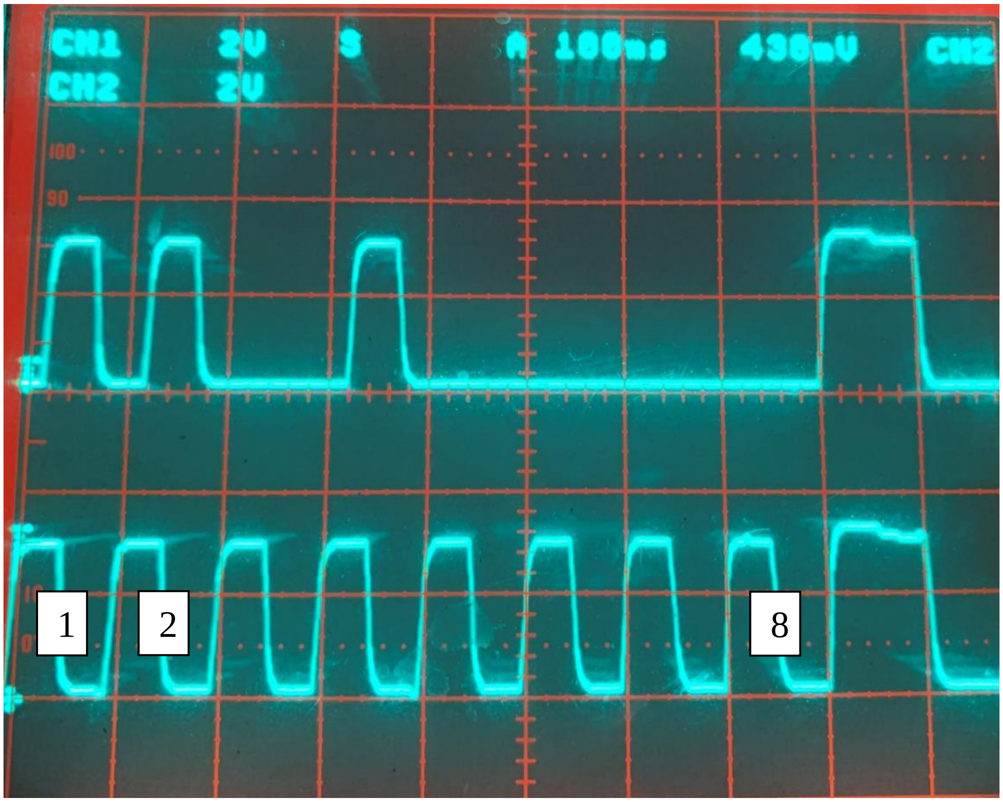
One open question was how much realtime is consumed by calling `yield()`? It shouldn't be surprising that this time varies widely depending on what the OS must accomplish before handing back control. Typically it was less than 1 ms, but once it was 18 ms.

As I sprinkled `yield()` through my code, I became concerned when I reached realtime critical sections. I must process SCL and SDA. SCL had a period of 50 ms, so I needed to detect a pulse that was 25 ms wide and then look at SDA. A variation of 18 ms made this potentially unreliable. Without `yield()` in these areas, the stack dumps came pouring out.

I addressed this problem by lowering the clock rate to a period of 100 ms and permitting a pulse width variation of +/- 15 ms.

Once I resolved one software bug in processing these bits, I was back on the air.

Looking at SCL and SDA at the ESP while a ping occurred:



SCL and SDA are fed from the FMC. The clock period is 100 ms. The ESP clocks in data on the falling edge of the clock, so we are looking at 1101 0000. Then the FMC changes its outputs to inputs while the ECP changes its inputs to outputs. The ECP sends a 11 ACK signal because both WiFi and Maker are operational.