AC Remote Instrumentation, Data Collection, and Monitoring

To monitor power usage, instrumentation may most readily measure current right off a branch circuit at the breaker box.

For example, various machines such as compressors and pumps, are typically each wired with their own individual breaker.

Diagnostics for machine health often start with the measurement of current draw. Monitoring AC current is a natural starting place for instrumentation.

How many lines?

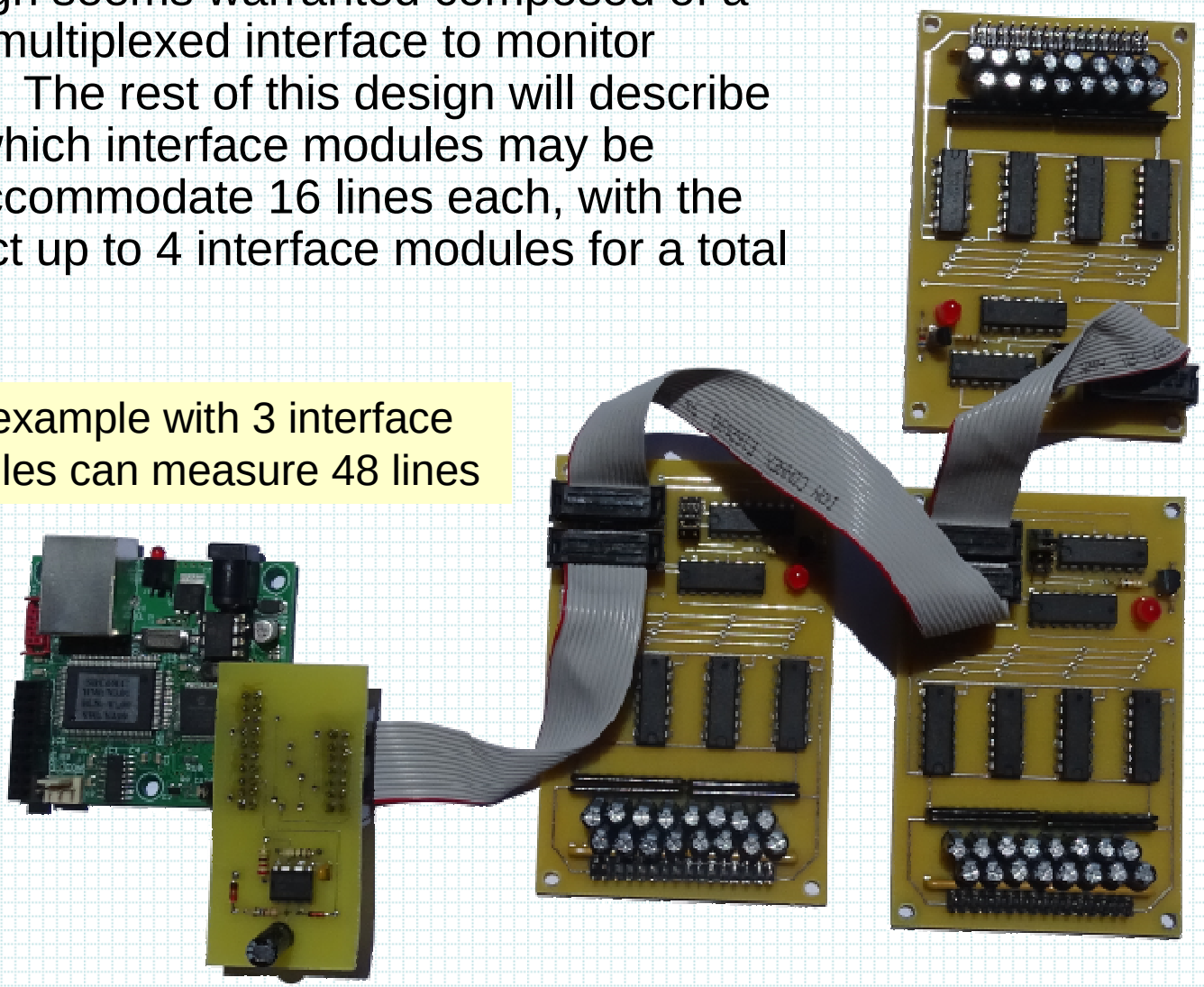
Inexpensive devices can be obtained, such as the Kill-A-Watt, to monitor a single outlet

For multiple lines, or designing a monitor for a breaker (as opposed to an outlet), A/D converters can be found in inexpensive controllers such as the PIC, which comes with multiple ADC inputs

However, even a typical home may have a breaker box with dozens and dozens of breakers. I have more than 1 breaker box, the larger containing 40 breakers

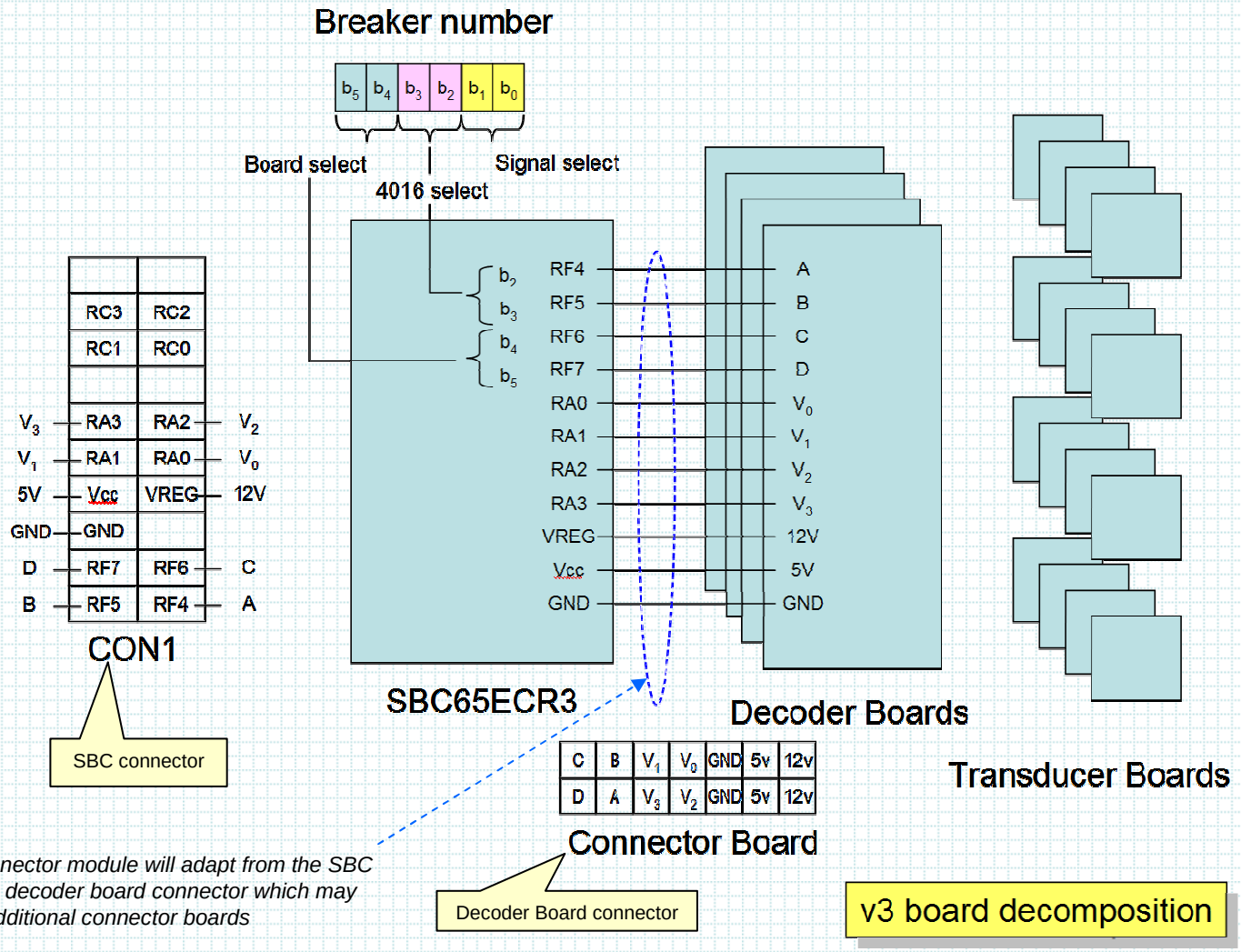
A modular design seems warranted composed of a PIC along with multiplexed interface to monitor additional lines. The rest of this design will describe a controller to which interface modules may be connected to accommodate 16 lines each, with the ability to connect up to 4 interface modules for a total of 64 lines

This example with 3 interface modules can measure 48 lines



The system will breakdown into modular components

- A Commercial Off-The-Shelf (COTS) Single Board Computer (SBC) controller
- Up to 4 decoder boards
  - (2) decoder board address lines
- (4) 4016 analog switches on each decoder board
  - (2) 4016 select address lines
- (4) analog channels from selected 4016 on selected board
- 6 bit transducer addressing by the controller
  - 2 bits board select
  - 2 bits 4016 select
  - 2 bits analog line select
- Up to 64 current transducers
  - Up to 16 per decoder board

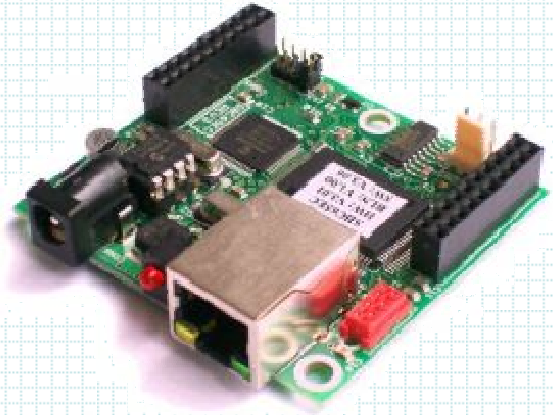


An interface connector module will adapt from the SBC connector to the decoder board connector which may daisy chain to additional connector boards

v3 board decomposition

Controller Module

- *Compact SBC with Ethernet*
- *RS232*
- *I2C*
- *12 Analog Inputs*
- *32 Digital I/Os*
- *free TCP/IP Stack*
- *Bootloader*
- *PIC18F6627*
- *Web Based Configuration*

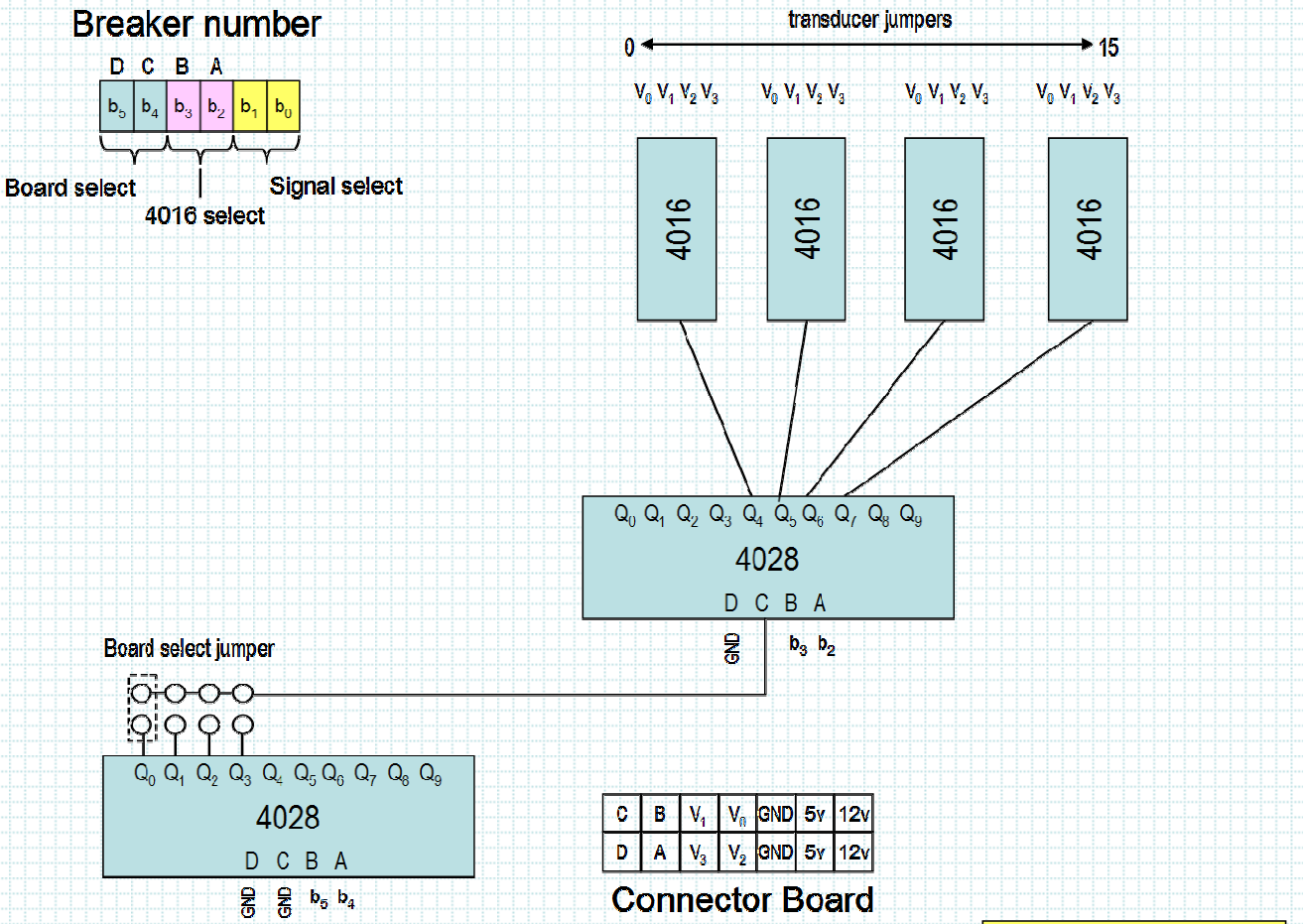


Modtronix SBC65EC

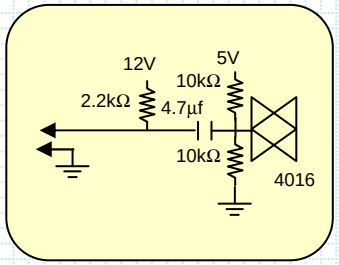
Modtronix is an Australian company from Sydney. They make a variety of SBCs with communication.

Interface Module use a pair of 4028 decoders

- The 2 msb address bits are fed to one 4028 (through a board select jumper) to enable the 2nd 4028
  - *Up to 4 interface modules may be daisy chained together, each with a different board select jumper position*
- The next 2 bits of address are fed to the 2nd 4028 to select which 4016 will be enabled
- Thus 4 analog lines from a selected 4016 will drive the 4 analog lines back to the controller A/D
- Selection of which of the 4 A/D lines is determined by the lsb 2 bits of the address within the controller software



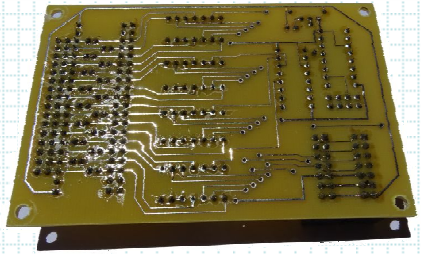
- Each 4016 has 4 analog inputs
- (4) 4016s provides 16 analog inputs per Interface Module
- Each of the 16 analog inputs is provided
  - A pair of header pins to a connect a current transformer module
  - A pull-up resistor to 12v
  - Capacitive coupler
  - A pair of bias resistors to mid-5v



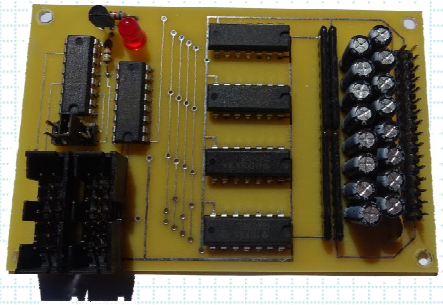
This per-input analog interface is a look-ahead to the following section that defines the current transformer module

Interface Module

- Daisy chains to 14pin connectors on other interface modules
- Daisy chain terminates on Interface Interconnect module
- Up to 4 Interface Modules may be uniquely addressed
- Each Interface Module can select from 16 addressable analog inputs
- Status LED indicates when the Interface Module is selected

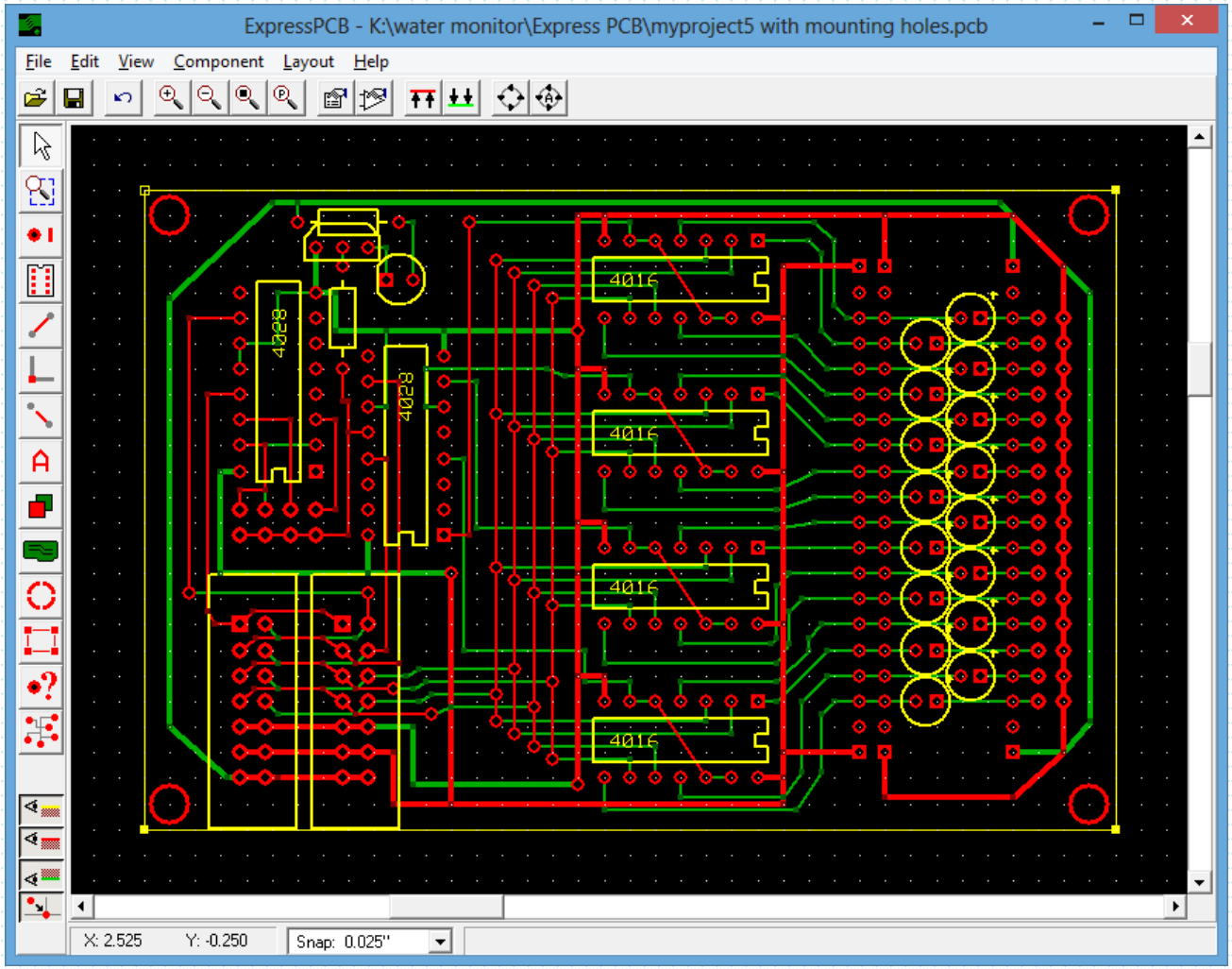


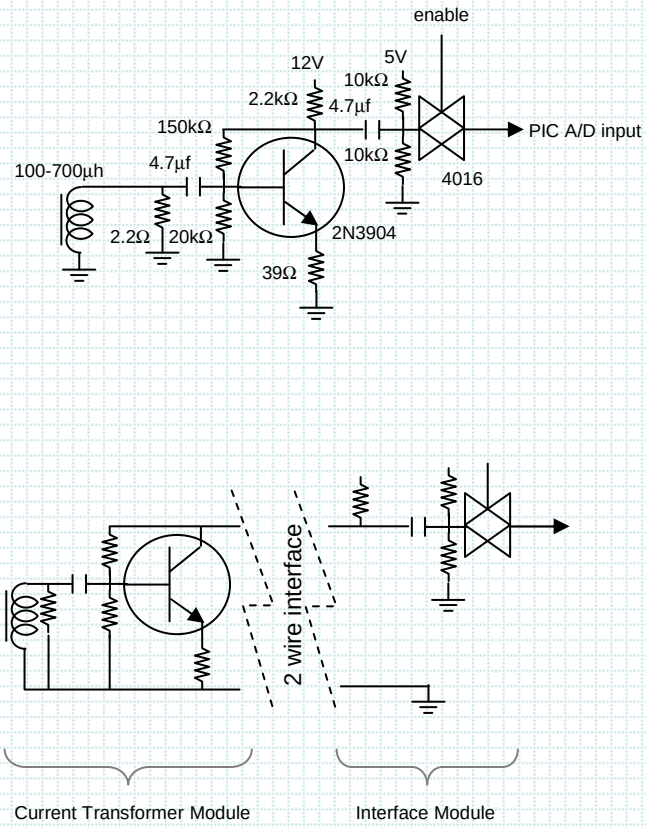
Back side



Component side

- 1 Interface Module layout

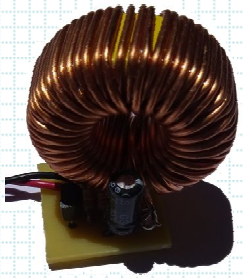




- The current transformer uses a single 12V NPN transistor pickup amplifier
- The signal is coupled to mid-5V to a 4016 analog switch
- Many 4016 outputs can be tied in parallel, where only one is enabled at any moment to drive the PIC A/D input
- The current transformer module is the single transistor with open-collector output, such that only 2 wires are needed to connect to the output pull-up resistor on the interface module

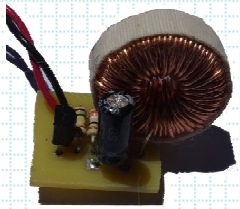
Current Transformer Modules

- Current transformer modules can be assembled using different size toroids
  - *Mainly to accommodate various wire gauge*
- Current transformer modules may be coated with encapsulant to make weather resistant



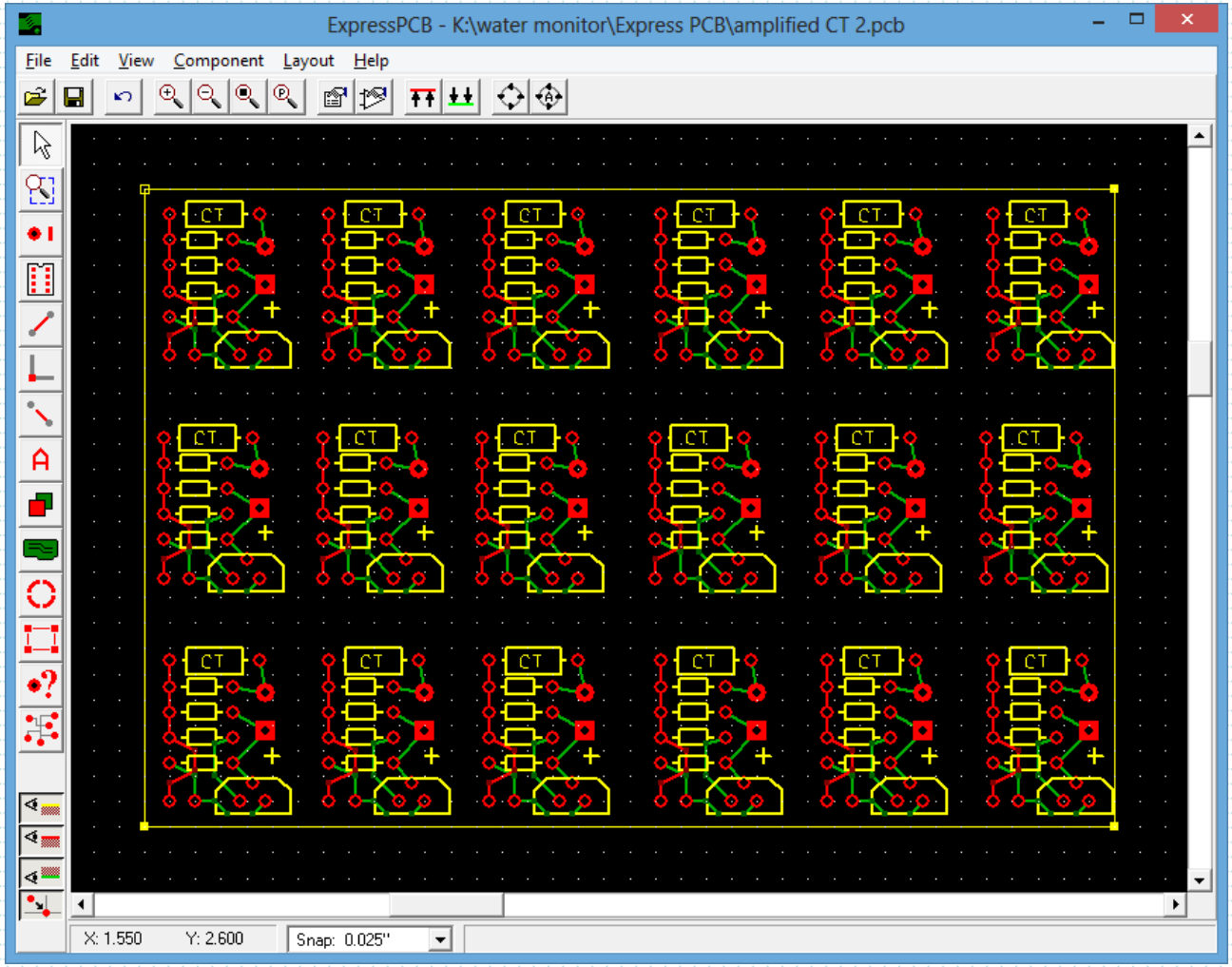
Large toroid

Plasti-dipped

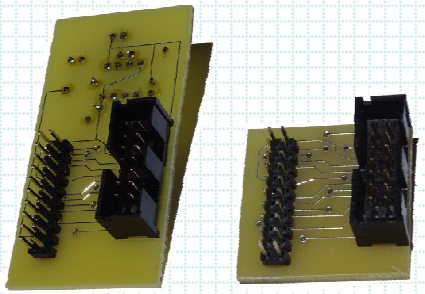


Small toroid

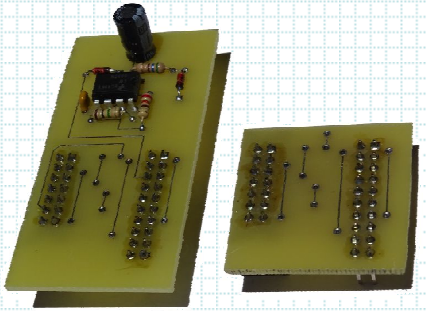
- 18 circuit board layout



- Connects to one of the 20pin connectors on controller
- Daisy chains to 14pin connectors on interface modules
- Versions *with* and *without* watch-dog timer



Connector side



Component side

External WatchDog is Better than Software WatchDog

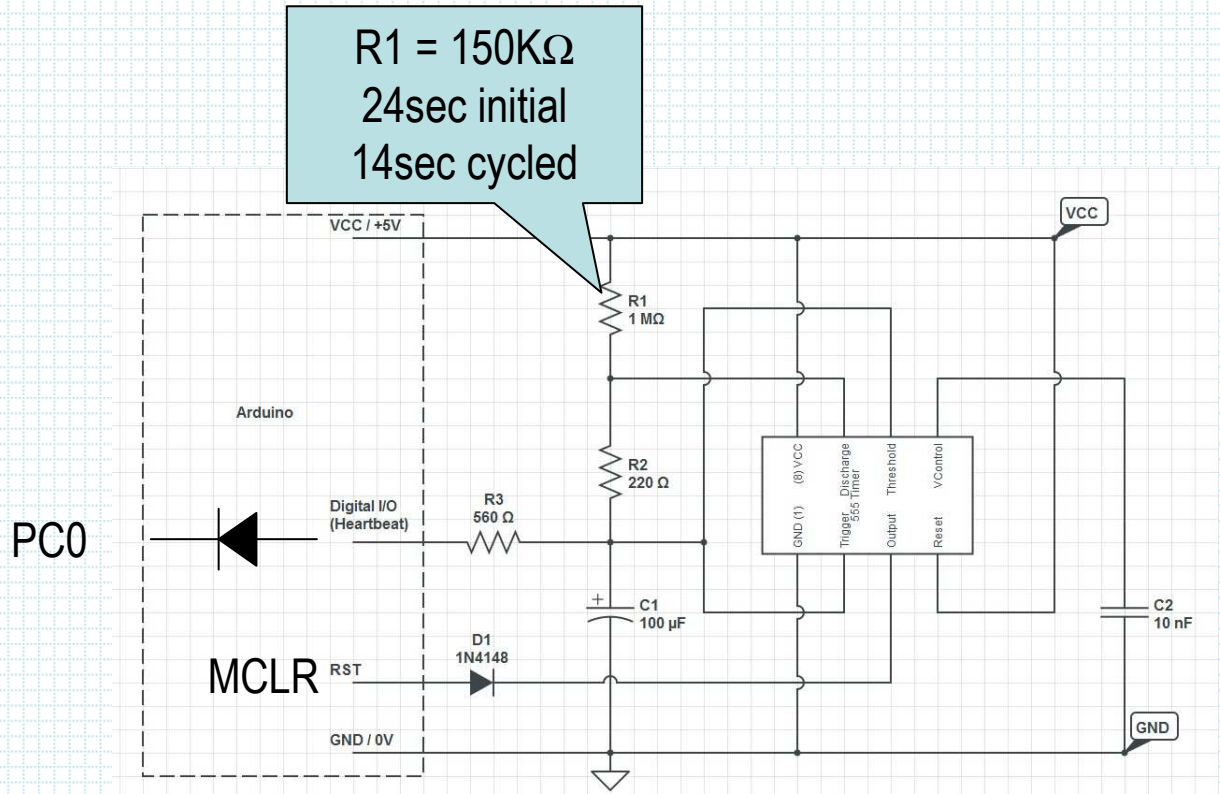
- An external watchdog timer cannot be disabled
- An external watchdog timer cannot be cleared by a single errant instruction
- In short, I found the controller frequently hung using only software watchdog timer, but I've never seen the controller hung with external watchdog timer

Why both versions of Interface Interconnect?

- When running controller code without external watchdog timer actions, the external watchdog timer causes repeated restart
- Therefore one needs a means of disabling the external watchdog timer, or a version of Interface Interconnect without watchdog.

External Watch Dog Timer

Heartbeat Operation
PC0 = output
PC0 = 0
Normal State
PC0 = input
PC0 = 1

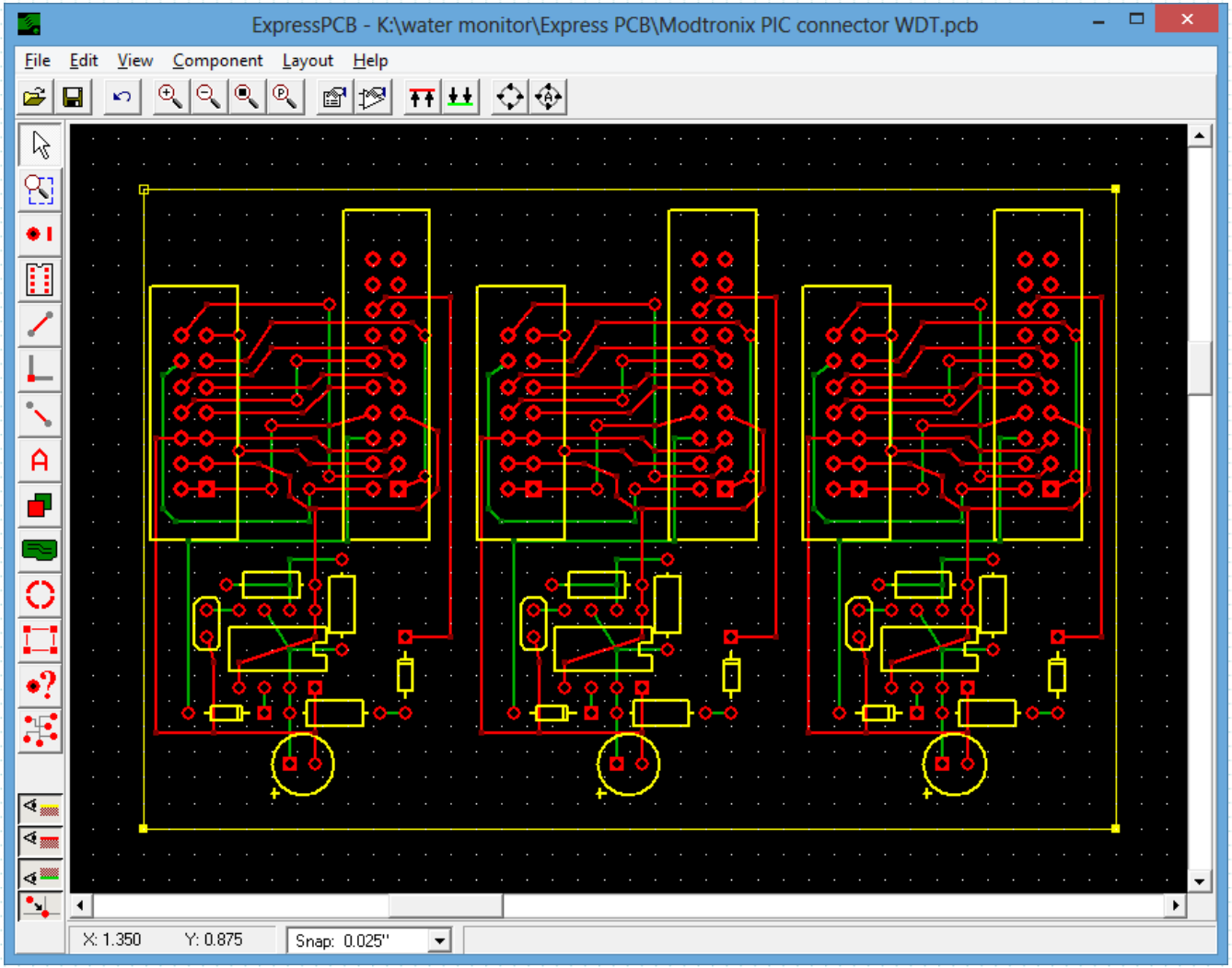


- Low output on PC0 from the controller is used to keep watchdog timer alive
- Watchdog timer pulls MCLR line to controller to force restart

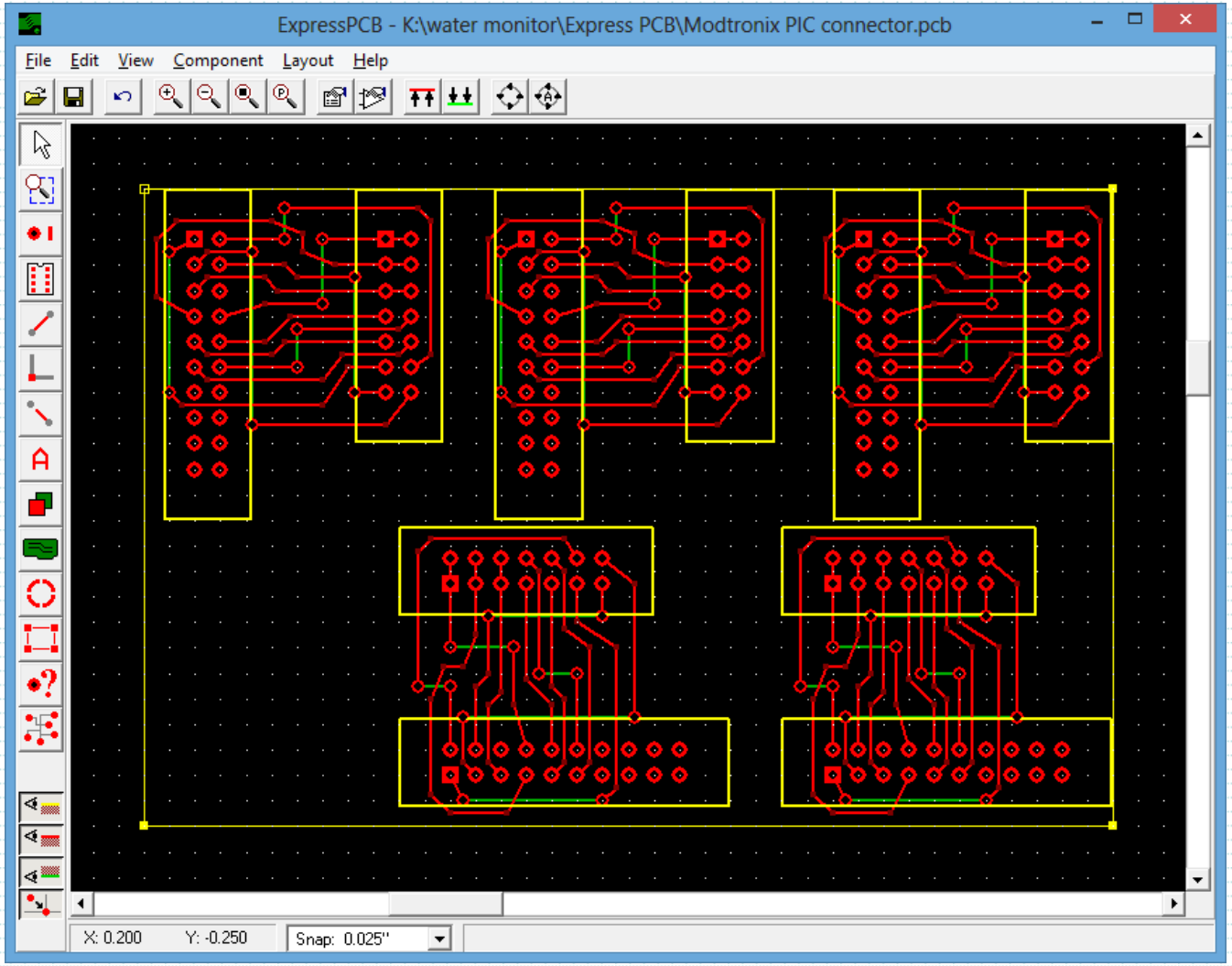
WatchDog Timer keep alive trigger

- The controller is intended to be polled regularly
- Activating the watchdog keep-alive is added to controller software polling response
  - *PC0 momentarily set to output logic low*
- If the controller has not recently responded to a poll, the watchdog timer forces reset
- Therefore the watchdog timer interval should be set to little longer than the intended polling interval
- *For example: if intended polling is <10seconds between polls, watchdog timeout should be >10seconds*

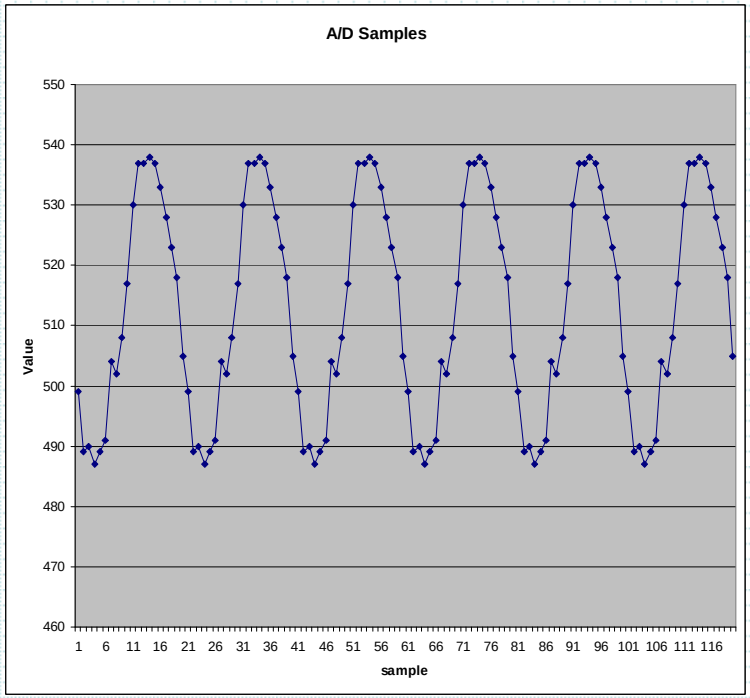
- 3 circuit board layout with watchdog timer



- 5 circuit board layout without watchdog timer



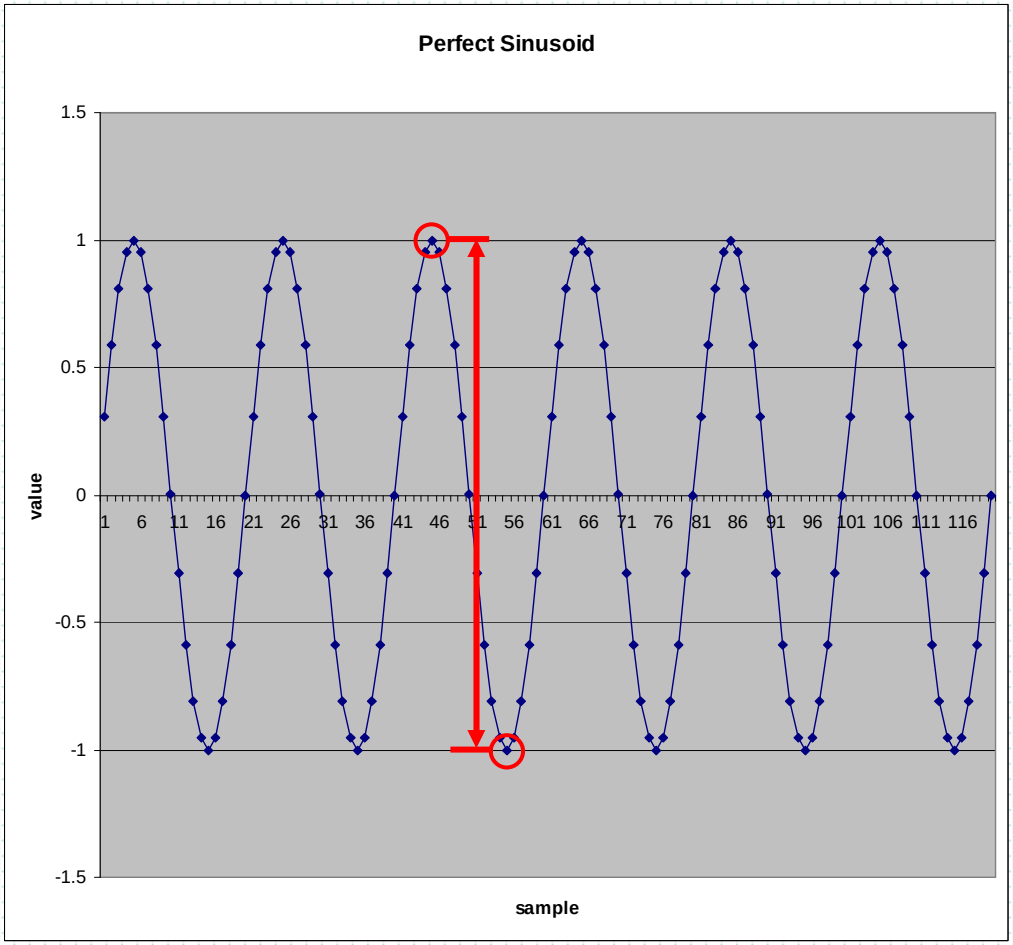
A/D Samples



- We note that with a 10bit A/D, the samples hover around 512
- This is because a 10 bit A/D has a max value of 1023, and the 4016 circuit inputs were biased at half voltage (mid 5V)

- Sampling a selected current transducer at a rate much higher than 60hz, we can see the 60hz sensed signal
  - *This is at a sampling rate of 1200Hz where there are 20 samples of every 60Hz cycle*
- Note that the samples are not uniformly sinusoidal.
  - *This is because the current transducer is not a perfect linear sensor*
  - *This will influence how we attempt to compute amplitude of a sampled waveform*

Amplitude of a pure sinusoid



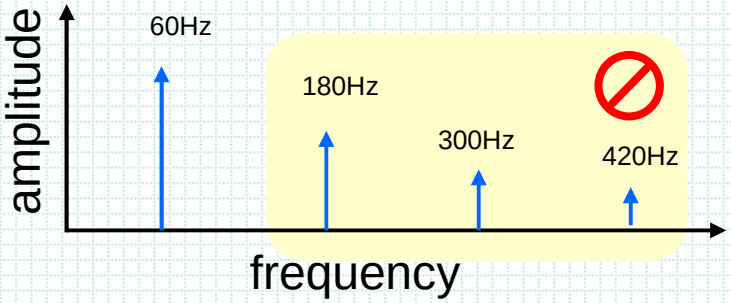
Some small sampled peak-to-peak measurement error exists regarding how high the sampling rate is and how close the highest sample is to the true sinusoidal peak … small as depicted with a 20 times sampling rate

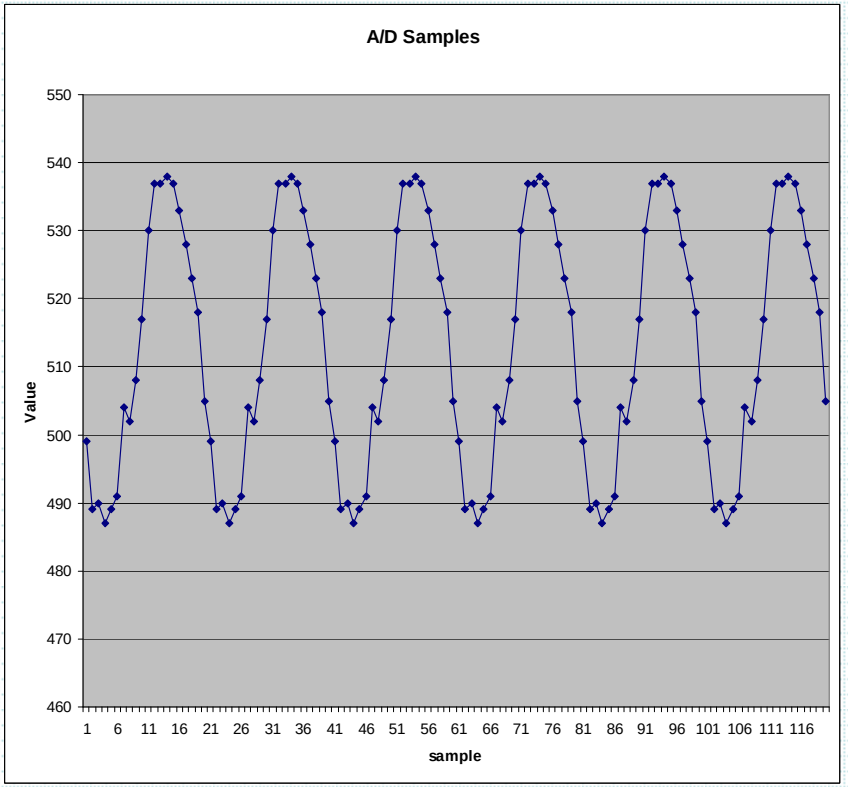
If we were sure we had samples of a perfect sinusoid, we merely need to subtract the minimum sample from the maximum sample observed over any interval of 1/60th of a second

That would be the Peak-to-Peak measurement of the sinusoid amplitude

… but we don't have a perfect sinusoid because of nonlinearities in the current transformer

Our non-perfect sinusoidal samples indicate the presence of odd order overtones produced by the circuit nonlinearities





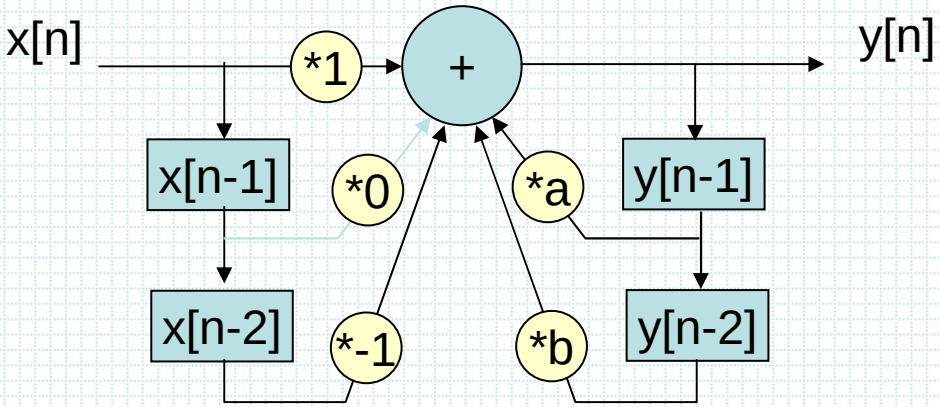
We can simply REMOVE all the overtones in software using digital signal processing, leaving just the sensed 60Hz pure sinusoid

Digital BandPass Filter

- Generated by: http://www-users.cs.york.ac.uk/~fisher/mkfilter
- 2 pole bandpass filter with 1200Hz sampling

Recurrence relation:

y[n] = (-1 * x[n- 2])
+ (0 * x[n- 1])
+ (1 * x[n])
+ (b * y[n- 2])
+ (a * y[n- 1])



input side summation is simplified to x[n] – x[n-2]

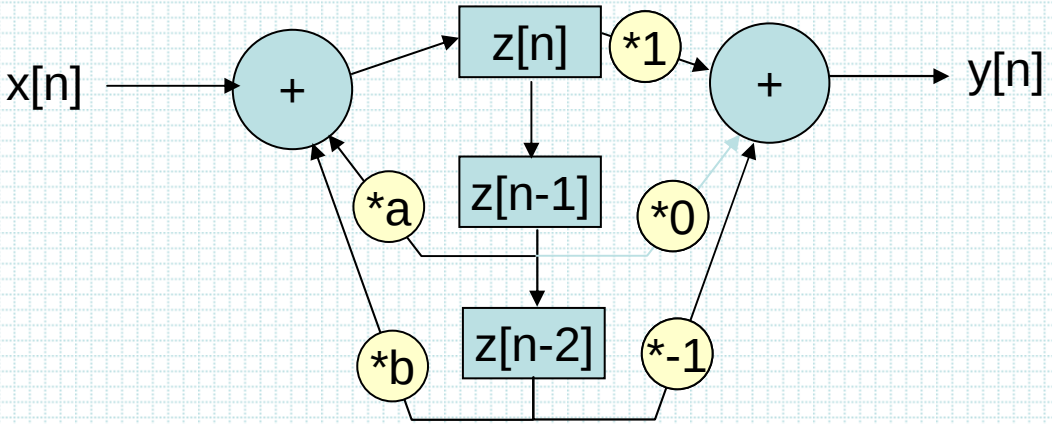
What the terms mean:

y[n]	Current output
y[n-1]	Previous output
y[n-2]	2nd Previous output
x[n]	Current input
x[n-1]	Previous input
x[n-2]	2nd Previous input

Alternate Form

Alternate form: *Needs fewer temporary values*

$$y[n] = (-1 * z[n-2]) + (0 * z[n-1]) + (1 * z[n])$$
$$z[n] = (1 * x[n]) + (b * z[n-2]) + (a * z[n-1])$$

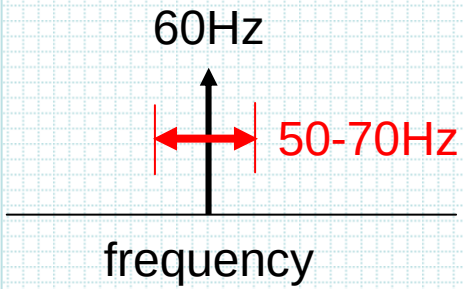


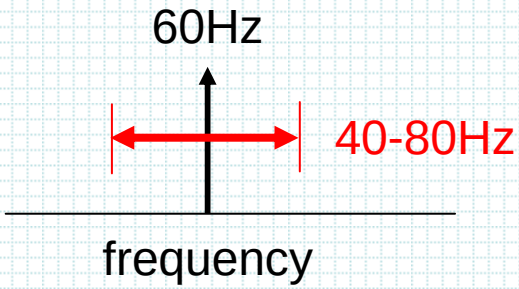
output side summation is simplified to z[n] – z[n-2]

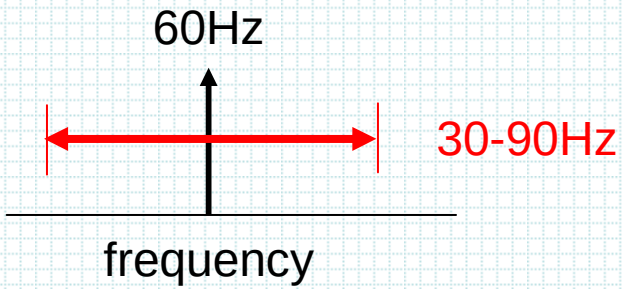
What the terms mean:

y[n]	Current output
x[n]	Current input
z[n]	Current intermediate
z[n-1]	Previous intermediate
z[n-2]	2nd Previous intermediate
***	one less temp value

BandPass Filter Parameters



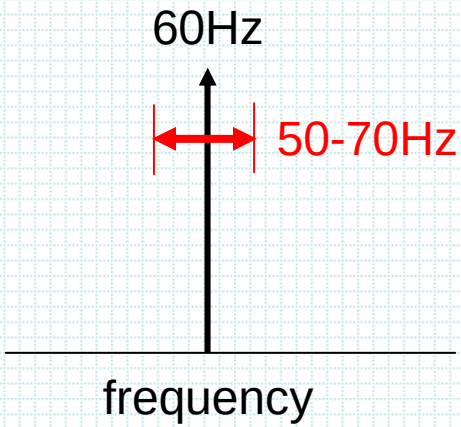


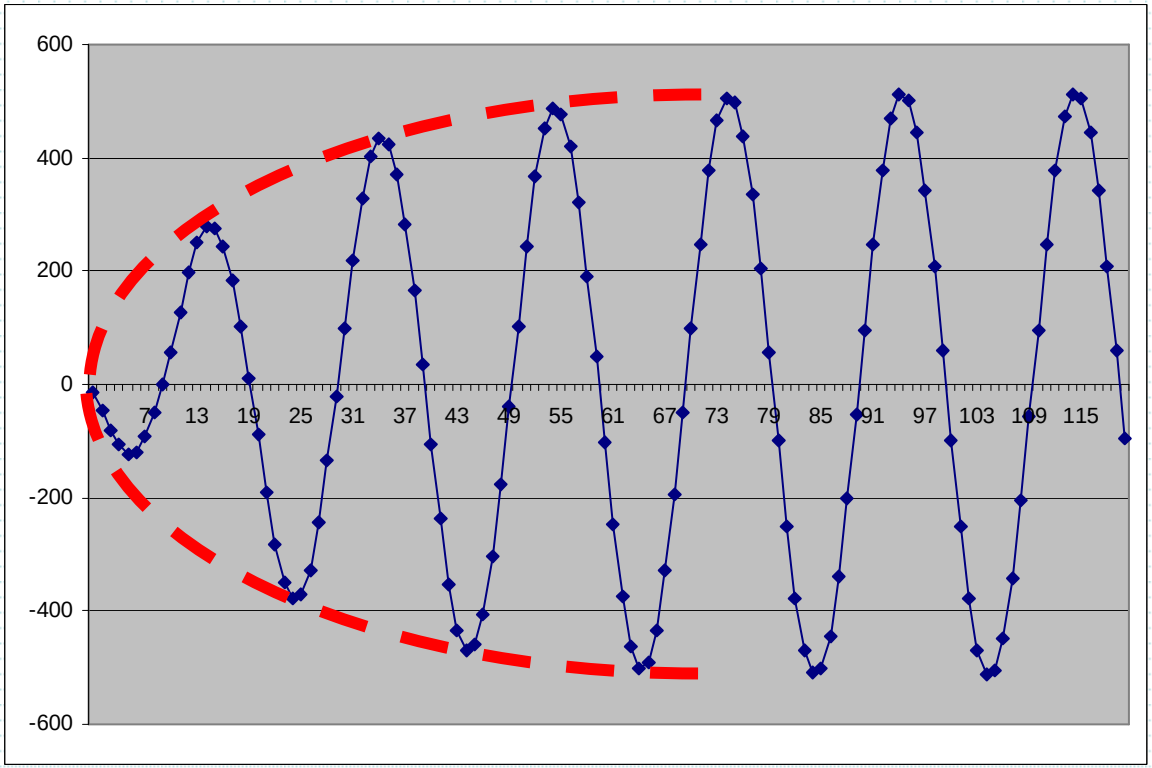


Parameters for 3 different width filter choices

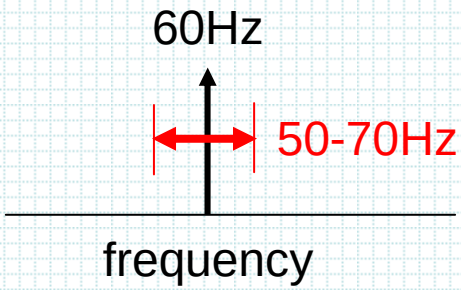
- 50-70Hz BandPass
  - a = -0.9004040443
  - b = 1.8098720166
- 40-80Hz BandPass
  - a = -0.8097840332
  - b = 1.7306877866
- 30-90Hz BandPass
  - a = -0.7265425280
  - b = 1.6625077511

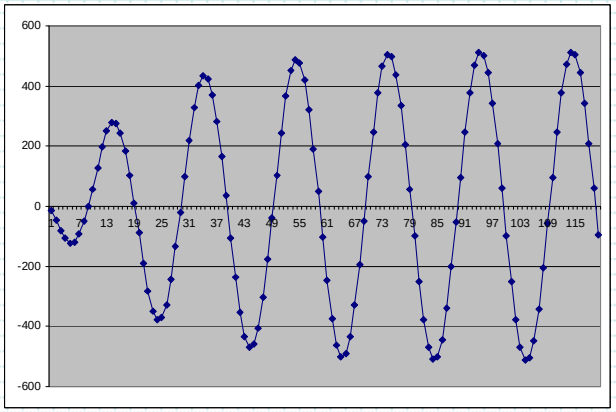
Effect of Digital Filter Width on Ramp-Up



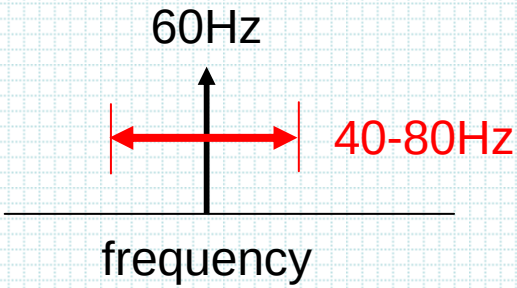


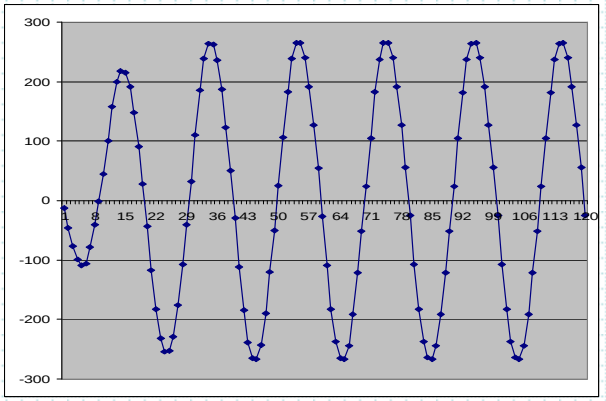
There is a direct link between the width of the digital filter and the ramp-up time for the filter output to reach steady state

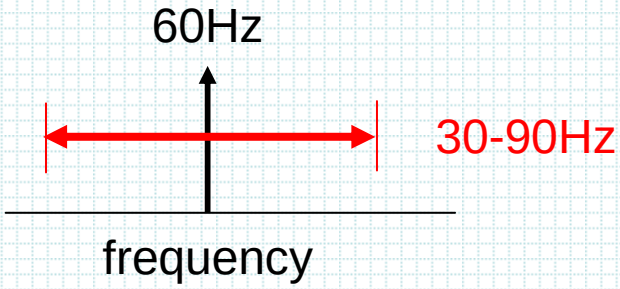


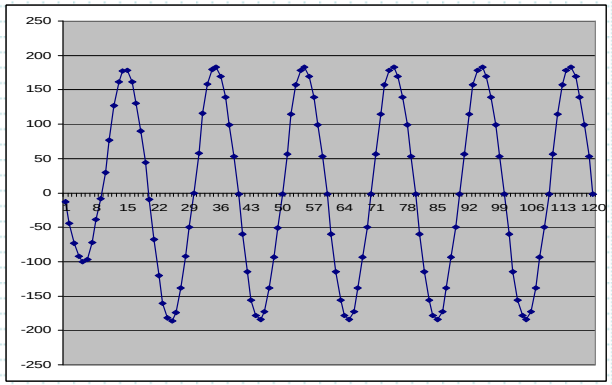


Too slow a ramp-up would require monitoring an input longer before taking a measurement, and slow down the number of measurements per second we could make.



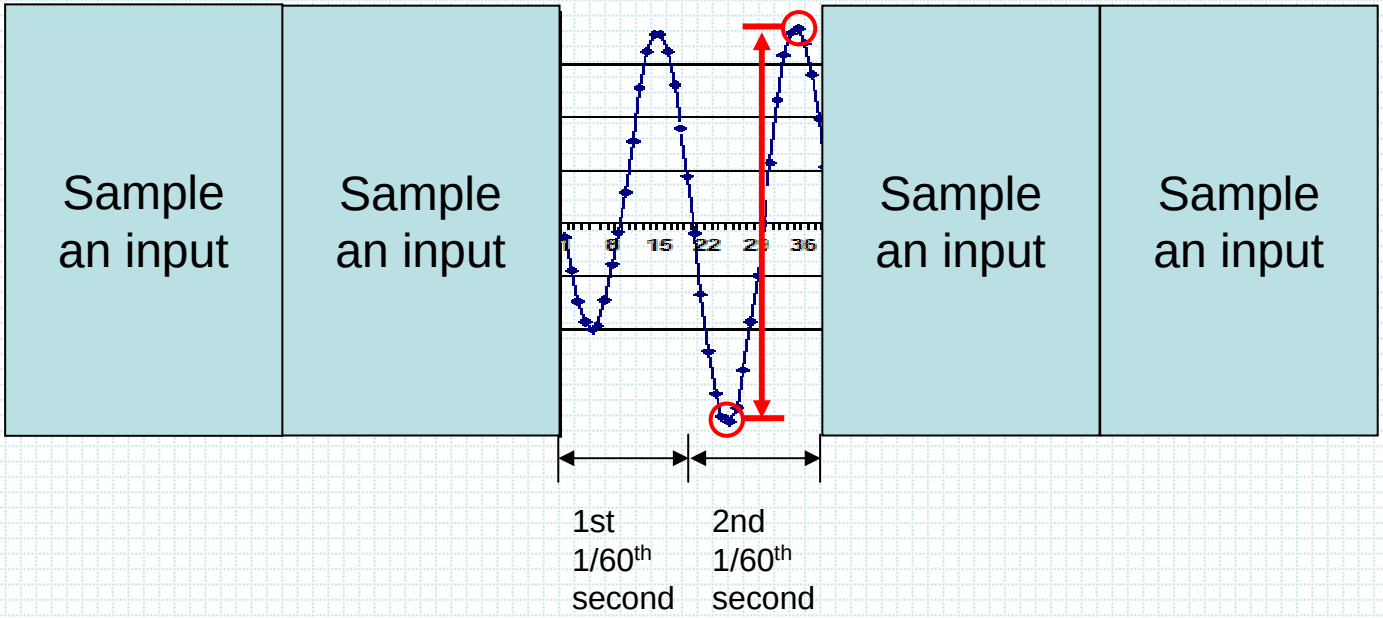






30-90Hz digital filter ramps up within a 1/120th second (half of 1/60th of a second)

- The controller can sample 30 inputs a second, watching each input for 1/60th of a second, and then capturing min & max values in the next 1/60th second
- At that rate it can sample a full compliment of 64 inputs every 2.13 seconds



Fractional Coefficients

Question: How does one multiply by coefficients close to 1 (or less than 1) using integer arithmetic?

a = -0.7265425280

b = 1.6625077511

Answer: Multiply the coefficients by 256 (shift by 1 byte), with the understanding that the least significant byte represents fractional component, with the 2nd byte representing integers

a*256 = 185

b*256 = 425

Adding a byte is just an integer trick to add a byte of fractional precision

*Just remember to divide by 256 (or simply discard least significant byte) to recover **whole integer** answer after multiply with inflated coefficients!*

Software Overview

- Modtronix provides an extensive collection of software with the SBC65EC
  - Web Server
  - Interrupt Service
  - Web Page substitution macros
- Modification to the Modtronix software can create a complete monitoring system

Interrupt Rate and ISR Variables

In websrv65_v310/src/net/tick.c

Add:

```
BYTE            tick12Count;
BYTE            tick20Count;
BYTE            breakerNumber;
int             min_sample;
int             max_sample;
int             Adc_sample;
BYTE            isamp;
long            Input;
long            Output;
long            Intermediate;
long            Intermediate1=8192;
long            Intermediate2=8192;
long            Temp;
BYTE*           p_Temp;
BYTE            diag;
```

Add to TickInit():

```
tick12Count = 0;
Tick20Count = 0;
breakerNumber = 0;
```

In websrv65_v310/src/net/tick.h

Change:

```
#define TICKS_PER_SECOND (100ul)// 10ms
```

To:

```
#define TICKS_PER_SECOND (1200ul) // 1/1200sec = .833msec
```

Add:

```
extern BYTE     tick12Count;
extern BYTE     tick20Count;
extern BYTE     breakerNumber;
extern int      min_sample;
extern int      max_sample;
extern int      Adc_sample;
extern BYTE     isamp;
extern long     Input;
extern long     Output;
extern long     Intermediate;
extern long     Intermediate1;
extern long     Intermediate2;
extern long     Temp;
extern BYTE*    p_Temp;
extern BYTE     diag;
```

Extern definitions added because the variables are defined in tick.c, while declared extern in tick.h which will be included with all other files that will reference them (mostly mxwebsrvr.c, & a couple others)

In websrv65_v310/src/projdefs.c

Change:

```
#define          ADC_CHANNELS        12
```

To:

```
#define          ADC_CHANNELS        40
```

ADC_CHANNELS was intended to provide access to each A/D pin on the PIC. We redefine it for independent measurement for each input on every interface module. Define this for only the number intended to be measured (less than or equal to 16 times the number of connected interface modules).

Set to the number of breakers to be polled
Comment out or delete the test for ADC_CHANNELS<0 || ADC_CHANNELS>12

Change:

```
extern WORD      AdcValues[ADC_CHANNELS];
```

To:

```
extern int       AdcValues[ADC_CHANNELS];
extern long      metric;
```

Adc_Values[ADC_CHANNELS] array was intended to be A/D samples … strictly positive values from 0 to 1023. We redefine it to be the max – min value of the output of the IIR digital filter. Numerics should be “int” rather than “WORD” but since our “difference” should always be positive, probably not really important to change.

In websrv65_v310/src/appcfg.c

Change:

```
WORD   AdcValues[ADC_CHANNELS];
```

To:

```
int    AdcValues[ADC_CHANNELS];
Long   metric = 0;
```

Interrupt Service Routine

In websrv65_v310/src/mxwebsrvr.c

Change:

```
////////////////////////////////////////////////
//High Interrupt ISR
   TickUpdate();
```

To:

```
////////////////////////////////////////////////
//High Interrupt ISR
//------------------------------------------------
//  Preset TMR0 for next interrupt
//  in 1/1200th of a second
//------------------------------------------------
TMR0H = TICK_COUNTER_HIGH;
TMR0L = TICK_COUNTER_LOW;

//------------------------------------------------
//  read the 10bit Analog to Digital Converter
//  output initiated during the prior ISR
//------------------------------------------------
Adc_sample = ((WORD)ADRESH << 8) | (WORD)ADRESL;
Input = Adc_sample;
metric += Input;

//------------------------------------------------
//  Digital IIR Filter "Temp" scaled by 2^8 coef's
//------------------------------------------------
Temp = 185*Intermediate2 - 425*Intermediate1;
```

The ISR routine begins with presetting for the next timer to trigger the next interrupt.

The A/D output is read from ADRESH & ADRESL, transferred to “Input”, and an input summation “metric” is accumulated. The metric is to later judge whether signal is present on an input.

“Temp” is the IIR Filter multiplication of prior intermediate values times filter coefficients inflated by 256

Interrupt Service Routine (continued)


```
//-------------------------------------------------
//   Digital IIR Filter "Temp" divided by 2^8
//   MS Byte must be all FF's or all 00's
//   (max value <2^24 which is 16,777,216)
//   nominal value 8,192*425 (larger coef) = 3,481,600
//-------------------------------------------------
p_Temp = (BYTE*)&Temp;
p_Temp[0] = p_Temp[1];
p_Temp[1] = p_Temp[2];
p_Temp[2] = p_Temp[3];

//-------------------------------------------------
//   Input summation
//-------------------------------------------------
Intermediate = Input - Temp;

//-------------------------------------------------
//   Digital IIR Filter delay
//-------------------------------------------------
Intermediate2 = Intermediate1;
Intermediate1 = Intermediate;

//-------------------------------------------------
//   Digital FIR Filter (IIR Filter output summation)
//-------------------------------------------------
Output = Intermediate-Intermediate2;
```

Because “Temp” is a partial product sum of intermediate values inflated by 256, after the multiply we immediately divide “Temp” by 256. Since this is 8 bits, we do a byte shift rather than performing a divide. “p_Temp” is merely used as a pointer to the 4 byte “Temp” variable.

No attempt is made to bit shift the MS bit of the MS byte of “Temp”. As long as the max value is <16,777,216, the MS byte would be all 00’s or all FF’s anyway.

Completing the digital filter is trivial …

1. Compute a new intermediate
2. Transfer (delay) prior intermediate outputs
3. Compute output

Interrupt Service Routine (continued)


```
//-------------------------------------------------
//  keep the max & min samples during 2nd cycle
//-------------------------------------------------
if (tick20Count >= 20)
{
    if (Output > max_sample)
            max_sample = Output;
    if (Output < min_sample)
            min_sample = Output;
}

//-------------------------------------------------
//  increment a count of 40 samples across
//  a pair of 60hz cycles (20 counts per cycle)
//  20*1/1200 sec = 1/60 sec
//-------------------------------------------------
tick20Count++;

//-------------------------------------------------
//  toggle a digital output each 60hz cycle
//-------------------------------------------------
if (tick20Count == 20) LATB0 ^= 1;
```

tick20Count is counting interrupts of each input. If we're observing the 2nd 60 Hz cycle, observe (and save) max and min values of Output.

Then, perform the actual increment of "tick20Count"

LATB0 is toggled merely to observe the 60Hz interrupt service cycles on an unused PIC output for diagnostic purposes.

Interrupt Service Routine (continued)


```
//-------------------------------------------------
//  do on last sample of a 60hz cycle ...
//-------------------------------------------------
if (tick20Count >= 40)
{
    //-------------------------------------------------
    //      compute & save filtered Vp-p value
    //      lowpass filter Vp-p (average with prior average)
    //      in an int array for current breaker number
    //
    //      expect inputs centered upon ~512
    //      40 samples of 512 make an input metric of 20,480
    //      metric must be at least HALF this or assume
    //      there is no transducer board on this input
    //-------------------------------------------------
    if (metric > 10000)
            AdcValues[breakerNumber] = (AdcValues[breakerNumber] + max_sample - min_sample)/2;
    else
            AdcValues[breakerNumber] = 0;

    //-------------------------------------------------
    //      toggle a digital output at end of 60hz cycle pair
    //-------------------------------------------------
    LATB0 ^= 1;
```

We already used tick20Count to determine if we're in the 2nd 60Hz cycle for min/max determination.

Here, we'll test tick20Count again … when it's reached 40 we're done with this measurement. Everything within this conditional branch is to be performed on a completed sample.

We accumulate A/D samples of an input in a variable named "metric". If we're selecting an interface address that doesn't enable a physical interface module, then "metric" won't have accumulated to an expected value of at least 20,480.

If we don't see at least 10,000, then zero out the value.

Presuming "metric" is above minimum, AVERAGE the new measurement (max_sample – min_sample) with the prior AdcValue. Averaging is simply a low pass filter to smooth out measurement fluctuation.

LATB0 is toggled merely to observe the 60Hz interrupt service cycles on an unused PIC output for diagnostic purposes.

Interrupt Service Routine (continued)


```
    //-------------------------------------------------
    //     reset temp's for next pair of 60hz cycles
    //-------------------------------------------------
    tick20Count = 0;
    max_sample = 0;
    min_sample = 0;
    Intermediate2 = 8192;
    Intermediate1 = 8192;
    metric = 0;

    //-------------------------------------------------
    //     advance breaker number, mod ADC_CHANNELS
    //-------------------------------------------------
    breakerNumber++;
    if (breakerNumber >= ADC_CHANNELS)
         breakerNumber = 0;

    //-------------------------------------------------
    //     output transducer board address,
    //     and select PIC ADC input for next 60hz cycle
    //-------------------------------------------------
    ADCON1 = 0x0B;
    TRISF = 0x00;
    LATF = ((breakerNumber&0x3C)<<2) | ((breakerNumber&0x3C)>>2);
    ADCON0 &= ~0x3C;
    ADCON0 |= (breakerNumber&&0x03)<< 2;
}
```

Restore temp variables to starting values for next input measurement

Step to the next breaker number, modulo ADC_CHANNELS

The least significant 2 bits of breaker address will select which of the 4 analog inputs to A/D sample.

The next 4 bits of breaker address are written to PIC outputs to select interface board, and one of the 4016 chips on that interface board

Interrupt Service Routine (continued)


```
//-------------------------------------------------
//   increment a count of 12 samples in 10msec
//   12*1/1200 sec = 1/100 sec
//-------------------------------------------------
    tick12Count++;

//-------------------------------------------------
//   if 10msec passed, increment legacy 10msec counter
//-------------------------------------------------
 if (tick12Count >= 12) {
    tickCount++;
    tick12Count = 0;
    if (--tickHelper == 0) {
           tickHelper = TICKS_PER_SECOND;
           tickSec++;
    }
 }

//-------------------------------------------------
//   Set ADC to convert newly selected input
//   to be read upon next ISR
//-------------------------------------------------
ADCON0_ADON = 1;
ADCON0_GO = 1;
```

The Modtronix web server has a 10msec “tickCount” counter

This corresponds to 12 interrupts at 1200Hz, so increment a tick12Counter every interrupt

Everytime tick12Counter passes 12, reset it, and count the legacy “tickCount”

Last thing we do in the ISR routine is to trigger the A/D converter to take another measurement

Initialization

In websrv65_v310/src/mxwebsrvr.c

The Modtronix server initialization begins at main()
Followed by an infinite loop

```
Void main(void)
{
    -
    -       various initializations
    -       performed once upon startup
    -       for the PIC and webserver
    -
    while(1)
    {
            -           various actions to
            -           be repeated forever
    }
}
```

Insert the following initialization HERE

```
//-------------------------------------------------
//   Set ADCON1:
//   b5=0, b4=0 ==> Vref+ = VDD, Vref- = VSS
//   b3, b2, b1, b0 = 1011 ==> AN0-3 Analog, AN4-11 Digital
//-------------------------------------------------
ADCON1 = 0x0B;
for (isamp=0; isamp<ADC_CHANNELS; isamp++)
    AdcValues[isamp] = 0;

//-------------------------------------------------
// Make RF7,6,5,4 all outputs -- transducer board address
//-------------------------------------------------
TRISF = 0x00;
```

- Set PIC inputs and outputs for interface modules
- Initialize the AdcValues array

External Watch Dog Timer

In websrv65_v310/src/cmd.c

```
////////////////////////////////////////////////
//ADC variables
else if (tagGroup == VARGROUP_ANALOG)
{
    if (tagVal < 40)
            pGetTagInfo->ref = cmdGetWordVar(ref, pGetTagInfo->val, AdcValues[tagVal]);
    TRISC = 0;          // set tickle line to output to external WDT
    LATC0 = 0;          // set tickle line low to discharge external WDT
    return 1;   //One byte was written
}
```

Everytime we process an ADC variable macro found on a fetched webpage, pull the line to the external watch dog timer

In websrv65_v310/src/mxwebsrvr.c

In main initialization and 1200 Hz ISR routine, return the line to the external watch dog timer to idle state

Put this in TWO places:

- In the main() initialization section before the infinite while(1) loop
- In the interrupt HighISR() loop where it'll be repeated 1200 times a second

```
//-------------------------------------------------
// Normal external watchdog timer state
// Make RC0 = input & RC0 = 1
//-------------------------------------------------
TRISC = 0xff;
LATC0 = 1;
```

The instructions will permit the program to run without WDT reset only as long as it continues responding to webpages containing ADC macros on a periodic basis. If there is any lapse, the WDT will pull the reset line to the PIC.

In websrv65_v310/src/cmd.c

The Modtronix server can substitute various macros embedded within web pages with variables.
It already has support for a macro to be replaced with an element of the AdcValues[array].

```
WORD cmdGetTag(GETTAG_INFO* pGetTagInfo)
{
    -
    -       Detects various predefined macro tags
    -       followed by a 2 digit tag number
    -
    else if (tagGroup == VARGROUP_ANALOG)
    {
            -  perform A/D measurement on input
            -  corresponding to tag number
            -  return that array element
    }
}
```

%n00
macro to be replaced with AdcValues[0]

%n04
macro to be replaced with AdcValues[4]

REPLACE the entire conditional branch body

```
{
    if (tagVal < ADC_CHANNELS)
        pGetTagInfo->ref = cmdGetWordVar(ref, pGetTagInfo->val, AdcValues[tagVal]);
    return 1;   //One byte was written
}
```

- In our system, all the AdcValues[array] are perpetually updated by ISR
- We don't have to perform a measurement when an analog variable group tag is encountered
  … Simply return the array element pointed to by the tag number

- You can put both PIC software and webpages into the modtronix flash
- Macros can be inserted into html pages which will be substituted with current AdcValues when fetched
- Macros can alternatively be inserted into a file in javascript format

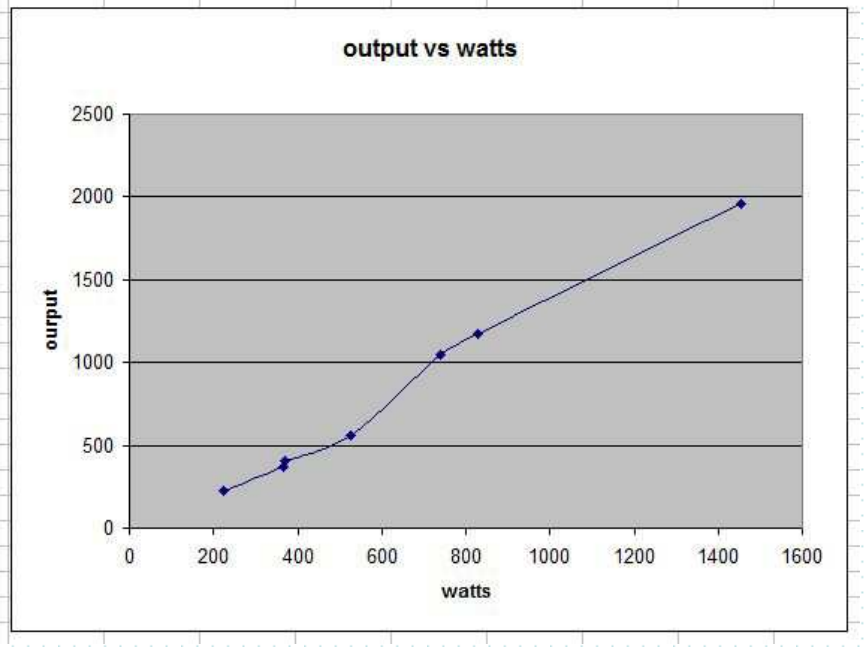
```
AdcValues=["%n00","%n01","%n02","%n03","%n04","%n05","%n06","%n07","%n08","%n09","%n0A","%n0B"];
```

- This is my preference, as the PIC is intended to be polled. Format for html display is not it's intended job
- Or you could put the macros in a comma or tab delimited format or anything of your choosing

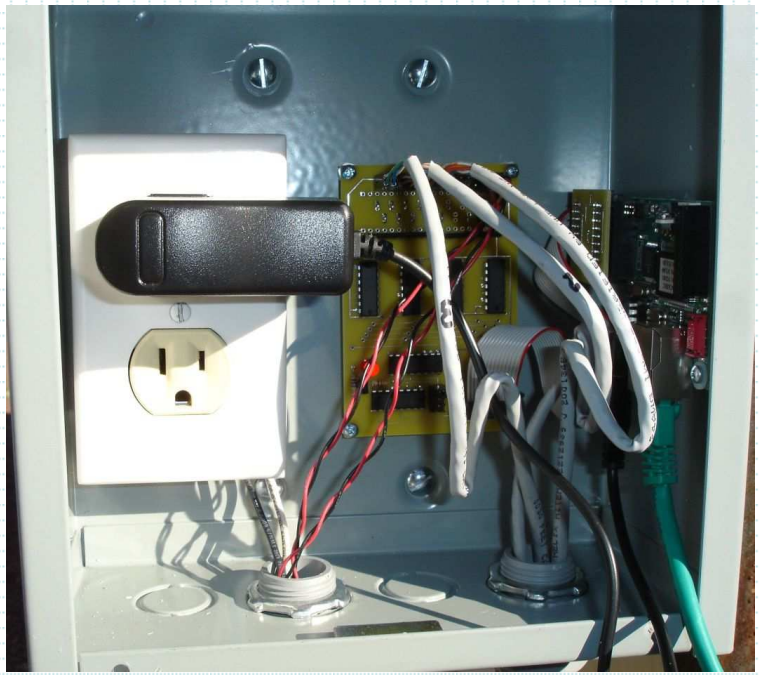
Using a kill-a-watt or your own current or watt meter, record the AdcValue output for various loads. It should hopefully be somewhat linear. You can then apply this as a scaling factor to directly convert to watts or current as meets your need.

If you're going to use a common torroid for all current transformer modules, you could put the scaling to watts or current into the modified modtronix code upon completion of each line sample. Otherwise it may be just left as unscaled AdcValue and scaled as desired when used for display or recording.

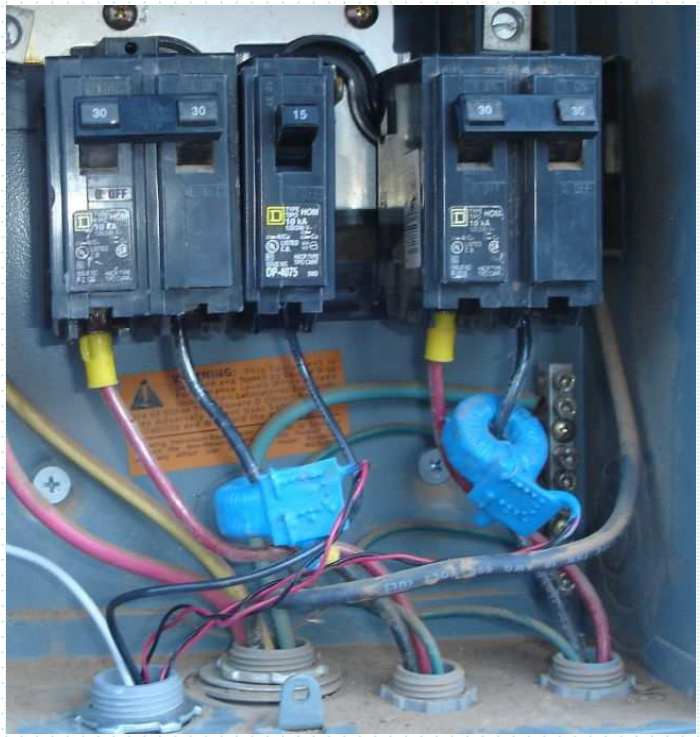
watts	output
224	224
365	367
370	410
525	562
740	1050
830	1176
1453	1962



A unit was deployed to monitor well and reservoir pump circuits



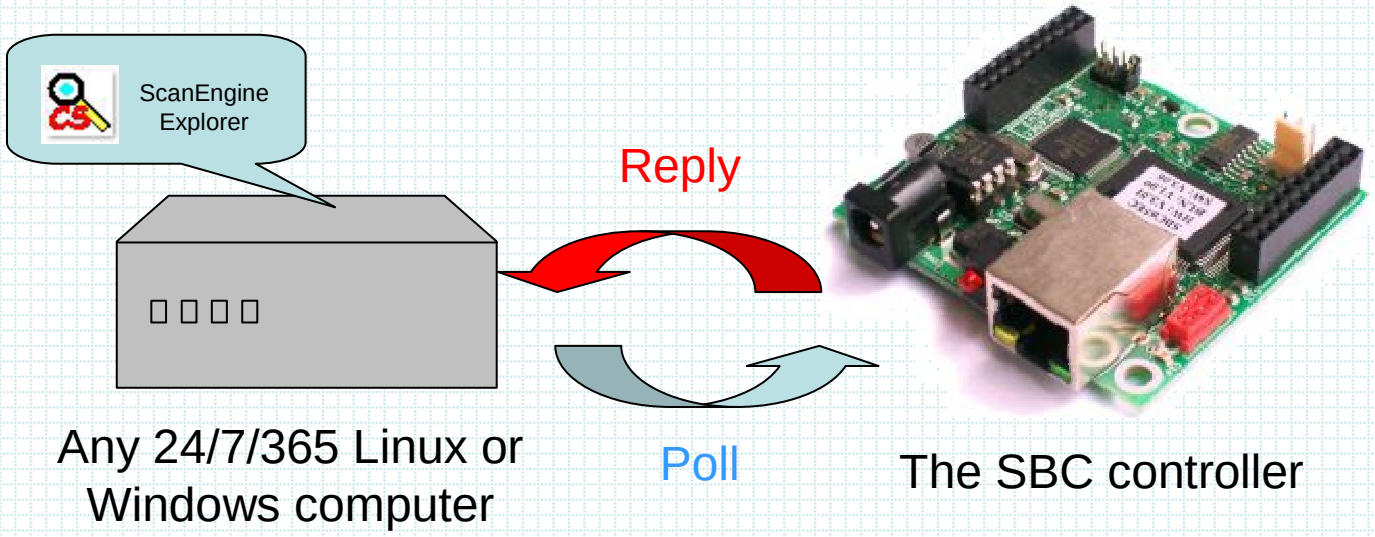
Controller and one interface module deployed next to breaker box
Outlet (on it's own breaker) added for DC adapters for SBC65EC, and an internet radio (no ethernet available at the controller's location)



A pair of current transformer modules off the breaker's for the well pump and reservoir pump. Both are 220v breakers. Just like 110v standard breakers a current transformer module is only needed on one side.

There's no point in monitoring without data collection that can do something

My own network management application software



What follows is an example of what a polling / monitoring application might do

The measurement system is completely independent of whatever the polling / monitoring application does

Functions of the monitoring application are:

- Poll the PIC controller for data
- Provide web status
  - Provide a current status system summary
  - Meaningfully chart the received data
  - Infer and chart water drawn
    - Hourly
    - Daily
    - Monthly
  - Alarm on fault condition
    - Water on failure
    - Pump failures
    - Failures can be logged, emailed, or sent as text to cellphone

Polling script

The modtronix base web server function will respond to http fetch requests. The code set in the modtronix PIC should include a web page with embedded macros to place latest measured values in an http fetched response.

ScanEngine Explorer Script

```
>Assign RepeatTimer = 6
Repeat

>Assign IP = 192.168.2.252 //PIC board
>Assign HTTPPage = jsdata.cgi
>Assign HTTPSitename = "Well Monitor"
>Assign HTTPUsername:Password = admin:pw
Fetch

If HTTPReplyCode == 200
   // process received data
Else
   // no data received
EndIf
```

The ScanEngine Explorer script is instructed to run every 6 seconds.

Every time it runs, it will send an http FETCH to the IP assigned to the PIC board, for the webpage I named “jsdata.cgi” which is what I named a webpage in the PIC

The script tests the reply code, such that it can execute different code when the PIC responds vs when there is no response

On an earlier slide a webpage placed on the PIC was described with Macros that would substitute collected data into any web response:

```
AdcValues=["%n00","%n01","%n02","%n03","%n04","%n05","%n06","%n07","%n08","%n09","%n0A","%n0B"];
```

So, returned http fetch response might look like this:

```
AdcValues=[“1329”,“0”,“332“, etc];
```

ScanEngine Explorer http scraping script

```
page [jsdata.cgi]
port [80]
find [AdcValues=]
name [AdcValues0]
find ["]
text_upto ["]
name [AdcValues1]
find [,"]
text_upto ["]
more of the same
```

Provide a name for the next scraped text
Advance to the next quote,
and scrape text to the next quote

Provide a name for the next scraped text
Advance to the next comma/quote,
and scrape text to the next quote

In ScanEngine Explorer a web page scraping script can be created and associated with any fetched page, and further assign each scraped text (values) with names we define here … in this case, AdcValues0, AdcValues1, etc

Scaling the Measurements

With a web page scraping script defined for the name of our fetched page (jsdata.cgi), our script can reference (*with "Object." suffix*) the scraped data

ScanEngine Explorer Script

```
New Variable well init
New Variable booster init
>Assign RepeatTimer = 6
Repeat

>Assign IP = 192.168.2.252 //PIC board
>Assign HTTPPage = jsdata.cgi
>Assign HTTPSitename = "Well Monitor"
>Assign HTTPUsername:Password = admin:pw
Fetch

If HTTPReplyCode == 200
    // process received data
    // normalize pump circuit current to amps
    Assign well = 0.007166667*Object.AdcValues0
    Assign booster = 0.007166667*Object.AdcValues1
Else
    // no data received
EndIf
```

Define script variables for measured AdcValues for well and booster pump currents

Scale the AdcValues to current in amps
The measured values may be properly scaled to current (A), or apparent power (VA)

Chart the Pump Current

Once the measurements have been scaled to current (A), the currents can be charted

ScanEngine Explorer Script

```
Assign well = 0.007166667*Object.AdcValues0
Assign booster = 0.007166667*Object.AdcValues1
>Assign LogFilename       = "Amps Last Hour“ //chart pump circuit amps
>Assign ChartSeconds      = 10
>Assign ChartType1        = MaxValue
>Assign ChartType2        = MaxValue
>Assign ChartLegend1      = Well
>Assign ChartLegend2      = Booster
>Assign ChartAutoscale1 = 25
>Assign ChartAutoscale2 = 25
Chart well , booster

>Assign LogFilename       = "Amps Last 10 hrs"
>Assign ChartSeconds      = 100
Chart well , booster

>Assign LogFilename       = "Amps Last 2 days"
>Assign ChartSeconds      = 500
Chart well , booster

>Assign LogFilename       = "Amps Last 10 days"
>Assign ChartSeconds      = 2000
Chart well , booster
```

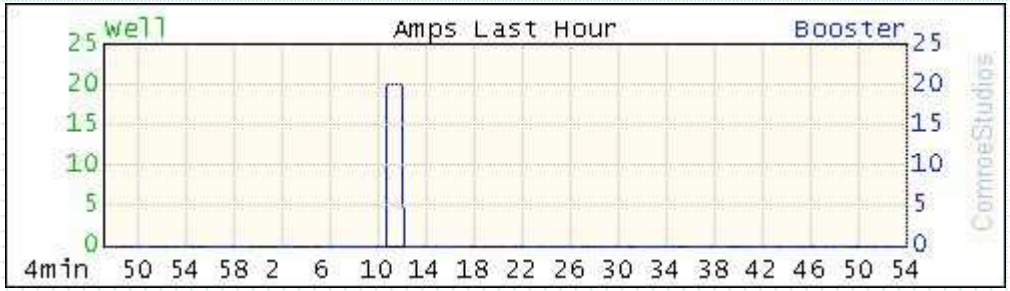
Define a 2 value chart
Autoscale for 25 Amps max value
Define 10 seconds per point

Make similar charts for 100 seconds per point, 500 seconds per point, and 2000 seconds per point

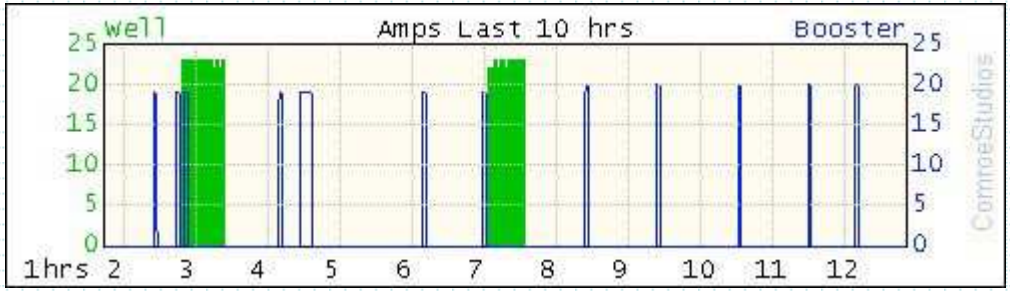
View the produced charts on the next slide →

Charts suitable for status web pages

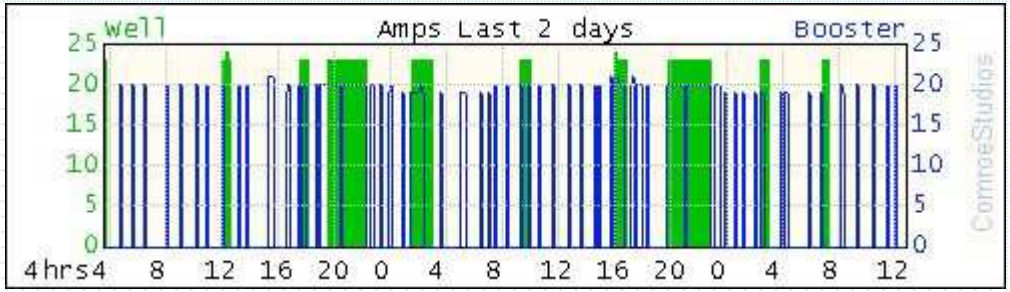
10 sec per point
X 400 points



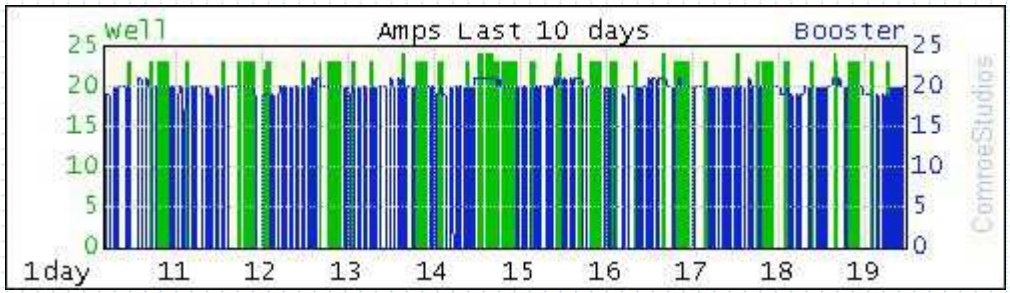
100 sec per point
X 400 points



500 sec per point
X 400 points



2000 sec per point
X 400 points



Counting Booster Pump Activations

- The well pump is ~470ft below ground, and pumps up to replenish the reservoirs.
- The booster pump is triggered on by pressure tank low pressure, and turned off by pressure tank high pressure settings.
- It takes a finite amount of water drawn from the reservoirs to bring the pressure tank back to max … ~72 gallons on our system
- Although it varies (it takes more gallons to replenish the pressure tank when water is being simultaneously drawn), none-the-less counting pressure tank booster pump activations provides a fairly accurate inferred flow rate meter without actually having a water flow rate meter.

ScanEngine Explorer Script

Add booster pump state and count variables

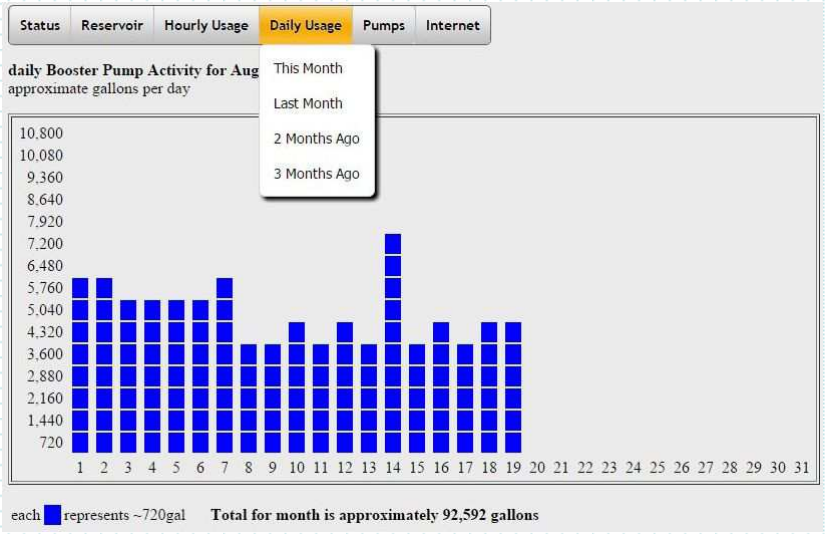
```
New Variable booster_pump_state init off
New Variable booster_count init 0
New Variable hourly_booster_count init 0
New Variable daily_booster_count init 0
```

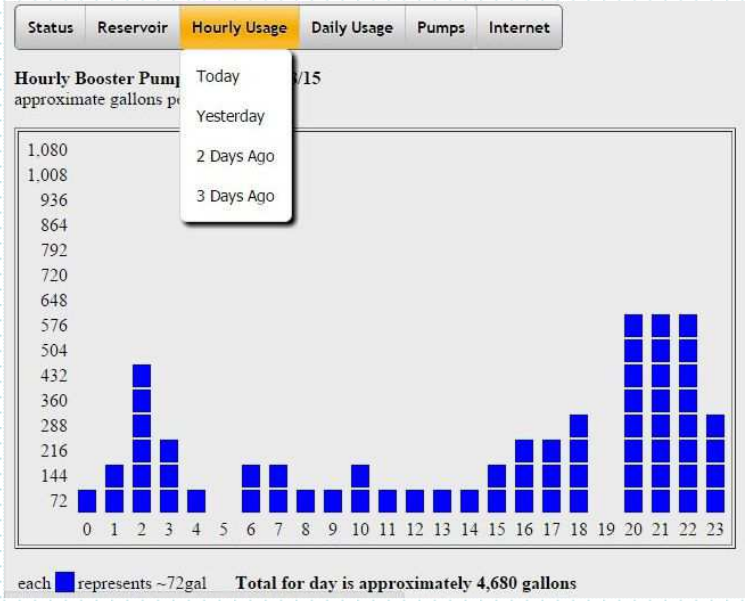
After charting, decide if booster pump is on or off (arbitrarily, test if it's drawing more than 5 amps)

```
If  booster >= 5  //determine booster pump status & count activations
    If booster_pump_state == off
        Assign booster_pump_state = on
        Assign booster_count = booster_count + 1
        Assign hourly_booster_count = hourly_booster_count + 1
        Assign daily_booster_count = daily_booster_count + 1
    EndIf
Else
    If booster_pump_state == on
        Assign booster_pump_state = off
    EndIf
EndIf
```

Increment booster counts every time we determine the booster pump has turned on

Usage can be depicted daily or monthly, displaying hourly usage or daily usage (respectively) for the current and prior 3 days and months





ScanEngine Explorer does not make these graphs. ScanEngine Explorer script creates jsdata files with array data of booster counts that javascript on a webpage can display graphically as illustrated here.

ScanEngine Explorer scripts create these jsdata files, which webpages can visually display with javascript

```
var daily=new Array(31);
var monthstring="August";
var daysinmonth=31;
daily=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
daily[0]=80;
daily[1]=87;
daily[2]=76;
daily[3]=75;
daily[4]=71;
daily[5]=77;
daily[6]=80;
daily[7]=56;
daily[8]=51;
daily[9]=60;
daily[10]=55;
daily[11]=60;
daily[12]=52;
daily[13]=100;
daily[14]=57;
daily[15]=67;
daily[16]=57;
daily[17]=65;
daily[18]=60;
```

jsdata file for monthly usage chart on prior slide

Booster count for prior day appended every midnight

```
var hourly=new Array(24);
var datestring="08/18/15";
hourly=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
hourly[00]="1";
hourly[01]="2";
hourly[02]="6";
hourly[03]="3";
hourly[04]="1";
hourly[05]="0";
hourly[06]="2";
hourly[07]="2";
hourly[08]="1";
hourly[09]="1";
hourly[10]="2";
hourly[11]="1";
hourly[12]="1";
hourly[13]="1";
hourly[14]="1";
hourly[15]="2";
hourly[16]="3";
hourly[17]="3";
hourly[18]="4";
hourly[19]="0";
hourly[20]="8";
hourly[21]="8";
hourly[22]="8";
hourly[23]="4";
```

jsdata file for daily usage chart on prior slide

Booster count for prior hour appended every hour

Maintaining these jsdata files requires script detection of when an hourly, daily, or monthly rollover occurs

ScanEngine Explorer Script

Add variables for today's date, hour, month, and day

```
New Variable today init Left(timestamp,8)
New Variable hour init 99
New Variable month init Left(today,2)
New Variable day init Mid(today,3,2)-1
```

timestamp returns fixed system time/date format

`mm/dd/yy hh:mm:ss`

After counting booster pump activations, decide if an hourly rollover occurred

```
If hour != Mid(timestamp,9,2)
    If hour != 99
        >Assign LogFilename = today_hourly_js
        Textlog "hourly[" + hour + "]=\"" + hourly_booster_count + "\";"
    EndIf
    Assign hourly_booster_count = 0
    Assign hour = Mid(timestamp,9,2)
EndIf
```

Every hourly rollover, append new line to daily jsdata file, clear the hourly count, and refresh the hour variable

Trigger for Daily Rollover

It may be observed from the earlier usage charts, that the script is maintaining CURRENT monthly and daily data, along with the prior 3 days & months

Every daily rollover, need to close today's daily jsdata file and rename older files

ScanEngine Explorer Script

After performing hourly rollover, decide if a daily rollover occurred

```
If today != Left(timestamp,8)
      Delete Textlog three_days_old_hourly_js
      Rename Textlog two_days_old_hourly_js , three_days_old_hourly_js
      Rename Textlog yesterday_hourly_js , two_days_old_hourly_js
      Rename Textlog today_hourly_js , yesterday_hourly_js
      >Assign LogFilename = today_hourly_js
      Textlog "var hourly=new Array(24);"
      Textlog "var datestring=\"" + Left(timestamp,8) + "\";"
      Textlog "hourly=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];"
      Assign today = Left(timestamp,8)

      >Assign LogFilename = this_month_daily_js
      Textlog "daily[" + day + "]=" + daily_booster_count + ";"
      Assign day = Mid(timestamp,3,2)-1
      Assign daily_booster_count = 0
      (continue to monthly rollover detection)
EndIf
```

Delete the oldest day jsdata file

Rename (age) the other jsdata files by one day

Create a brand new (and empty) jsdata file for the new day

Remember the new date

Every daily rollover, append new line to monthly jsdata file, clear the daily count, and refresh the day variable

The daily rollover continues with testing for monthly rollover

ScanEngine Explorer Script

After performing daily rollover, decide if a monthly rollover occurred

```
If today != Left(timestamp,8)
        (continued from daily rollover)
        If month != Left(timestamp,2)
                Delete Textlog three_months_old_daily_js
                Rename Textlog two_months_old_daily_js , three_months_old_daily_js
                Rename Textlog last_month_daily_js , two_months_old_daily_js
                Rename Textlog this_month_daily_js , last_month_daily_js
                Assign month = Left(timestamp,2)
                >Assign LogFilename = this_month_daily_js
                Textlog "var daily=new Array(31);"
                Textlog "var monthstring=\"" + monthstring + "\";“
                Textlog "var daysinmonth=" + daysinmonth + ";“
                Textlog "daily=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];“
        EndIf
EndIf
```

Delete the oldest month jsdata file

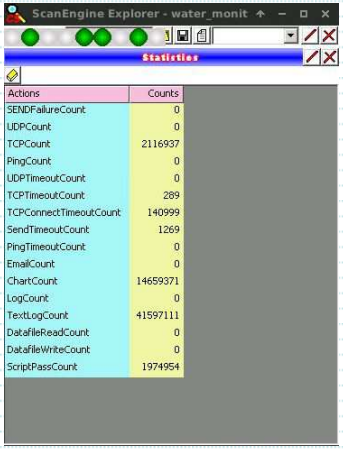
Rename (age) the other jsdata files by one month

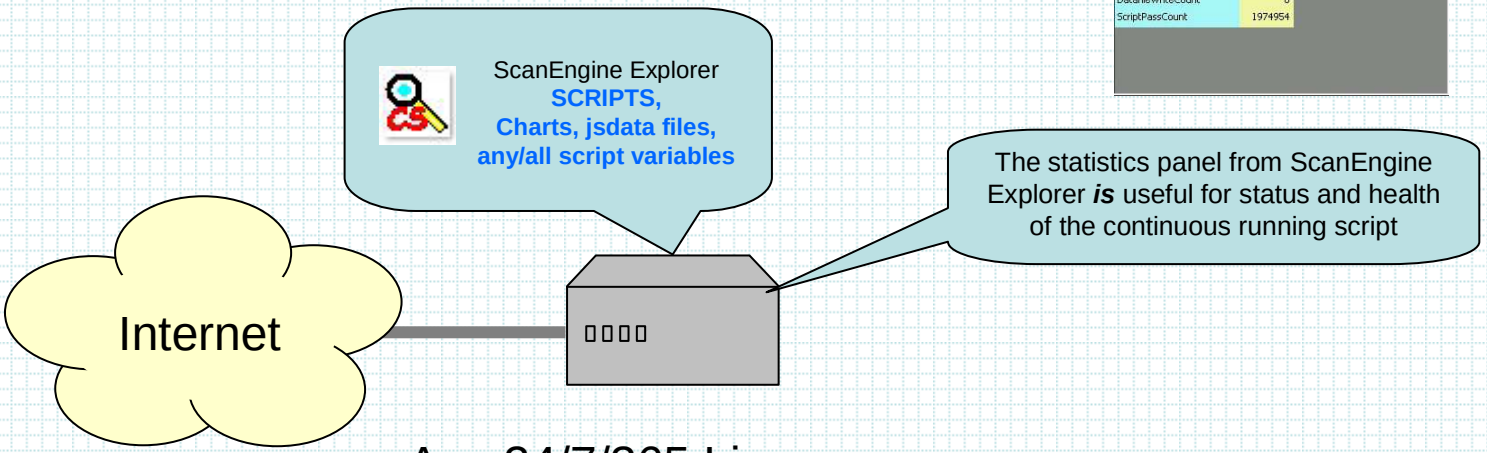
Remember the new month

Create a brand new (and empty) jsdata file for the new month

Programatically calculating *daysinmonth* and *monthstring* is not shown as neither interesting, instructive, nor challenging

While ScanEngine Explorer has a user interface, a web based application interface for the monitor data is most appropriate. ScanEngine Explorer's user interface is specifically for interacting with *network accessible devices* and the *script*, not the arbitrary data a script might produce or maintain.





Any 24/7/365 Linux or Windows computer

add any web server

Install any free web server on the machine running the polling script, along with any web pages for display of charts, scripts, or any/all available script variables. The web based user interface is completely independent from the scripted polling.