C++ Arduino FTP WiFi ESP32

A DIY IoT Power Monitor

deangi

28 Jan 2024 CPOL 0

An ESP32 with a current transformer sensor is used to monitor current flowing in a circuit. On/Off events are recorded and uploaded to an FTP server. Battery operation is supported for remote applications. Initial application is monitoring well pump activity.

This article is currently in progress. This version is not yet publicly viewable

Introduction

Often there are AC loads that are automatically operated based on some conditions. Examples might be space heaters, furnaces, water heaters, pumps, or other machinery.

Frequently these devices do not provide any information regarding their energy use or schedule when they are operational. Often it's desired or necessary to monitor these loads to make sure things are functioning within parameters, or are perhaps failed, broken, stuck on or off, or otherwise operating out of an expected tolerance.

For this application we will use a current transformer (CT) sensor to detect the flow of current in a circuit serving an AC load. This is done by attaching the CT to a "hot" wire serving the load. The CT is electrically isolated from the AC voltage of the load, but can sense current flowing through the "hot" wire. This is because the current flowing in the hot wire generates a magnetic field. The current transformer allows us to monitor the strength of this magnetic field and this will indicate the amount of current flowing through the "hot" wire.

Physically, the CT sensor looks like sort of an awkward "clamp" that has a wire coming from it. This one is given a part number of SCT013 with a notation of "100A/50mA". This means that the CT can sense currents of up to 100Amps. At 100 amps the CT will generate it's own sensor current of 50 ma (50 milli amps). This means that the output current could be somewhere from 0 to 50ma for a "hot" wire current of 0 to 100 Amps. It is an approximately sensor.

The CT has a wire exiting that brings out it's sensor current. By connecting a "ballast resistor" across these wires we can convert that 0 to 50ma to some voltage that could be measured. Let's call this resistor Rb. We can then compute the output voltage of the CT as:

Vout = Rb * Ic (where Ic is the current delivered by the CT - 0 to 50ma)

Since the ESP32 is measuring Vout, we can re-arrange this formula to show how to calculate Ic when we measure Vout:

Ic = Vout/Rb

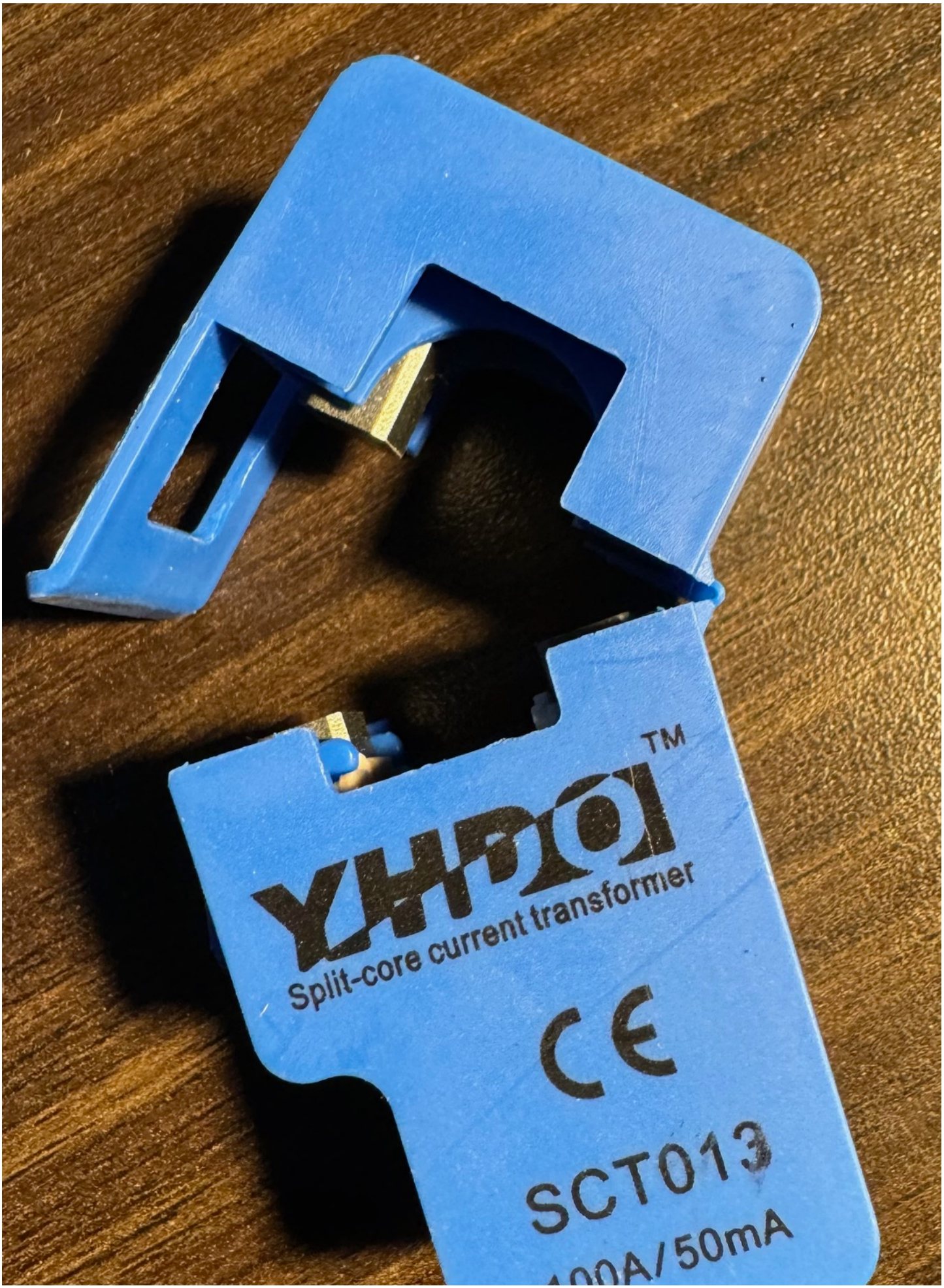
We can further calculate the "hot" wire current by knowing that relationship of (0-10A) to (0 to 50ma). So the "hot" wire current I(hot) is:

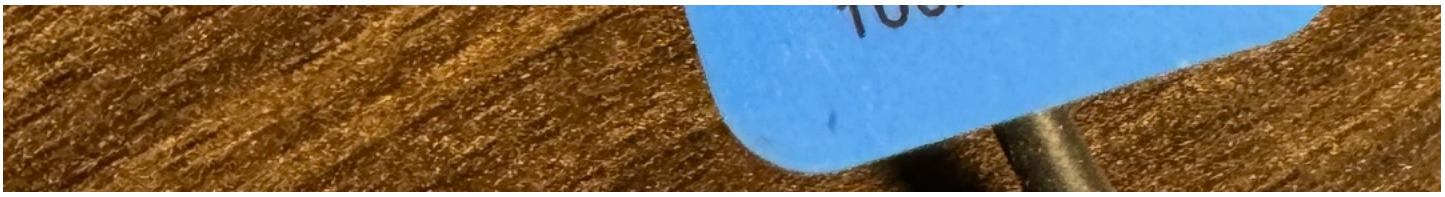
I(hot) = (Ic)*(100a/50ma) = (Ic)*20

If we then substitute the above to calculate Ic from the Vout we measure:

I(hot) = (Vout/Rb)*20

To attach the CT to a "hot" wire we can see that it "splits" as shown in this photo:


YHDC
TM
Split-core current transformer
CE
SCT013
100A/50mA



We can open the CT and clamp it around a "hot" wire so that it looks like this:

You can see there is not an electrical connection between the "hot" wire and the CT. The CT just senses the magnetic field produced by the current flowing in the "hot" wire.

Note: I purchased this CT on Amazon - there were several models available. If you know the current levels you want to measure, there are models that are more sensitive for measuring lower "hot" wire current levels. 100A is a lot of current, most loads in an ordinary home top out at 50A - for example an electric range, electric dryer, perhaps an electric furnace or water heater.

If you only need to measure 10 or 20 A, then it's better to get a more sensitive CT as it will provide more accurate results than using a 100 A sensor to measure 10 or 20A.

Software Architecture

The software will have just a few functions - measure the current using the CT and log that to an event log file in the on-board flash file system. Then periodically it will upload the event log file to a remote FTP server where that data can be stored and analyzed.

Further, we may need to attach this to some remote area where power is not necessarily available, so we want the software to run in a batter powered (read - very low power consumption) mode. This means that the board will spend most of it's time in a low power "sleep" mode where it's doing nothing at all. Periodically it will "wake" up and make a power measurement and possibly log an event. Then it goes back to sleep again.

At much more rare intervals during the wake up period, the software will turn on the WiFi radio and attempt to upload any saved events. The FTP service on a remote computer such as a PC, MAC, or even a Raspberry PI will run the FTP service and receive the data from the IOT device. We do this only occasionally because the WiFi radio consumes a lot of power, so we only activate it occasionally to extend battery life.

So the software will have a measurement part, a logging and communications part. These are designed carefully to allow lowest possible power consumption.

Measurement Software

The measurement software will deal with acquiring data from the CT sensor and converting this into a measurement of the current flowing in the "hot" wire. The ESP32 has a built-in analog to digital converter (ADC) which allows some external voltage to be measured and converted to a digital reading.

In the case of the ESP32 the ADC measures some external voltage and converts that into a number from 0 to 4095 (12 bits) which represents the voltage which is nominally 0 to 3.3V. The ADC has some pre-scaling that allows smaller voltage ranges to be measured as well. For this application we'll be using the 3.3V range. This means that the ADC can tell us the different between two voltages that are as small as 3.3V/4096 or about 805.66 microvolts.

So we can then convert the ADC reading (0 to 4095) into a voltage by:

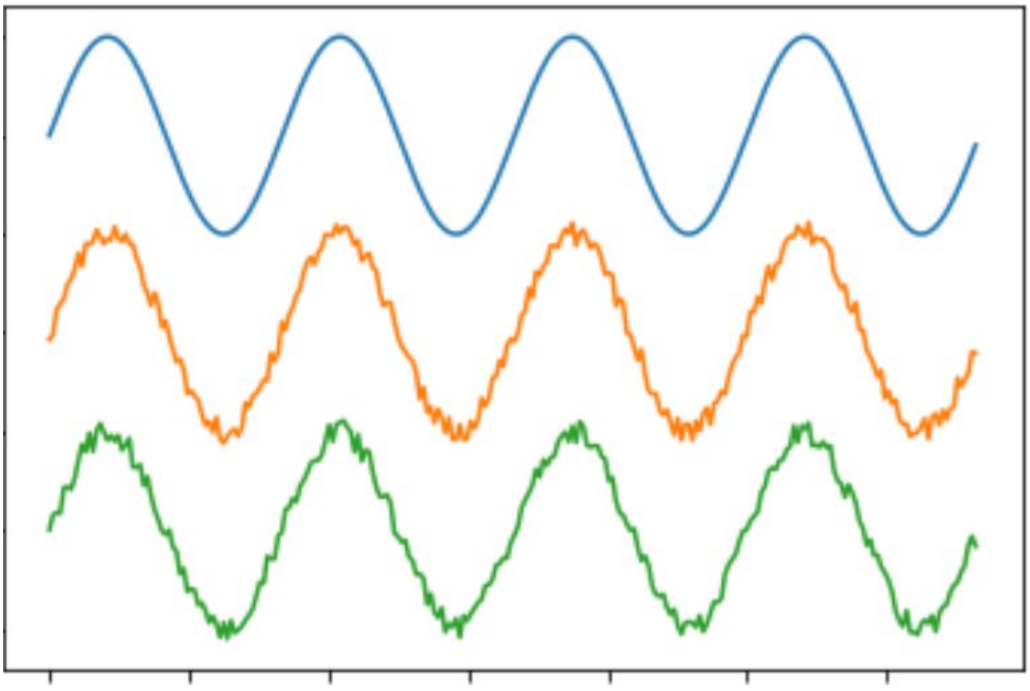
Volts = ADCreading * (3.3/4096)

or

Volts = ADCreading * 0.0008056666

Now over the years I've learned that measuring something in the real world gives you the thing you want to measure **PLUS** some stuff you don't really want or need - namely **plus noise and non-linearities**.

In addition, I learned that the signal being measured was CONSTANTLY changing value because it was an AC waveform. The sample below shows the pure AC waveform (top), the waveform plus noise (middle), and the waveform plus noise plus non-linearities (bottom)



RMS measurement

The fact that we're measuring an AC voltage means that we need to know the "DC equivalent" voltage. For example, your home probably has 100VAC or 120VAC service from the power outlets. If we looked at this waveform we would find it is constantly varying as a sine wave 50 or 60 times a second (depending on where in the world you live). So if the voltage is constantly varying, how can we say it is 120VAC?

The answer is we try to convert the varying voltage into an equivalent that indicates it's average from a power viewpoint. In this case the VAC means the RMS (root-mean-square) value of the sine wave from the outlet in your home.

It turns out that RMS is a well defined mathematical concept (see RMS on wikipedia for details on the mathematics). The application of RMS to AC measurements is also defined on wikipedia here.

For our purposes, in order to measure the current, we'll need to measure the RMS of the AC voltage waveform coming from the CT/ballast resistor circuit. We can then apply the formulas above to get a current value for the "hot" wire.

We also have to contend with the noise component on the signal from the CT that we're measuring. To deal with both the RMS and noise problems we'll measure the AC waveform at precisely timed intervals, and apply the RMS calculation. This computes a "mean" so if we have a lot of captured values, we will be "averaging out" some of the noise.

Using the ESP32 we'll measure the AC waveform at a precise interval so that we get an integer number of samples per cycle - for example 16 of 32 samples per cycle of the AC waveform. We'll also capture multiple cycles to increase the number of data points we can average and help reduce the noise component of the measurement.

So, we need to program the ESP32 to take all these measured samples of the AC waveform, and then process them into an RMS reading in volts. To do this we'll use an ESP32 timer to give interrupts at precise intervals that we can choose by setting up the timer. At each interrupt interval we'll use the ADC to take a sample and store those samples in an area of memory reserved for them.

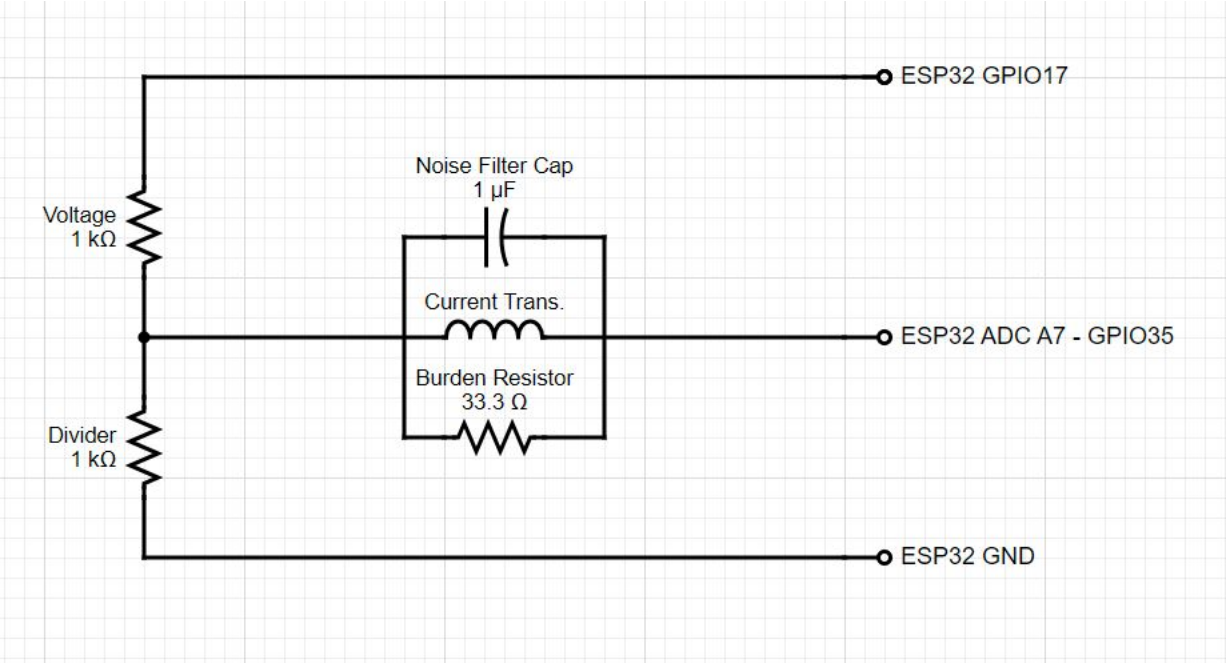
When all the samples have been collected, the ESP32 can compute the RMS value.

One additional complication that happens here is the the CT generates an AC waveform which goes to positive and negative voltage values. Unfortunately the ESP32 can only measure voltages from 0 to 3.3V so it won't be able to measure the negative going parts of the waveform. In fact if we present these negative voltages to the ADC input pin, we may end up damaging the ESP32 hardware - apparently it's rather sensitive to negative voltages and it could damage some of the internal circuitry.

To overcome that difficulty we need to add some value to the voltage before sending it to the ESP32 so that the ESP32 ADC pin never sees a voltage less than 0.

Below is the CT interface circuit that is used.

- On the left side we have a voltage divider that takes 3.3V from GPIO 17 and divides it in half to 1.66 V in the center. We use a GPIO to drive this so that when we go into deep sleep we can set GPIO17 to 0V and not draw any current through the voltage divider during sleep cycles.
- In the center there are three components connected in parallel - the CT sensor, a burden resistor of 33.3 ohms, and a filter capacitor. The filter capacitor helps to remove or filter out some of the noise component.
- On the left we have three connections to the ESP32 board, one is GND (ground), one is GPI35 (center) which is set up to use as an ADC input pin (ADC 7 from the ESP32's viewpoint), and one is set up as a voltage source (top) - GPIO17 which we will set to an output mode and set the value to '1' so it will deliver 3.3V to the voltage divider network.
- At the ADC we will see an AC waveform (with noise and non-linearities of course) which is centered at 1.66V and goes up and down from that, never going over 3.3V and never going below 0 V. This allows the ESP32 ADC to digitize the voltage.



Code To Measure the RMS value of the signal from the CT

First we need to initialize some hardware. We want to digitize the AC waveform coming into ADC line A7. The sampling points need to be evenly spaced, and spaced such that an integral number of samples are taken per cycle of the AC signal. For this application we are sampling 16 times per cycle (we call this the "oversample ratio" or OSR=16). Since the waveform is 60 Hz (in some countries it might be 50 Hz) - this means there are 60 complete cycles per second. So each cycle takes 1/60th of a second or 0.016667 seconds per cycle (16.667 milliseconds). Since we want to sample 16 times within each cycle, we need to take an ADC sample at 16.667/16 = 0.001041667 seconds or 1041.667 microseconds.

In order to do this accurately, we need to use some hardware, we can't really do it from software alone. The technique used will be to set up a timer that generates a periodic interrupt every 1041.667 microseconds. Inside the ISR we will take a sample with the ADC and store it away in a buffer of such samples. This insures that the samples are taken at just the right spacing.

Here is the code to set up the timer interrupt. When the interrupt happens as the ESP32 is running along, it will immediately save everything it's doing and call a special function called an interrupt service routine (ISR). This ISR will do the work of getting a sample and storing it away in the sample buffer.

First we allocate a timer - the ESP32 has three (0, 1, and 2) and we're staying away from timer 0 because it's already used by the base ESP32 software. So we'll use timer 1.

First we have to set up timer 1 to generate interrupts at the right time - there's some flexible (complex) hardware involving counters, clock rates, and pre-scalers that have to be set up - lot's of information on this is available if you search for ESP32 timers on the web.

Secondly we're setting up the timer to call our ISR - which is called "onTimer" below - this means that when the timer interrupt happens, the function onTimer() will be called. You can't pass any arguments to an ISR, so it's just the name of the function to call. Of course in an ISR we always want to do the absolute minimum amount of work, and no diagnostic print outs, etc.

Finally we compute a "count" which will represent the amount of time between interrupts - and is a function of clock rates, pre-scaling of the clock rate, and the count of clocks which represents 1041.67 microseconds.

C++ Shrink ▲

```
#define CLOCKRATE 80000000 /* Hz */
#define TIMERDIVIDER 4

//--------------------------------------------------------------------
void setupMeasurement()
{
  // initialize ADC - no setup initialization is required - we're using
  //    default settings

  // zero out the buffer
  //for (int i = 0; i < NSAMPLES; i++) samples[i] = 0;

  // initialize timer for interrupt every 1041.6666 is as close as we can get
  // (1000000us/(16*60) us
  // timer 1 - set up to generate periodic interrupts for reading the ADC
  My_timer = timerBegin(1, // Timer 1
                        TIMERDIVIDER, // prescaler, 80MHz/4 = 20 MHz tick rate
                        true);  // true means count up
```

```
  timerAttachInterrupt(My_timer, &onTimer, true); // attach the ISR to read analog samples

  // we're trying to measure a sine wave (noisy one, but kind of a sine wave)
  // of a voltage coming off of the current transformer sensor
  //
  // We measure at a rate to get exactly 16 samples for every sine wave
  //
  float measIntervalSec = 1.0/(60.0*OSR); // for 16x osr, 1041.67 us

  int count = (int)(measIntervalSec*CLOCKRATE/TIMERDIVIDER + 0.5); // round to nearest integer

  timerAlarmWrite(My_timer, count, true); // timer for interrupts
  timerAlarmEnable(My_timer); // and finally, Enable the dang interrupt
}
```

The next code segment is the ISR itself. There's a lot of comments, but only two actual lines of code here! (Keep ISR's as simple as possible, but no simpler). There is a buffer in memory called samples[] which is large enough to store the max number of samples we're collecting (NSAMPLES).

At each interrupt, a sample is taken by calling the adcRead() function. The result is a 12 bit number from 0 to 4095 which is stored in the next available space in the samples[] buffer. When the buffer is filled (sampleCount >= NSAMPLES) the ISR just returns and doesn't take any additional samples.

So after a bunch of timer interrupts, each calling the ISR, we end up with a buffer (samples[]) full of digitized voltages on the ADC input pin. Once that is completely filled, we can proceed (in main code, not ISR) to make the computations necessary to compute the RMS value of the voltage from the CT, and thus compute the current flowing in the "hot" wire as shown in the formulas above.

C++

```
hw_timer_t *My_timer = NULL; // handling a timer on the ESP32 chip with interrupt

//-----------------------------------------------------------------------
// Timer Interrupt Service Routine (ISR)
//
// Software is using a timer with ISR
// to read the ADC and store samples in a buffer.
// Then in mainline code we will do the RMS calculation after all the
// samples are available.
void IRAM_ATTR onTimer()
{
  // the ISR is always active, but only runs the ADC when a sampling
  // is triggered.
  // To trigger the sampling interval, mainline code sets sampleCount=0
  // Then on the next ISR execution, it will start collecting samples
  // and storing them in the samples[] buffer.   When NSAMPLES have been
  // stored, the ISR stops reading the ADC until mainline code again
  // sets sampleCount=0
  //
  if ((sampleCount>=0) && (sampleCount < NSAMPLES))
  {
    samples[sampleCount++] = analogRead(ADCPIN);
  }
}
```

Next we need to actually take the measurement - which means taking all the ADC samples and storing them in the buffer. Then we need to do a bunch of calculations to convert all those samples to a single RMS voltage measurement!

In the code below you'll see several functions:

- readAnalogSamples() - trigger the reading of a buffer full of samples by setting sampleCount = 0 and then waiting for the ISR to fill up the buffer. When the buffer is full we will see the sampleCount be equal to NSAMPLES.
- measureRms() - this function does the work of processing all the samples into a single RMS voltage measurement.
    - first we compute the average or mean of all the samples. Remember all the samples are integer numbers between 0 and 4095 representing voltages from 0 to 3.3V. Because the waveform is offset by 1.66 volts, so all the ADC readings will have approximately half of the 12 bit count - or approximately 4096/2 or 2048 - plus or minus the value from the CT. So we compute the mean, and then subtract that from each sample to get the actual CT output voltage which will be positive and negative (samples[i] - mean).
    - Next we compute the sum of the squares of the CT voltage readings after the mean is subtracted. This is the variable sum which is a 32 bit integer.
    - Now to compute the RMS we divide sum by the number of samples and then take the square root of that value. This is the RMS value in ADC counts (0..4095). Then we multiply that by 3.3/4096 which converts the RMS value to actual volts!
- makeMeasurement() - again lots of comments, but just a few lines of code - call readAnalogSamples() to get the buffer full of samples, then measureRms() to convert all those samples to a single RMS voltage reading from the CT. Finally we'll convert the RMS voltage to "hot" wire amps by calling cvtRmsToAmps(). Finally we return the measurement as amps flowing in the "hot" wire.

C++ Shrink ▲

```
//---------------------------------------------------------------------
// Initiate a read of analog samples from the ADC
void readAnalogSamples()
{
  int dly=17*CYCLES;
  sampleCount = 0; // this triggers the ISR to start reading the samples

  // This should cause the ISR to read samples for next CYCLES of 60Hz (16.67 ms per cycle)
  delay(dly); // we're delaying for 17 cycles, by then the ISR should be finished reading the samples
  if (sampleCount!=NSAMPLES)
  {
    print("ADC processing is not working");
  }

  timerWrite(My_timer,0); // disable timer, we're done with interrupts
}

//---------------------------------------------------------------------
// Measure the RMS value of the samples recorded
float measureRms(int* samples, int nsamples)
{
  // this is tricky because of noise and because of the cyclic nature of the signal...
  //
  // first calculate the mean of the samples, or use something fixed
  // this is because the ADC measures positive voltages only (0 to +3.3 v)
  // so if we have a signal like a sine wave, we "bias" it to 3.3/2 volts
  // using an RC divider so that we can measure +- 1.65 volts in a 0 to 3.3v
  // scale.
  int32_t sum=0; // 32 bit sum
  for (int i = 0; i < nsamples; i++) sum += samples[i];
  int mean = (int)(sum/(int32_t)(nsamples));


  // RMS is root-mean-square
  // so compute sum of squares, divide by nsamples, take square root
  // now compute sum of (x-mean)^2
  sum=0;
  for (int i = 0; i < nsamples; i++)
  {
    int32_t y = (samples[i] - mean);
    sum += y*y;
  }
  float ym = (float)sum/(float)nsamples;
  float rms = sqrt(ym);
  rms = rms * 3.3/4096.0; // scale to volts, 3.3v (MAX) is a count of 4095
#ifdef WANTSERIAL
  //sprintf(msgbuf,"mean=%d",mean); print(msgbuf);
  //sprintf(msgbuf,"meansq=%ld",sum); print(msgbuf);
  //sprintf(msgbuf,"rmssamples=%f",rms); print(msgbuf);
  //sprintf(msgbuf,"rmsvolts=%f",rms); print(msgbuf);
#endif
  //for (;;) ; // temp - hang here forever
  return rms;
}

//---------------------------------------------------------------------
// convert measured vrms to amps using some primitive calibration data
// and linear interpolation
float cvtRmsToAmps(float vrms)
{
  // This converts the RMS reading (in volts) to an amperage reading (in amps)
  //
  // The nature of real life is (signal + noise + non-linearity)=measured value
  // We have tried to take out some of the noise component by averaging over
  // a few dozen cycles of the sine wave.
  // This routine will attempt to address the non-linearity portion by applying
  // a calibration.   We did a calibration by reading the RMS voltage with some
  // known loads with a "good" meter.   So we know the actual measured rms, the
  // actual watts, and we can compute a factor to convert between the two.
  //
  // calibration data from WattsUp watt meter, assuming 115Vrms
  //   0W  ->  0.00A -> measured 0.010 noise
  //   60W ->  0.52A -> measured 0.018 vrms      0.52/(0.018-0.01) = 65
  // 1070W ->  9.30A -> measured 0.122 vrms      9.30/0.122 = 76.2
  // 1590W -> 13.82A -> measured 0.198 vrms     13.82/0.198 = 69.8

  // expected with 50ma=100A primary current, and 33.3 ohm burden resistor (100/0.05)/33.3 = 60.06
  // so the ideal, linear, spherical constant is 60.06
  // We measured things like 65, 76.2, and 69.8 using known loads
  // So we're in the right ball-park.
  //
  // We'll constuct a piece-wise linear interpolation curve to take the actual
  // measured RMS of an unknown load and convert it to some calibrated current
  // value.
```

```
  //
  if (vrms < 0.01) return 0.0;
  if (vrms < .122) return vrms*(65.0+((vrms-0.01)/.122)*(76.2-65.0));
  if (vrms < .198) return vrms*(76.2+(vrms/.198)*(69.8-76.2));
  return vrms*(60.0+(vrms/1.67)*(60-69.8));
}

//-------------------------------------------------------------------
float makeMeasurement()
{
  // measure the current using the current transformer sensor
  // It generates some voltage that get's fed into the ESP32's Analog->Digital converter
  // Then we do some math to make that into current (amps)
  //
  // remember: measured signal = (real signal + noise + non-linearity)
  //
  float rms;
  float amps = -1.0;
  readAnalogSamples();  // read many chcles of the sine wave at like 16 samples/cycle

  if (sampleCount==NSAMPLES)
  {
    rms = measureRms((int*)samples, NSAMPLES);  // convert samples to an RMS voltage
    amps = cvtRmsToAmps(rms); // convert RMS voltage to amps
#ifdef WANTSERIAL
    //sprintf(msgbuf,"Measured=%f volts, Amps %f",rms, amps); print(msgbuf);
#endif
  }
  return amps;
}
```

At this point we have the amps of current in the "hot" wire, and need to decide what to do with it.

For the my application, the CT will monitor the "hot" wire current feeding a well pump. When the pump is "ON" it will draw 8-9 amps of current. Off of course is 0 amps.

There is an entry in the configuration file that gives a "AMPSON" current value, like 2 or 3 amps. If the measured current is more than this threshold, the code marks the pump as ON, otherwise it marks the pump as "OFF".

What the code does with the amps measurement is decide if the pump is "ON" or "OFF". When it detects a transition of the pump state (from "OFF" to "ON" or from "ON" to "OFF") it will record the new state along with a date/time stamp in the on-board log file. So we end up with a log file that has time-stamped ON or OFF events.

Then a few times a day (at 6 hour intervals is what the code is set at now) - during the wakeup period, the code will turn on the WiFi, connect to an AP, and try to upload the log file to the FTP server specified in the configuration file. If it is successful, the log file on-board is deleted and we start with a clean log file. If unsuccessful at uploading, the log file is untouched and additional data is added to it.

This means that if the IoT device is in range of a suitable WiFi signal it will upload the data, otherwise it will just keep accumulating it on-board to be uploaded at a later attempt.

Here is an example of the log entries that are recorded and uploaded to the FTP server. The ON and OFF events show the current in amps that the ESP32 measured.

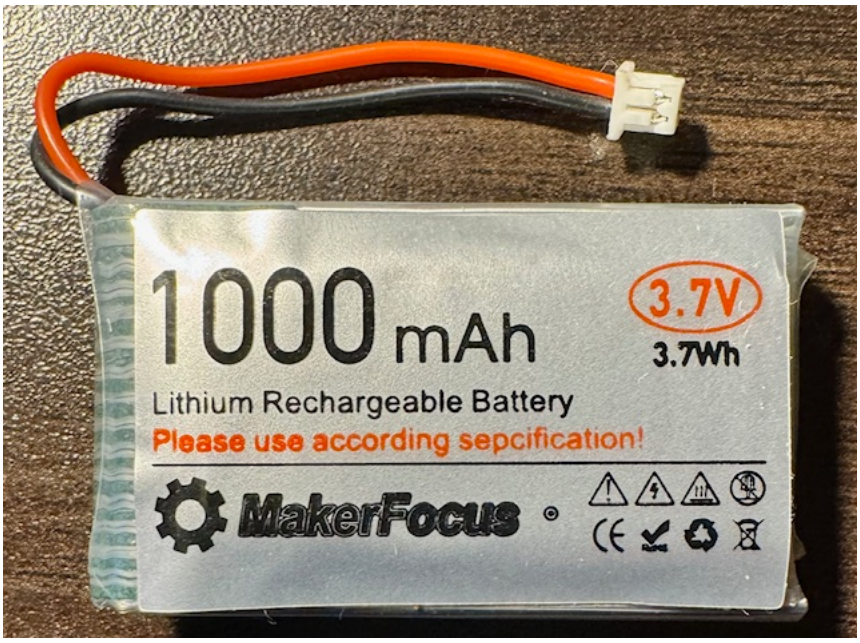
```
2024/01/27,10:04:21,ON,8.919955
2024/01/27,11:56:32,OFF,0.000000
```

Low Power Handling

To make it possible for this application to run for a significant period of time on just a battery, several things are necessary.

First we need to choose an ESP32 board that supports battery operation. The board used here allows a tiny battery to be plugged in. When the ESP32 board is powered by the USB connection, the board will charge the battery. When the USB is unplugged, the ESP32 is powered by the battery.

I found these batteries from Maker Focus that are 1000 mah capacity. In this application I've found that they can power the IoT system for about 20 days between charges. I've connected a small solar cell with USB output to the board's USB port to "top off" the battery when there's some sunshine. With this arrangement it can run almost forever - unless you live somewhere that has no sunshine for 20 days in a row! (Sorry about that).



After choosing a board and battery, we need to make sure the circuit used is using as low a power as possible. In this case, we're driving the voltage divider network from a GPIO 17 pin, rather than directly from the 3.3V source on the ESP32 board. This means that the voltage divider network won't draw any power when we're in deep-sleep mode and GPIO 17 is set to 0 volts.

Finally we need to design the software on the ESP32 to really be power aware. Several things can be done here:

- use "deep sleep" mode as much of the time as possible
- Keep the non sleep times as short as possible
- Run the CPU at a slower clock rate if possible
- Keep the WiFi radio OFF as much as possible.

To accomplish this the following are designed into the software:

- The ESP32 will be mostly in deep sleep mode, waking occasionally to read the CT
- A log file in the on-board flash file system will store the results
- The ESP32 will be down-clocked to 80MHz from normal 240 MHz.
- The WiFi radio will only be turned on a few times a day to try to upload measurement data.

Using these techniques the ESP32 system I fielded can run for 15-20 days on a 1000mah battery. If a solar cell is connected, the system can run almost indefinitely.

Deep Sleep Mode

The ESP32 platform includes a number of "modes" which utilize varying degrees of power when they are operational. Normal or Active mode operation means the ESP32 is running its main CPU continuously. This is the mode which utilizes the most power because most of the chip is operational - the CPU, the memory, the flash, timers, GPIO/ports, and even perhaps the radio (WiFi or Bluetooth).

To conserve power, there are some lower-power modes that can be employed - basically, this means that parts of the chip are powered down and some functions and features cannot be used.

In Active mode, some power savings can be achieved by turning off the radio when its features are not needed. Additionally, the CPU clock can be "turned down" or run at a slower rate. This causes the chip to execute slower, but reduces the power used in active mode.

In addition to active mode, there are like five different power-saving modes - which is too many to describe here. The chip manufacturer has a good description of these modes here.

For this application, we've chosen to use the mode with (almost) the lowest power consumption - around 10 microamps (for the chip alone) - called deep sleep mode. (**Note**: I measured 20ua in deep sleep mode for the entire board fed from a 5V power source). In this mode, almost all the chip's functions are depowered. The only parts still active are:

- RTC controller
- ULP coprocessor
- RTC FAST memory
- RTC SLOW memory

This means the CPU, the RAM, almost all the peripherals are depowered. Power consumption is reduced to the micro-watt level in this deep sleep mode.

The only things "awake" during this mode are some small amount of special RAM memory and the bits of circuitry to keep track of what time it is and how long the chip has been in deep sleep - and also a GPIO for external signaling.

To recover from this deep sleep mode, we need to "wake up" the CPU - which means going back into active mode. However, in deep sleep mode, the CPU and memory system are powered down - so no instructions are executed and no RAM memory contents are preserved. Of course, flash memory where the program itself is stored is preserved, although it is powered down during deep sleep mode.

"Wake up" means the process of basically rebooting the CPU from a completely powered down mode. During this boot-up process, the code needs to know if it is doing a boot up because of what is called a cold start - meaning the system has just been powered on and is booting up for the first time, or if the boot up is due to a wake-up event from a low power mode such as deep sleep.

Fortunately, the manufacturer has provided special IO registers that can be read to determine what type of boot up is happening. Then, the code can handle the boot process differently based on what it has read from this boot-up reason register.

Below, you'll see the code at the beginning of the `setup()` function that executes on every boot or wake up cycle. There's a call to a function `esp_sleep_get_wakeup_cause()` to get a code that describes the type of wake-up that is being done. Later on in the `setup` function, we will use this code to determine what type of wake up processing to do.

C++ Shrink ▲

```
void setup()
{
  // --- deep sleep mode note ---
  // Since this application puts the ESP32 into deep sleep mode most of
  // the time, waking from deep sleep means the contents of memory are
  // unknown, so the entire program is read back into memory from flash
  // and we execute the setup() routine.   All the application logic
  // happens here, then we go back to sleep.
  //
  // So the normal loop() function is never actually executed!
  //
  char buf[32];
  esp_sleep_wakeup_cause_t wakeup_reason;
#ifdef WANTSERIAL
  Serial.begin(115200);
#endif
  print(SIGNON);

  // see why we woke up
  wakeup_reason = esp_sleep_get_wakeup_cause(); // see why we woke up - cold boot or deep sleep

  setCpuFrequencyMhz(80); // take it easy on CPU speed to reduce power consumption

  switch (wakeup_reason)
  {
    case ESP_SLEEP_WAKEUP_EXT0     : print("Wakeup caused by external signal using RTC_IO"); break;
    case ESP_SLEEP_WAKEUP_EXT1     : print("Wakeup caused by external signal using RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER    : print("Wakeup caused by timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : print("Wakeup caused by touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP      : print("Wakeup caused by ULP program"); break;
    default : print("Wakeup was not caused by deep sleep"); break;
  }

  // Initialize SPIFFS (file system)
  if(!SPIFFS.begin(true))
  {
    print("An Error has occurred while mounting SPIFFS");
    //return; what to do here?  We can't do much without the file system
  }

  pinMode(LED,OUTPUT);
  pinMode(VPPPIN,OUTPUT);

  digitalWrite(LED,HIGH); // use this as 3.3v src for measurement voltage divider
  digitalWrite(VPPPIN,HIGH); // power the current tx sensor for a little bit
```

The code at the bottom of the setup() function is what puts the ESP32 back into deep sleep mode. It calculates the time to sleep (tts) by taking the total sleep period and subtracting how long the ESP32 was awake for this cycle. This is passed in the esp_sleep_enable_timer_wakeup(tts) call to set up a time to wake up the ESP32 at the right time. Then the esp_deep_sleep_start() function turns out the lights until the next wakeup cycle.

C++

```
  // and go back to sleep here
  ++bootCount; // count this boot-up sequencerm
  digitalWrite(LED,LOW); // turn of comfort LED
  digitalWrite(VPPPIN,LOW); // set voltage divider output pin to 0v
  sprintf(buf,"Awake for %d ms",millis());
  print(buf);
```

```
long tts = (timeToSleepMs - millis()) * ms_TO_uS_FACTOR;
esp_sleep_enable_timer_wakeup(tts);
esp_deep_sleep_start();

// and, the ESP32 never executes any code past the deep_sleep_start!
```

Communications

The software supports two communication functions - one to get the current date/time from an NTP server over the WiFI connection (so it knows the correct date and time). The second function is an FTP upload function which reads any log entries and uses FTP to append these entries to a log file on the FTP server.

If the ESP32 is connected to the internet through WiFi, it can reach out to a special server to find the current time and date. These servers are called Network Time Protocol servers or NTP. The software uses an NTP client to get the date/time and set it in the ESP32 so that the ESP32 can keep track of date and time going forward.

The ESP32 will also try to upload the IoT data it's gathered (ON and OFF events) to a remote computer using an FTP client module. This allows the ESP32 to automatically upload it's data for further archiving and processing to a remote computer system. The FTP upload uses APPEND mode to append the data to an existing file on the remote server. The path and file name on the remote server are specified in the configuration file that is read on boot up.

In the setup() function you will see code that takes the measurement and logs it, as well as code that will periodically (every 6 hours) try to connect to WiFi and upload data.

C++ Shrink ▲

```
if (wakeup_reason == ESP_SLEEP_WAKEUP_TIMER)
{
  // most of the time, the app wakes up from deep sleep and makes
  // a current measurement.   If the current is above some threshold
  // value (specified in config.ini file) then we consider the load (pump)
  // is ON, otherwise it's OFF.
  // If it changed state (OFF->ON) or (ON->OFF) we log that in the flash
  // file system
  // Probably the pump will only go one once per day, so most of the time
  // not much happens and we just go back to sleep.
  //
  // However, every so often (currently 6 hr intervals) we turn on WiFi and
  // try to upload any data to a remote server.
  //
  float amps;

  String ttag = rtc.getTime("---Time=%Y/%m/%d,%H:%M:%S");
  print(ttag.c_str());

  // ------- do wakeup tasks here
  setupMeasurement();
  amps = makeMeasurement();
  sprintf(buf,"Amps=%f",amps);
  print(buf);
  logMeasurement(amps);

  // ------- do hourly tasks here
  int hr = rtc.getHour();
  if (lastHr == -1)
  {
    lastHr = hr;
  }
  else if (lastHr != hr)
  {
    lastHr = hr;
    if ((hr % 6) == 0) // every 6 hours
    {
      // hourly tasks
      connectToWiFi(); // first connect this ESP32 to the WIFI network
      pushDataToServer();
      getNtpTime(); // resync time
      turnOffWiFi(); // turn off when done
    }
  }
}
```

Special credit and thank you to Rui Santos (Random Nerd Tutorials) for the NTP and FTP server features. Rui has an extensive web site of examples of using many different features of the ESP32 and attached sensors and actuators - well worth checking out.

Cold Boot Processing

In the setup() function the software implements a set of tasks to be done only on cold boot. These are mainly reading the configuration file and connecting to WiFi and getting the current date/time from NTP. The software stores the configuration data in a special memory (RTC memory) which is the only memory that is preserved after a wake up from deep sleep mode. This way the ESP32 doesn't have to read the configuration file every time it wakes up - although it could do that, it would take many millions of CPU cycles each wake-up time. So to conserve power we read this information only one time at cold boot and not on each wake up from deep sleep mode.

C++ Shrink ▲

```
// Stuff to do on an initial boot from power on or reset (not done on wakeup)
if (bootCount == 0)
{
  // read needed data from the config file - stored in RTC memory so it survives deep sleep
  readKey(CONFIGFN,"SSID=",ssid,127);
  readKey(CONFIGFN,"PASSWORD=",password,127);
  readKey(CONFIGFN,"FTPHOST=",ftpHost,127);
  readKey(CONFIGFN,"FTPUSER=",ftpUsername,63);
  readKey(CONFIGFN,"FTPPASSWORD=",ftpPassword,63);
  readKey(CONFIGFN,"FTPFILE=",ftpFile,127);
  readKey(CONFIGFN,"FTPPATH=",ftpPath,127);
  readKey(CONFIGFN,"AMPSON=",buf,31);
  ampsForOnMeasurement = atof(buf);
  readKey(CONFIGFN,"MEASUREMENTINTERVALMS=",buf,31);
  timeToSleepMs = atol(buf);
  Serial.print("Meaurement Interval (ms) ");
  Serial.println(timeToSleepMs);
  if (timeToSleepMs < 2000) timeToSleepMs = 2000;
  if (timeToSleepMs > 120*1000) timeToSleepMs = 120*1000;

  // initialize the RTC
  rtc.setTime(0,0,0,1,1,2024); // default time 00:00:00 1/1/2024
  if (connectToWiFi() == 1)
  {
    needNtp = true;
    getNtpTime(); // get NTP time if possible
  }
  turnOffWiFi(); // now we can turn off the WiFi modem to save power
}
```

Points of Interest

Learning about current transformer (CT) sensors, and how to measure RMS were really interesting exercises.

Learning about some linear piecewise calibration to take care of some of the integral non-linearities of the sensor was also - well - captivating - it was a simple concept, but the implementation took a bit of thought. It could be made more general in the future.

The source is available on GITHUB

History

Version 1.0, January 2024

License

Written By

deangi

Team Leader

United States

Just a tinkering retired engineer interested in Internet-Of-Things devices, building them, programming them, operating them.

Comments and Discussions

Posted 28 Jan 2024