A DIY web enabled water meter

deangi
16 Jan 2024 CPOL 0

An ESP32 is used to track water usage and serve web pages with water use data

An ESP32 is connected to a water meter and tracks water usage. Water use reports are stored in flash memory. A web server and telnet server allow water use data to be accessed over the network via a WiFi connection.

Introduction

Water is becoming a resource of increasing concern in many parts of the world. Utility served water connections typically meter usage and provide data and billing in monthly increments. Well served water connections may not even have a metering system installed.

In my case the second scenario (well connection) was in place. A few summers ago a neighbor mentioned that they were having trouble with their water well running low and not being able to supply their needs.

This conversation caused me to realize that I had no way of knowing how much water was being used, or if the well was keeping up (there is a large reservoir tank). In fact until the day came when I turned on a tap and no water came out, I would be completely unaware of any problems.

Thinking about this and looking around a bit I discovered there were some relatively inexpensive water meters available that could be installed, and to my delight I found one that had a wire coming out of it. The tinkerer mindset kicked in and I began to scheme that the wire along with a suitable IOT

system (internet of things) could probably allow me to get a handle on how much water was being used.

A bit of research lead me to the ESP32 platform which contained not only the necessary GPIO but also provided WiFi and Bluetooth radios for communications. In addition there seemed to be sufficient memory and flash to allow the creation of a suitable IOT application to track water use.

Background

Development boards with the ESP32 architecture are readily available and relatively inexpensive. Searching Amazon will reveal hundreds of suppliers and form factors. I choose a board with a small display built in and a capacity for a battery backup. The board was called a LILYGO ESP32 T-Display Module and is available for about $24 or less on Amazon.

This module could be easily connected to the water meter, and would track water use. Because of the availability of a WiFi connection, it can act as a sort of miniature web server and serve a page describing the water use it was measuring.

Software Architecture

The software consists of an application developed using the Arduino IDE with the ESP32 board development kit installed. In addition a number of libraries are included to handle things like WiFi

connections, Network Time Protocol (NTP), a miniature web server, and a telnet server.

In addition, the main job is to watch for and count pulses from the water meter to track how many gallons of water are used. So we have basically software with a measurement part and a communications part.

Measurement Software Part

The measurement was enabled by installing a water meter (done by a registered plumber!). As I looked for water meters, I noticed one with a wire coming out of it. This intrigued the tinkerer engineer part of me - wondering could I use this to have some embedded system track water use?

I found this one on Amazon - there are probably many others available with similar capabilities. As you can see it has a mechanical meter tracking water flowing through the meter, and it also has a wire coming out which I was sure would enable some kind of computerized measurement of the water use.

Shortly after having the meter installed, I was there with an oscilloscope connected to the wire to see what kind of signal might be present. Two conductors came out of the meter, and it soon became apparent that they were connected to some kind of internal switch. This switch would open and close at certain times indicating the amount of water use flowing through the device.

A bit of research and further testing turned up the fact that inside the water meter was a tiny turbine which turned when water was flowing through the meter. Through some gears, this turbine was connected to the mechanical counter inside the meter so you could read the water use on the mechanical meter. In addition it turned out somewhere there was a magnetic switch which was positions so that a magnet attached to some gear would move past the magnetic switch causing it to open and close as the turbine turned.

Further testing indicated that this switch closed and opened once for each 10 gallons of water that flowed through the meter. So if I could somehow convert this switch open/close cycle to a voltage, it could be read by the ESP32. This is easily accomplished using a circuit known as a "pull up" resistor as shown below.

As you can see, a single ESP32 input port pin (Called GPIO17) is connected to sense the switch state. The 10K resistor connected to 3.3 V means that if the switch is open, the GPIO17 pin will see 3.3V which will be read by the ESP32 as a '1'. When the switch is closed, the GPIO17 pin is connected to ground (GND) which means GPIO17 will see 0V and will be read as a '0'.

So this means the ESP32 software can read '0' or '1' and that indicates the switch is open or closed. So as water is flowing through the meter, the ESP32 if it keeps reading the GPIO17 pin will see it as a '1', then for a while it will be '0', then it will go back to '1'.

If we look at the waveform, we would see this:

Switch Bounce

This is the ideal waveform, but in the real world, there's always noise. In the case of a switch, we almost always have a mechanical device that causes two conductive arms to touch (closed) or not touch (open). When we mechanically move these, there is an instant of time where the arms make first electrical contact. But due to the momentum of the movement, these two things slam into each other, and may momentarily "bounce" a bit - sort of like dropping a ball on the ground. This bouncing behavior causes multiple (not one) open/close cycles at the millisecond level.

Since a computer can read it's GPIO very fast, if we're not careful in the software, the computer will see a single closure of the switch as 2, 3, or even more momentary opens/close cycles. Since this would confuse our water meter software, the software needs to be able to ignore this bouncing behavior and count it as only one cycle.

To perform the measurement, the software needs to look for an "edge" in this pulse, either the high to low (falling) edge or the low to high (rising) edge. There will be a variable that counts these pulses and add one to it each time it sees a pulse. In this case the software will look for the falling edge (since that is the start of a new 10 gallon interval from the meter). By multiplying the number of falling edges by 10, we can measure the flow through the meter in gallons.

Interrupts and state machines, oh my!

To accomplish this we need to realize that the ESP32 that is running this application is doing several different things - it wants to count these pulses, but it also wants to service web requests, and telnet requests, so we need to be careful that while the ESP32 is servicing some request, it doesn't miss a pulse coming in.

To accomplish this, we'll use a timed interrupt - which means that a certain special function will get called at a regular periodic interval. We can set up a piece of internal hardware in the ESP32 called a 'timer' to do this.

An ESP32 timer can be set up to generate an ESP32 processor interrupt - say every 1ms (1000 times a second). This interrupt can be configured to call a special function - called an Interrupt Service Routine or ISR. This means the ISR will be called 1000 times a second, no mater what else the ESP32 is doing - the ESP32 will take time away from what it's doing to call this ISR.

Now, we need to be really careful in an ISR to do just the very bare minimum amount of work - since it's being called 1000 times a second and all. So we will write the ISR with some care to make sure it does the minimum necessary work, and uses a minimum of time and other resources. In other words, we won't allocate dynamic memory in the ISR, we won't do any serial port print outs, we won't do any really complex calculations, etc.

What this ISR needs to do is to sample the GPIO pin that is connected to the water meter signal as shown above. We will be looking for a time when this signal reads as a '1' during one ISR call, and then 1/1000 of a second later when the ISR is called again, the GPIO signal reads as a '0'. This indicates a falling edge (high to low transition). When we see this condition, we'll just ignore the input signal for a certain amount of time (the debounce interval - say 50 or even 100ms). Then we'll read it a second time, and if it's still a '0' - we'll declare a falling edge has been detected.

To do this, we'll set up the ISR as a "state machine" - which is a software construct where the are some defined states and the code decides how to move between these states. In this case we have the following states:

- (state 0) Looking for an edge (change in reading of GPIO pin from last ISR call)
- (state 1) Waiting for debounce delay time to expire
- (state 2) Reading the GPIO pin again, and seeing if it's really an edge
- (state 3) handling the detected edge - if it's a falling edge, increment the pulse counter

The software will need three variables to implement this state machine:

- pulse counter - to be incremented when we see a debounced falling edge
- current state - which state (of the above three) the state machine is currently at
- last GPIO reading - value of the GPIO pin at the previous execution of the ISR

Here is the code for the ISR that implements the state machine. You'll see that this ISR detects both rising and falling edges and calls an edge handler routine - which uses falling edges to increment the counter.

C++ Shrink ▲

```
int state=STATE_WAIT_FOR_EDGE;
int lastGpioState=LOW;
int debounceCounter=0;
int edge=0;

#define BOUNCE_WAIT (100) /* ms */

//-------------------------------------------------------------------
// Handle a debounced detection of a rising (0-1 transition) or
// falling (1-0 transition) of a pulse

void handleEdge(int edge)
{
  // count only falling edges as 10 gallons!
  // -1 means rising, +1 means falling
  digitalWrite(LEDPIN,(edge<0 ? HIGH : LOW));
  if (edge > 0) pulseCounter++; // count falling edges
}

//-------------------------------------------------------------------
// This is ISR (interrupt service routine) code that gets run
// by the action of a continuous running timer inside the ESP32
// In SETUP we will program the timer to generate an interrupt
```

```
// every millisecond.   Then we can expect this code to run
// once every millisecond.
//
// Every millisecond, the code will look at what state the
// state machine is in and take appropriate action, moving from
// waiting for the first edge, ignoring the switch for a debounce
// time, and handling the edge if one was detected.
//
// ** IMPORTANT NOTE **
// because this is an ISR, we want to minimize the amount of work
// to be done.   In this case just reading some GPIO pin, some if
// statements, managing the state variable, and incrementing the
// pulseCounter variable.
// NO Serial IO!   NO long calculations, no creating objects or
// destroying objects.  K/I/S/S - keep it simple and stupid.
//
//----------------- Interrupt Service Routine ------------------------
void IRAM_ATTR onTimer(){
  // simple state machine
  int currentGpioState=digitalRead(INPUTPIN);
  if (state==STATE_WAIT_FOR_EDGE)
  // -- first state: WAITING for an initial edge by reading the GPIO and
  //    seeing if it changed from last time we read it,
  //    which may indicate that the switch is changing state
  {
    if (currentGpioState != lastGpioState) // found first edge, might be noise
    {
      // ok, the GPIO pin has changed state, now we ignore it for
      // a while by going to the debounce wait state.
      state=STATE_DEBOUNCE; // wait for a while and see if it was noise
      debounceCounter=BOUNCE_WAIT; // how long are we going to ignore it?
    }
    return;
  }
  else if (state==STATE_DEBOUNCE)
  {
    // we're just tapping our fingers, ignoring the switch, for a while
    // we use a counter to count down until it gets to 0 then we check
    // if the gpio pin has really changed

    if (debounceCounter==0)
    {
      // Ok, the ignore timer has expired, and we check the gpio pin again
      currentGpioState=digitalRead(INPUTPIN);
      if (currentGpioState != lastGpioState) // ---> yep, it is still an edge, count it as
debounced
      {
        state=STATE_HANDLE_EDGE; // go to handle edge state next interrupt
        edge = lastGpioState-currentGpioState; // -1 means rising, +1 means falling edge
        lastGpioState = currentGpioState;
      }
      else // ---> nope, it really didn't change, so maybe it was was just noise
      {
        state=STATE_WAIT_FOR_EDGE; // let's just go back and wait for an edge some more
      }
    }
    else
```

```
    {
      // counter hasn't gone down to 0, so we're just waiting for a while
      // and ignoring the GPIO pin
      debounceCounter--;
    }
    return;
  }
  else // STATE==STATE_HANDLE_EDGE
  {
    // Finally, we've found a debounced edge, let's call this to handle it
    handleEdge(edge); // called for rising (0->1) and falling (1->0) edges
    state=STATE_WAIT_FOR_EDGE; // and now we go back to waiting for the next edge
  }
}
```

Now we have some software that reliably detects falling edges and counts them, so we know how many pulses have come from the meter. We just need to multiple the pulse count by 10 to get the number of gallons that have passed through the meter. Unfortunately the meter may not have started out a 0, or our ESP32 system may have been off-line during some pulses, so there may be some meter offset amount of gallons that have passed through the meter when the ESP32 wasn't on line. So we compute the total use as the pulseCounter time 10 plus some meter offset value. We use "long" or 32 bit integers for this calculation since normal "int" integers can only count to +32767 gallons.

```
//--------------------------------------------------------
// Read amount of water used
#define GALLONSPERPULSE (10L)
long ReadGallons()
{
  return meterOffsetGallons + pulseCounter * GALLONSPERPULSE;
}
```

There is a bit of setup code that is necessary (in the setup() function) to configure the timer for the appropriate delay value (1ms in this case) and to attach the ISR to an interrupt. There are three available timers (0, 1, and 2). Timer 0 is already used by the base ESP32 code, so we'll use timer 1 for this application:

```
  //------------ Set up the timer and ISR -----------------
  // timer 1 - 1ms interrupts for debounce logic

  // select timer 1 and set it put for 1MHz count rate
  My_timer = timerBegin(1, 80, true);
  timerAttachInterrupt(My_timer, &onTimer, true);
  timerAlarmWrite(My_timer, 1000, true); // timer for 1ms interrupts (1000 counts a 1MHz)
  timerAlarmEnable(My_timer); // Enable the interrupt
```

Communications Software Part

To allow the meter to communicate it's measured results, we need to somehow get the data from the ESP32 to the outside world. The application has been equipped with actually four methods for doing this - most of which are for diagnostic purposes only. These methods are:

1. Serial (RS232) communications - diagnostics and debugging - only when plugged into USB
2. On-board display showing usage - diagnostics and debugging - local visual only
3. Telnet communications - diagnostics and debugging as well as remote access
4. HTTP (web) communications - main remote access

For serial port access, this is only available when some remote access wired connector is plugged in - it's usually a USB cable plugged into a PC of some kind. It's not very convenient - but can be quite useful for initial development and debugging.

There is a small on-board display that can show 6 or 8 lines of text. Typically this shows the latest measurement as well as the IP address assigned when the ESP32 connects to WiFi.

Web Access To The Water Meter

Once connected to WiFi, the meter data can be accessed via a browser like Edge or Chrome by simply typing in the IP address of the meter. The ESP32 application has a tiny web server built in which can serve a few pages showing the water use data. The IP address is displayed on the first line of the on-board display.

There are several HTTP pages that are supported by the web server.

- /
    - This is the main page as shown above
- /log
    - This is to show the entire contents of the log file. Note: If the log file gets too large, use the compress page below to shrink it to one entry per day.
- /daily
    - This is to show the water use at daily level
- /compress
    - This will compress the log file daily water use reporting
- /maintenance?metercorrection=123
    - This will set an "offset" which can accommodate for losses or inaccuracies in the water use level. The next recorded log entry will include this offset value in water use reporting.

There is some code required to set up the web pages of the ESP32 internal web server. This code happens during the setup() function of the ESP32 Arduino IDE application. You will see the calls to

server.on() which set up the paths for supported pages along with some code to be executed when that page is requested. This is a bit of "newer" C++ which basically allows you to define an anonymous function to be called when that web page is requested.

What happens is that when the web page is requested, the anonymous function that has been set up by the server.on() command will be executed, and the resulting page will be sent to the requestor.

In the case of the home page for example, there can be "variables" which are in the actual home page but before the page is sent out, the variables are replaced with some other string value. This is how the variable parts of the page like the time and date and measurement data are placed onto the web page that is served to the requestor.

Shrink ▲

```
//--------------------------------------------------------
// Setup the expected web page handlers
// - This is an assynchronous web server, when requests
//   come in - specific handler request routines are
//   called.
//
// Supported pages
//  /    - index.html - show root page to the requester
//  /log - show the log file to the requester
//  /daily  - show the dayly (summarized) log file to the requester
//  /compress - compress the log file and show the compressed file
//  /maintenance?metercorrection=123    - set a meter correction factor
//--------------------------------------------------------
// Route for root / web page
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
  request->send(SPIFFS, "/index.html", String(), false, processor);
});

// Route to load style.css file
server.on("/style.css", HTTP_GET, [](AsyncWebServerRequest *request){
  request->send(SPIFFS, "/style.css", "text/css");
});

// Route to read log file
server.on("/log", HTTP_GET, [](AsyncWebServerRequest *request){
  request->send(SPIFFS, LOGFN, String(), false, processor);
});

// Route to summarize log file (V0.8)
server.on("/daily", HTTP_GET, [](AsyncWebServerRequest *request){
  summarizeLogFile();
  request->send(SPIFFS, SUMMARYFN, String(), false, processor);
});

 // Route to summarize log file (V0.8)
server.on("/compress", HTTP_GET, [](AsyncWebServerRequest *request){
  summarizeLogFile();
  compressLogFile();
```

```
    request->send(SPIFFS, LOGFN, String(), false, processor);
  });

  // Route to maintenance page
  server.on("/maintenance", HTTP_GET, [](AsyncWebServerRequest *request){
    String message; // /maintenance?metercorrection=123

    if (request->hasParam("metercorrection"))
    {
      message = request->getParam("metercorrection")->value();
      long correction = message.toInt();
      meterOffsetGallons += correction;
      String msg = String(rtc.getTime("%Y/%m/%d,%H:%M:%S,")+String(correction)+String(",gal.
(CORRECTION FROM WEB)"));
      appendToLogFile(msg);
    }
    request->send(SPIFFS, LOGFN, String(), false, processor);
  });
```

Here is an example of a function called to supply the value of a variable. You will see that it handles variables called "READING" and "LOGINFO".

```
//-------------------------------------------------------------
// Replaces placeholder with value in web pages
// called from deep within the ESP32 webserver somewhere
// when a page is requested that needs some variable data
String processor(const String& var)
{
  // This is a service routine called by the web page service
  // to generate dynamic content - in this case date time and
  // current water use reading - to put on the page served to
  // the http requestor
  if(var == "READING")
  {
    return String(rtc.getTime("%Y/%m/%d %H:%M:%S  "))+String(ReadGallons())+String("gal.");
  }
  if(var == "LOGINFO")
  {
    return String(SIGNON);
  }
  return String();
}
```

In the root file system, there are three files that are needed.

- config.ini
- index.html
- style.css

These three files are included in the github source repository - you'll see a link at the end of the article if you want to download the entire source code.

Telnet Access To The Water Meter

There is also a built in Telnet server which allows a command-line access to the usage data and a few other diagnostic features. This works with standard Telnet clients that serve a command line interface to the ESP32 (not SSH, plain old Telnet). A very simple "shell" like command interpreter has been built in the software which supports the following commands:

- ls
  - Example: ls
  - This will display a list of all the files in the root folder - which is /
- cat
  - Example: cat /config.ini
  - This will type out the contents of a file. All file names must be prefixed with the /
- cp
  - Example: cp /config.ini /config.bck
  - This will copy a file - for example copy file config.ini to a new file config.bck
- rm
  - Example: rm /config.bck
  - Remove a file from the root folder
- ap
  - Example: ap /test.txt This is a line of text to append to the end of the file
  - Append a line of text to a file
- report
  - Example: report
  - Report last water use entry and meter status

The Telnet server is not SSH equipped. So you need to use standard Telnet to access it. I often use the open-source application putty. Here is an example of the putty setup screen for accessing the Telnet server:

Configuration File

There is a small configuration file that contains four values that needs to be read on startup. This is placed in the root folder of the SPI Flash File System (SPIFFS) named config.ini

When the ESP32 boots up these values are read to configure the water meter for your application.

The required values are:

- SSID=wifissidname
- PASSWORD=wifipassword
- TIMEZONE=0
- METEROFFSET=0

The SSID and PASSWORD must be set so the water meter can attach to the WiFi network.

The TIMEZONE value must be set to the number of seconds offset from GMT for your local time zone. For example, USA west coast is at time zone GMT-8 so the offset is -8 hours or -28800 seconds.

The METEROFFSET is used to give some initial value to the water usage in the case that some water has passed through it before the ESP32 was attached. Note it is easy to just set this to 0 and to use the web interface to program in any necessary offset value (/maintenance page - see above)

Robustness

The water meter has been in use for over a year, and several improvements have been made to assist in stability and robustness. After all one does want something like a water meter to be quite boringly reliable and stable.

- WiFi reconnect - if the WiFi access point goes off-line, the software will detect this and automatically try to reconnect. This is checked once per second - so if the WiFi goes down, when it comes up it should reconnect fairly quickly - within 20-30 seconds after the WiFi comes back online.
- Auto-reboot - another stability addition was to automatically restart the water meter at 2AM in the morning. The meter will go off-line during this period - but it's fairly brief (a few seconds). This should help with keeping the meter working if some unexpected error happens. If the meter is down and not coming back on-line, hopefully after the next 2AM reboot it will be back in service.

Credits

Special credit goes to Rui Santos for lots of useful code including services for NTP, HTTP, and Telnet. Great work Rui! https://randomnerdtutorials.com

Source code on Github.

History

V1.0, Jan 14, 2023

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.

Watch

Comments and Discussions

Discussions on this specific version of this article. Add your comments on how to improve this article here. These comments will not be visible on the final published version of this article.

Add a Comment or Question | Email Alerts | Search Comments

Spacing Relaxed | Layout Normal | Per page 25 | Update

-- There are no messages in this forum --

Discussions posted for the Published version of this article. Posting a message here will take you to the publicly available article in order to continue your conversation in public.

Add a Comment or Question | Email Alerts | Search Comments

Spacing Relaxed | Layout Normal | Per page 25 | Update

-- There are no messages in this forum --