Let's build…
An NTP Server

By Bruce Hall, W8BH

PART 1: BASIC TIMEKEEPING

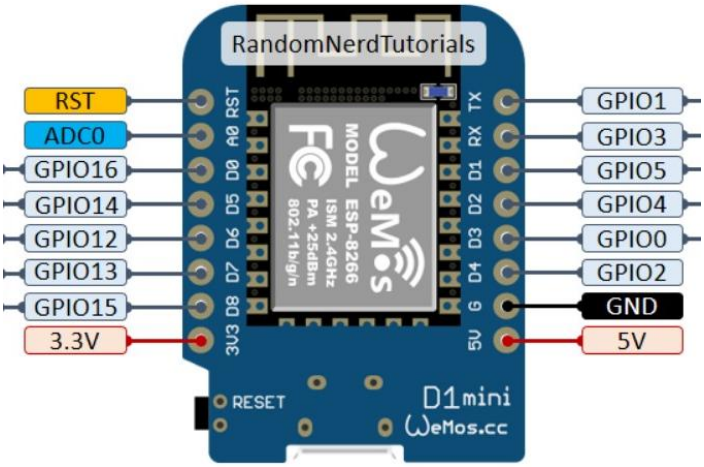
"What's that?", you might ask. Is it a clock? Is it a Wi-Fi server? It's a little of both.

NTP stands for Network Time Protocol, which is the basis of distributing time information on the internet. Developed in the 1980's by David Mills, it is perhaps the oldest internet protocol still in active use.

This device is a GPS-disciplined clock *with microsecond resolution*. It provides stunningly accurate time information to any device on the local wireless network. It does not suffer the delays associated with remote time providers. And it is especially useful when Internet-based services are not available.

I like to start small and build my projects step-by-step. With each step I add a piece of hardware and/or software, testing as I go. It is a great way to learn. Read on if you are interested in building an accurate AND inexpensive NTP server.

STEP 0: THE ESP8266



We will base our NTP server on the Expressif ESP8266. This small module combines an 80 MHz, 32-bit MCU and Wi-Fi radio. It is available in several different configurations. The one shown here, suitable for breadboard prototyping is the Wemos D1 mini. A good reference guide to the various modules and pinouts is here: [ESP8266 Pinout Reference]

I assume that you are comfortable with the Arduino IDE, have programmed an MCU before, and have an ESP8266 board in your

hands. The obligatory first step is uploading a blink sketch. Again, head on over to Random Nerd Tutorials for helpful setup information: Installing ESP8266 in Arduino IDE. Briefly,

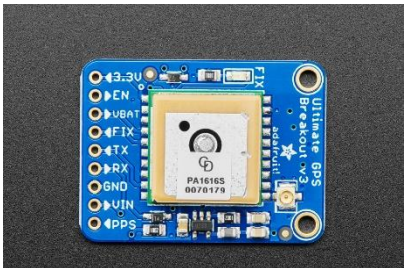
1. In the Arduino IDE, go to File > Preferences.
2. Select "Additional Board Manager URLS and add the following URL: http://arduino.esp8266.com/stable/package_esp8266com_index.json
3. Go to Tools > Board > Boards Manager.
4. Search for ESP8266 and press install for "ESP8266 by ESP8266 Community". Wait until installed.
5. Return to Tools > Board and select ESP8266 Boards > D1 R2 and Mini
6. Plug the board into your computer using a USB cable.
7. Go to Tools > Port and select the port your board is using.

The D1 mini has a built-in LED on GPIO2. Its logic is inverted: turn it on by writing logic 0 and turn it off by writing logic 1.

Upload a blink sketch to your board! In the IDE, go to File > Examples > 01. Basics > Blink and upload in the usual manner. If successful, the LED (located next to the PCB antenna at the top right corner of the board) will blink once every 2 seconds. Since many boards are already programmed to blink, I like uploading a "double blink" sketch instead: step 0.ino. It lets me know that *my* upload was successful.

STEP 1: ADDING A GPS MODULE

Let's add some hardware. GPS modules are now inexpensive and widely available. For testing I use the Adafruit module (#746), the uBlox NEO-M8N, and Amazon clones based on the NEO-6M.



Adafruit "Ultimate GPS" module #746

Connect your GPS module to the ESP8266 as follows:

You will need just 2 wires for power and 1 line for data.

ESP8266	GPS module
3v3	"Vin" or "Vcc"
GND	GND
D5	Tx

The GPS module works best with a clear view of the sky. Try to position your GPS unit near a window. The module is a receiver on the 1.5 GHz band, listening for communications from GPS satellites orbiting the earth. Each satellite transmits its location and timestamp. If at least four satellite signals are received, the module can compute its own location with very good accuracy. In our application, we will ignore the location data but use the timestamp.

Let's write some software to test the serial connection.

Here is a simple sketch to see the data. Download the Step1 sketch from GitHub, reproduced here:

```
#include <SoftwareSerial.h>

SoftwareSerial gpsSerial(D5,D6);              // set up UART for GPS data

void setup() {
  gpsSerial.begin(9600);                      // GPS Data @ 9600 baud
```

```
  Serial.begin(115200);                          // Serial monitor @ 115200 baud
}

void loop() {
  if (gpsSerial.available()) {                   // any data from GPS?
    char c = gpsSerial.read();                   // yes, so get the next character
    Serial.print(c);                             // and echo it to serial monitor
  }
}
```

The first line adds a built-in Arduino library for serial data communication (ESP8266 has a hardware serial interface but it used by the board itself). The next line specifies on which pins data should be received (pin D5) and transmitted (D6). Note that we are only receiving data in this example, so D6 is not used. Feel free to connect D6 the GPS "Rx" pin if you like.

The setup() runs only at the beginning of the sketch. Here we set the data transmission rates. The GPS module transmits asynchronous serial data on its "Tx" line at a rate of 9600 baud. (Some older modules use 4800, and some newer models you 38600, so you should check the specs of your module if you are using different hardware.)

In loop(), the sketch polls the GPS serial Tx pin for character data. The highlighted line above does all the work. When a character is available, this line reads the character into variable 'c'. The following line just echoes, or 'prints' that character to the serial output monitor.

Upload the sketch and run it. Then open the serial monitor and set the baud rate to 115200. If everything is configured correctly, you will see a flood of data from the GPS module. What does it all mean?

STEP 2: PARSING THE GPS DATA

The GPS data is in NMEA format. Each line of output corresponds to a NMEA sentence, and each sentence contains multiple data elements separated by commas. Here is an example for one type of NMEA sentence:

"$GPGGA,033757.000,3942.9046,N,08410.5099,W,2,8,1.05,311.0,M,-33.4,M,0000,0000*61"

You can write your own parser to extract the data, but ready-made libraries are available and do the job quite nicely. I use Mikal Hart's "tinyGPS++", available at https://github.com/mikalhart/TinyGPSPlus. Let's modify the code, using this library to extract the time data. Except for addition of the library its associated gps object, the code starts the same:

```
#include <SoftwareSerial.h>
#include <TinyGPS++.h>

SoftwareSerial gpsSerial(D5,D7);
TinyGPSPlus gps;

void setup() {
  gpsSerial.begin(9600);                         // GPS Data @ 9600 baud
  Serial.begin(115200);                          // Serial monitor @ 115200 baud
}
```

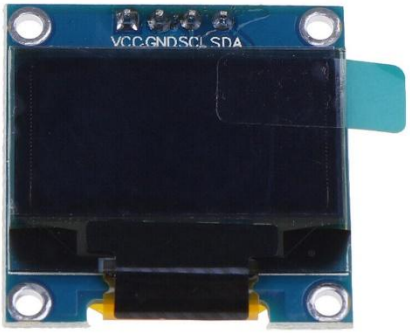
```
void loop() {
  if (gpsSerial.available()) {                // any data from GPS?
    char c = gpsSerial.read();                // yes, so get the next character
    if (gps.encode(c)) printTime();           // got NMEA, so print the time.
  }
}
```

Notice gps.encode() on the last line. This accepts the GPS data, character by character, and it returns true whenever a full line of data has been received and decoded. The only thing left to do is to print the time:

```
void printTime() {
  int h  = gps.time.hour();                   // get the hour
  if (h<10) Serial.print("0");                // make it 2 digits: '6'->'06'
  Serial.print(h); Serial.print(":");         // print the hour
  int m = gps.time.minute();                  // get the minutes
  if (m<10) Serial.print("0");                // make it 2 digits: '6'->'06'
  Serial.print(m); Serial.print(":");         // print the minutes
  int s = gps.time.second();                  // get the seconds
  if (s<10) Serial.print("0");                // make it 2 digits: '6'->'06'
  Serial.print(s); Serial.println(" UTC");    // print the seconds
}
```

Download Step2.ino. Run the sketch and you will see the current UTC time. The time is printed at the completion of each NMEA data string, which happens several times per second. Position the GPS unit near a window, if possible. It might take several minutes for your GPS unit to acquire enough satellites.

STEP 3: ADD AN OLED DISPLAY



I've made several different clocks. The clock I built for my ham radio shack uses large seven-segment displays. My favorite clocks use medium-sized LCDs. But for this project I am using a tiny 0.96" OLED display. Why suffer such a small display? Those of us of a certain age appreciate larger, bolder numbers.

This project uses an OLED display for several reasons:

1. Though small, these monochrome displays provide excellent readability and contrast – even in daylight.
2. OLED displays require only two I/O pins. The ESP8266 has few unrestricted I/O pins, making this display easier to integrate than others.
3. OLED displays are inexpensive.
4. Displaying time, while important, is not the primary function of this project. We are providing time information to networked devices – not human eyes.

Without going into a lot of detail, the ESP8266 places many restrictions on how its I/O pins can be used. If we added an LCD display using the SPI bus, it would be difficult to accommodate the other hardware (GPS input, RTC backup, etc).

This OLED display has four pins. Connect its clock pin (SCL or SCK) to ESP8266 "D1" and its serial data pin (SDA) to the ESP8266 "D2" pin. Also connect 3.3v power (3v3) and ground.

ESP8266	OLED module
3v3	VCC
GND	GND
D1	SCL
D2	SDA

This display will require a library to drive it. I use the Adafruit SSD1306 library. If you don't already have it installed, go to Tools > Manage Libraries. Search for and install the library named "Adafruit SSD1306".

Test the display and driver with only 4 lines of code, highlighted below. From Step3.ino:

```
#include <SoftwareSerial.h>
#include <TinyGPS++.h>
#include <Adafruit_SSD1306.h>                  // Adafruit OLED library

SoftwareSerial gpsSerial(D5,D7);
TinyGPSPlus gps;
Adafruit_SSD1306 led(128,64,&Wire);            // OLED display object

void setup() {
  gpsSerial.begin(9600);                       // GPS Data @ 9600 baud
  Serial.begin(115200);                        // Serial monitor @ 115200 baud
  led.begin(SSD1306_SWITCHCAPVCC,0x3C);        // turn on OLED display
  led.display();                               // and display Adafruit logo
}

void loop() {
… more code …
```

These four lines a) include the library, b) create a properly-sized OLED object, c) initialize the display, and finally d) display the Adafruit logo. Which reminds me: Adafruit spends a lot of time on supporting their products, so please consider buying from them.

STEP 4: DISPLAY THE TIME

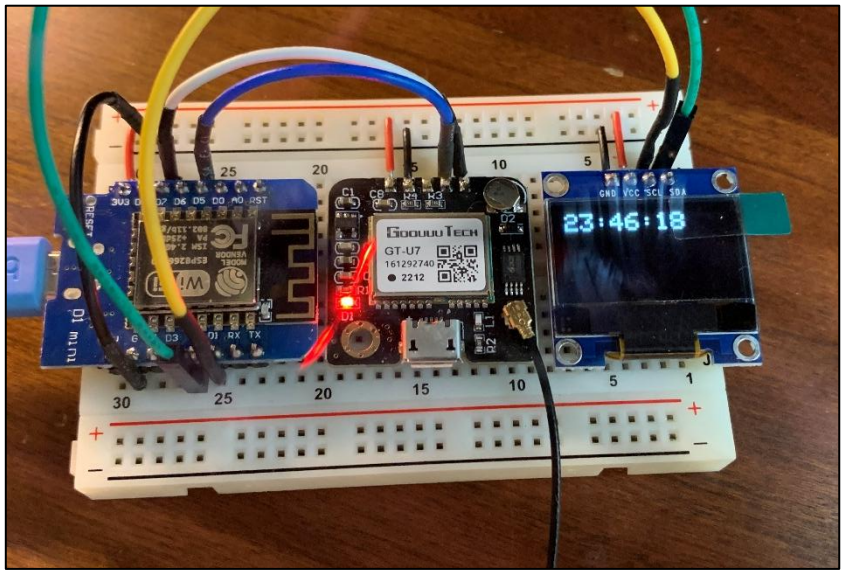
We have a working display, and we have time information from the GPS. Put the two together so that we show the time on our OLED display. Look at the printTime() routine in step 2 above, and consider changing it so that the time output goes to the OLED display instead of the serial monitor. Here is the code:

```
void displayTime() {
  led.clearDisplay();                          // erase previous screen
  led.setCursor(0,0);                          // start at top-left
  int h  = gps.time.hour();                    // get the hour
  if (h<10) led.print("0");                    // make it 2 digits: '6'->'06'
  led.print(h); led.print(":");                // print the hour
  int m = gps.time.minute();                   // get the minutes
  if (m<10) led.print("0");                    // make it 2 digits: '6'->'06'
  led.print(m); led.print(":");                // print the minutes
  int s = gps.time.second();                   // get the seconds
  if (s<10) led.print("0");                    // make it 2 digits: '6'->'06'
  led.print(s);                                // print the seconds
  led.display();                               // show the result on screen
}
```

The only differences, besides the name of the procedure, are the three highlighted lines above. In addition, we need additional setup() code to specify the color and size of our text:

```
led.setTextColor(WHITE);                    // white on black text
led.setTextSize(2);                         // size 2 is easily readable
```

Get Step4.ino from GitHub and upload it to your ESP8266. We have a working GPS clock using only 35 lines of code.



The author's breadboard for Steps 4-8, containing (left to right): Wemos D1 mini, GPS module, and OLED display.

This breadboard contains all the hardware you need for a full-fledged NTP server.

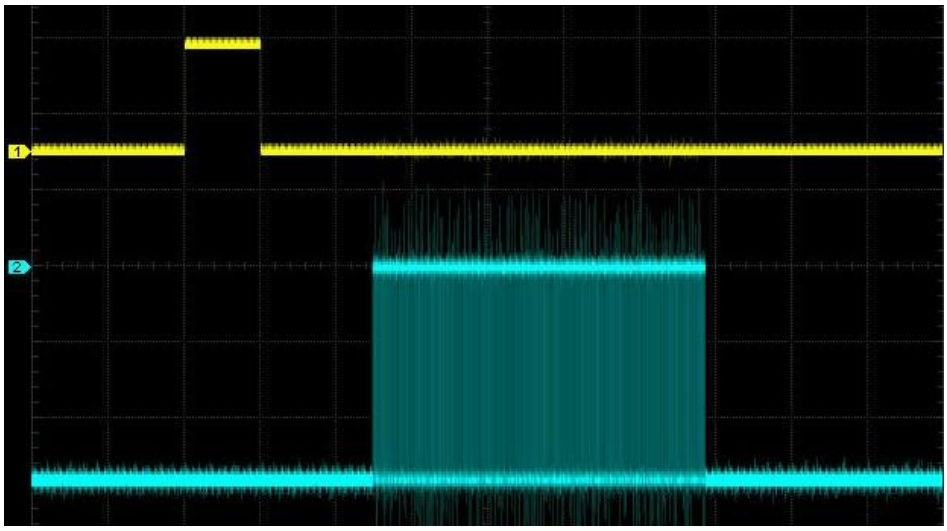
STEP 5: SYNCHRONIZING WITH GPS

You may notice something odd about the displayed time. Depending on where you live, it may be one or more hours fast (or slow). This is because GPS time does not know what time zone you are in – it reports time in UTC (Coordinated Universal Time).

In addition, carefully compare your clock with an Internet time website, like timeanddate.com, time.is, or time.gov. You may notice that the OLED clock is *always a fraction of second slow* compared to the online clock. Why is that? From one of my earlier GPS clock articles:

It takes time to send (and read) GPS data. Each NMEA sentence sent from the GPS module at 9600 baud takes about 50-100 milliseconds. Modules vary in terms of which sentences they send. My Adafruit module sends 4 sentences every second, on average, with the whole packet 200-400 milliseconds in duration. By the time the data is received and decoded, it is already old! Even the fastest microcontroller and display are at the mercy of this delay. The clock is always a few hundred microseconds slow. No big deal, perhaps, but it is enough delay to be visible.

Fortunately, there is a better solution. Most modules also provide a 1pps (1 pulse-per-second) output. The leading edge of this pulse typically falls at the top of the second. In other words, the start of the pulse corresponds to the start of the second. Serial RS-232 data immediately following this pulse will give the time corresponding to that pulse. The digital oscilloscope image shows the relationship between the 1pps pulse in yellow and the serial data in blue.



The rising edge of the 1pps signal marks the beginning of the second. Each square in the horizontal direction equals 100 mS, or one tenth of a second. In the example above, the 1pps signal is 100 mS wide. Serial data begins about 250 mS after the start of the second and lasts for roughly 425 mS. After watching this display for a minute or so, it was interesting to see that the start and duration of the serial data are both variable.

Here is the basic algorithm:

1. Continuously read the GPS data and decode it, as before.
2. When a GPS 1pps signal is received, display the time one second ahead.

The most interesting part is *adding one second* to the time. What? We need to advance the clock one second because a 1pps signal indicates the start of a *new* second. In other words, the available time data when the pulse occurs is always one second *behind the actual time*. So, add a second.

To prove it, let's create a clock that displays the current time whenever the GPS 1pps signal arrives. We will handle this signal with an interrupt, which requires a few special coding elements. First, we create an interrupt handler that runs whenever the pulse occurs. This handler does only one thing: set a flag that we can act on later. Here is the code for the flag and handler:

```
volatile int syncFlag = 0;

IRAM_ATTR void ppsHandler() {                  // 1pps interrupt handler
  syncFlag = 1;                                // flag the need to sync
}
```

The keywords *volatile* and *IRAM_ATTR* indicate that the code and data must be handled in a special way. Next, designate one of the ESP8266 pins as an interrupt pin. This requires a special statement in the setup() routine:

```
  attachInterrupt(digitalPinToInterrupt(       // enable 1pps GPS time sync
    D7), ppsHandler, RISING);
```

Notice that it designates D7 as the interrupt pin and specifies the routine to call whenever a RISING pulse is detected. The final change is to modify the program loop so that we display the time whenever the syncFlag has been set:

```
void loop() {
  if (gpsSerial.available()) {                // any data from GPS?
    char c = gpsSerial.read();                // yes, so get the next character
    gps.encode(c);                            // and feed it to GPS parser
  }
  if (syncFlag) {                             // 1pps signal received from GPS?
    syncFlag = 0;                             // yes, so reset the flag
    displayTime();                            // and display the time
  }
}
```

The first part of the loop is unchanged. The highlighted section is new. It checks for the syncFlag and calls displayTime() whenever the flag is raised. We also must reset the flag for the next 1pps pulse.

This code displays time within a few milliseconds of each 1pps pulse. Try it! Download Step5.ino from GitHub, run it, and compare the time to an Internet clock. Time on the OLED display should update simultaneously with the Internet clock. But is it the correct time, or one second behind?

In the Step5 source code, I suggest a few lines in displayTime() that add one second to the time. They are commented out by default. Uncomment these lines to "add the missing second".

WRAP UP

That's it for Part 1. We created a GPS-based clock with millisecond accuracy. Our NTP server does not require any additional hardware –the rest of the project is just SMOP, or a "simple matter of programming." Read Part 2 to learn about NTP and how to turn this clock into an NTP timeserver.

Last updated: April 23, 2026

Project Resources

- Part 1 (this document)
- Part 2: NTP Software
- Part 3: Builder's Guide
- Schematic
- Interactive BOM
- Source Code
- PCB Gerbers
- Enclosure STL files
- Operating Guide