

NTP clock

Build an NTP-based clock using a microcontroller and LCD display.

Bruce E. Hall, [W8BH](#)



Introduction.

I recently made a WWVB clock, which is a lot of fun. As soon as I posted it, someone asked if I had a GPS clock. So I did that, too. It didn't take long before another inquired "Do you have code for an NTP-based clock?"

This article describes a clock that derives its time from NTP. For my clock, I chose an ESP32 microcontroller module and a 2.8" 320x240 pixel LCD display. No other components are needed. I assume that the reader is comfortable with basic breadboarding and C programming. I am using the Arduino IDE, but the algorithms here can be used in almost any programming environment. Keep reading for a step-by-step description of the clock and how to build it.

What is NTP?

NTP stands for [Network Time Protocol](#). It is an Internet protocol used to synchronize computer clocks to a time reference. It was originally developed by Prof. David Mills at the University of Delaware, and is one of the oldest Internet protocols in current use. Its current implementation is described by draft protocol [RFC5905](#). By using NTP, devices are synchronized over the Internet within tens of milliseconds of [UTC](#) (Coordinated Universal Time).

The Expressif (ESP32 and ESP8266) modules are ideally suited for an NTP clock, since they have the ability to connect to the Internet via built-in WiFi. To get the time, all we need to do is establish an Internet connection and request the current time from an [NTP time server](#).

Project Files

[Part 1 \(this document\)](#)
[Part 2: Builder's Guide](#)
[Source Code](#)
[PCB Gerbers](#)
[Schematic](#)
[Enclosure STL files](#)

STEP 1: CONFIGURING YOUR MICROCONTROLLER



There are many different ESP32 modules to choose from, each with different board layouts and configurations. For this project I am using the 38-pin [ESP32-WROOM-32 Board by HiLetGo](#), available on Amazon for \$11. If you use a different module, pay close attention to the pinouts and your wiring.

Steps 1-3 assume you are using an ESP32. For a smaller and less expensive module, you may also use an ESP8266 module called the “D1 Mini”. To configure an ESP8266, refer to steps 1A-3A in the appendix.



Wemos D1 Mini

You should be comfortable with the Arduino IDE and know how to program a microcontroller. You should also have a suitable breadboard and 5V power supply.

Here is a quick run-down of using the ESP32 in the Arduino IDE:

1. Copy the following URL into your Arduino Boards Manager list.

https://dl.espressif.com/dl/package_esp32_index.json

2. Next, configure the IDE. I am currently using IDE version 1.8.13.
 - a) Choose Tools-> Board -> ESP32 Boards -> ESP32 Dev Module
 - b) Tools -> Upload Speed -> 921600
 - c) Tools -> Port -> (choose the computer port attached to the ESP32)

For more information on the ESP32, including an installation [tutorial](#), visit [Randomnerdtutorials.com](#). On my GitHub account I include a “step1.ino” blink sketch which you may use to confirm your setup.

STEP 2: CONNECT THE DISPLAY

The display is a 320x240 pixel TFT LCD on a carrier board, using the ILI9341 driver and an SPI interface. Search eBay and Google for “2.2 ILI9341” and you will find many vendors. The current price for the red Chinese no-brands, shown at right, is \$6-7 depending on shipping. I use the 2.8” version which costs a few dollars more.



My display has 9 pins, already attached to headers, for the LCD and an additional row of 5 holes without headers for the SD card socket. Our project will use the 9 pins with headers.

There are 4 pins on the display that connect to the microcontroller, and 4 pins that are power/ground related. The following table details the connections. GPIO numbers are also listed in case you are using a different ESP32 module.

Display Pin	Display Label	Connects To ESP32:	Function
1	Vcc	"5V"	Power
2	Gnd	"Gnd"	Ground
3	CS	GPIO 5	Chip Select
4	RST	"3.3V"	Display Reset
5	DC	GPIO 21	Data/Cmd Line
6	MOSI	GPIO 23	SPI Data
7	SCK	GPIO 18	SPI Clock
8	LED	"5V"	LED Backlight Power
9	MISO	n/c	SPI Data

Connect the wires and apply power via the 5V/Gnd pins or via the USB port. Make sure the backlight is ON – if not, immediately disconnect and check your wiring. The most common failure at this point is improper wiring. Please note that the display RST pin is not 5V tolerant, therefore connect it to the ESP32 3.3V pin, or to 5V through a 10K resistor.

STEP 3: INSTALL THE DISPLAY SOFTWARE

For TFT support I am using the "TFT_eSPI" library by Bodmer, version 2.2.14. To install it, go to the Arduino library manager (Sketch->Include Libraries->Manage Libraries), search for "TFT_eSPI", and install. You can also find the latest code on GitHub at https://github.com/Bodmer/TFT_eSPI

Once the TFT Library is installed, you will need to configure it by modifying the User_Setup.h file in your TFT_eSPI library directory. I prefer setting the configuration in my sketch, rather than modifying a file, but this is not a choice. Edit your User_Setup.h file to include *only* the following DEFINES:

```
#define ILI9341_DRIVER
#define TFT_MOSI 23
#define TFT_MISO 19
#define TFT_SCLK 18
#define TOUCH_CS 22
#define TFT_CS 5
#define TFT_DC 21
#define TFT_RST -1
#define LOAD_GLCD
#define LOAD_FONT2
#define LOAD_FONT4
#define LOAD_FONT6
#define LOAD_FONT7
#define LOAD_FONT8
#define LOAD_GFXFF
#define SPI_FREQUENCY 4000000
#define SPI_READ_FREQUENCY 2000000
#define SPI_TOUCH_FREQUENCY 2500000
```


Displaying the current UTC time is a single line. Here is how to display time on the serial monitor:

```
Serial.println(UTC.dateTime()); // display UTC time
```

The `dateTime()` routine is very powerful: it can display time in any of several standard formats, such as [ISO8601](#), or display any combination of time elements that you specify. All of the possible elements are described on the [ezTime GitHub page](#). For example, to display the time in `hh:mm:ss` format, you would use `dateTime("h:i:s")`. To display the date in `dd/mm/yyyy` format, you would use `dateTime("d/m/Y")`. Suppose you don't want leading zeros; or you want a two-digit year; or the day of the week (in English, short or long format); or the day of the year; or `am/pm`; or `AM/PM`. It is all there.

NTP returns the time in UTC, and knows nothing about what time zone you are in. And it knows nothing about daylight savings time. In previous articles I describe how to [code DST yourself](#), and how to save time and effort using a dedicated library. Fortunately, `ezTime` contains time zone and DST support. No other libraries are needed.

`ezTime` handles time zones according to the POSIX format, described [here](#). Every time zone is described by a string, which includes the name. The time zone string for my location, in the Eastern US time zone is: `"EST5EDT,M3.2.0/2:00:00,M11.1.0/2:00:00"`. The following tables breaks it down.

To create a local time zone in `ezTime`, first establish the object like this:

```
Timezone local;
```

Then set its Posix string:

```
local.setPosix(TZ_RULE);
```

`ezTime` alternatively supports the "Olson format", which is easier to read. For example, instead of the above string I could simply say:

```
local.setLocation("US/Eastern");
```

The Olson method is simpler, and doesn't require you to find or construct the time string for your zone. However, it does require your code to do an additional internet lookup. It also assumes that the Olson server is running and providing correct time zone information.

Element	Meaning
EST	Name of time zone in standard time (EST = Eastern Standard Time in this case.)
5	Hours offset from UTC, meaning add 5 from this time to get to UTC. It can also specify minutes, like <code>-05:30</code> for India.
EDT	Name of time zone in Daylight Saving Time (DST), EDT stands for Eastern Daylight Time
,M3	DST starts in March
.2	On the second occurrence of
.0	a Sunday
/2:00:00	at 02:00 local time
,M10	DST ends in November
.2	on the second occurrence of
.0	a Sunday
/2:00:00	at 02:00 local time

Putting it all together, the following sketch will get NTP time and display it in my local timezone:

```
#include <TFT_eSPI.h> // https://github.com/Bodmer/TFT_eSPI
#include <ezTime.h> // https://github.com/ropg/ezTime
#include <WiFi.h>

#define WIFI_SSID "yourWifiName"
#define WIFI_PWD "yourWiFiPassword"
#define TZ_RULE "EST5EDT,M3.2.0/2:00:00,M11.1.0/2:00:00"

TFT_eSPI tft = TFT_eSPI(); // display object
Timezone local; // local timezone variable
```

```

void setup() {
  tft.init();
  tft.setRotation(1); // portrait screen orientation
  tft.fillScreen(TFT_BLACK); // start with empty screen
  WiFi.begin(WIFI_SSID,WIFI_PWD); // attempt WiFi connection
  waitForSync(); // wait for NTP packet return
  local.setPosix(TZ_RULE); // estab. local TZ by rule
}

void loop() {
  events(); // refresh time every 30 min
  if (secondChanged()) // is it a new second yet?
    tft.drawString(local.dateTime("h:i:s"), // display time on screen
      50,50,4);
}

```

This is a working clock that can display UTC or local time, accurate to within a few tens of milliseconds. ezTime refreshes the time via NTP about every 30 minutes (1801 seconds, to be precise). Best of all, you have access to all of the usual [Arduino time variables](#). For instance, now() returns Unix time, and hour() returns the current UTC hour. The rest of this project is just window dressing; you can apply this step 4 code to whatever type of project or display you like. I am using the same display layout as I did for my GPS clock. I am not using any of ezTime's powerful formatting features, in order to keep the code base consistent and easier to maintain across my clock projects.

STEP 5: A VFD DISPLAY WOULD BE NICE

If you were around in the 1980's, you might remember clocks with glowing blue vacuum fluorescent displays, like this one. They are bright enough to read in daylight and are dimmable for nighttime use. I had one in my bedroom and I loved it.



We will mimic this look by using a similar color and seven-segment font (GFX font 7). Add a DEFINE at the top of the sketch make it easy to change the color later.

```
#define TIMECOLOR TFT_CYAN
```

Here is a routine to display the time in global variable t:

```

void displayTime() {
  int x=10, y=50, f=7; // screen position & font
  tft.setTextColor(TIMECOLOR, TFT_BLACK); // set time color
  int h=hour(t); int m=minute(t); int s=second(t); // get hours, minutes, and seconds
  if (h<10) x+= tft.drawChar('0',x,y,f); // leading zero for hours
  x+= tft.drawNumber(h,x,y,f); // hours
  x+= tft.drawChar(':',x,y,f); // hour:min separator
  if (m<10) x+= tft.drawChar('0',x,y,f); // leading zero for minutes
  x+= tft.drawNumber(m,x,y,f); // show minutes
  x+= tft.drawChar(':',x,y,f); // show ":"
  if (s<10) x+= tft.drawChar('0',x,y,f); // add leading zero if needed
  x+= tft.drawNumber(s,x,y,f); // show seconds
}

```

The highlighted lines check the hour, minute, and second values. If any is less than 10 (and therefore only a single digit), a zero is displayed in front of number. Time now always displayed in the form HH:MM:SS. Because the number of digits is constant, the previous time does not need to be erased. And the display flicker caused by this erasure is eliminated. The clock is starting to look nice, isn't it?

Now let's add the date. ezTime gives us access to the day, month, and even the day of the week. Displaying them is similar to displaying the time:

```
void displayDate() {
  int x=50,y=130,f=4; // screen position & font
  const char* days[] = {"Sunday","Monday","Tuesday",
    "Wednesday","Thursday","Friday","Saturday"};
  tft.setTextColor(DATECOLOR, TFT_BLACK);
  tft.fillRect(x,y,265,26,TFT_BLACK); // erase previous date
  x+=tft.drawString(days[weekday()-1],x,y,f); // show day of week
  x+=tft.drawString(", ",x,y,f); // and
  x+=tft.drawNumber(month(),x,y,f); // show date as month/day/year
  x+=tft.drawChar('/',x,y,f);
  x+=tft.drawNumber(day(),x,y,f);
  x+=tft.drawChar('/',x,y,f);
  x+=tft.drawNumber(year(),x,y,f);
}
```

The highlighted lines show how to display the day of the week. A constant array is used to hold strings for each day of the week. The library function, weekday(), returns a value 1 through 8, corresponding Sunday through Saturday. We need a value of 0 through 7, since Arduino arrays are 0-based, so subtract 1: days[weekday()-1] returns the correct string.

Finally, we want to update the date display when the date changes. To do this, modify the updateDisplay() routine so that, if the time changes, look for a date change, too:

```
void updateDisplay() {
  if (t!=now()) { // is it a new second yet?
    displayTime(); // and display it
    if (day(t)!=day()) // did date change?
      displayDate(); // yes, so display it
    t=now(); // Remember current time
  }
}
```

The Step 5 clock displays UTC time and date. A screen border is also added to enhance the display.

STEP 6: LOCAL TIME

UTC time is great for your ham shack, or if you happen to live in Liverpool, but even Liverpool residents have Summer Time at UTC plus 1 hour. Now let's display local time, taking daylight saving time into consideration.

To do this, add a global variable, lt, to hold the most recent local time. Then modify the updateDisplay routine so that we compare the current local time with the most recently displayed one.

```
void updateDisplay() {
  if (t!=now()) { // is it a new second yet?
    time_t newLt = local.now(); // get local time
    displayTime(newLt); // and display it
    if (day(newLt)!=day(lt)) // did date change?
      displayDate(newLt); // yes, so display it
    t=now(); // remember current UTC
    lt = newLt; // remember current local time
  }
}
```

```

}
}

```

The clock now accurately displays the time and date according to local time zone.

Up until now, we have displayed the time in 24-hour format. But local time is usually expressed in 12-hour format (13:00 is referred to as 1:00). Add a define at the top of the sketch to give us this option:

```

#define USE_12HR_FORMAT true // preferred format for local time

```

Then code the option in the displayTime() routine as follows:

```

int h=hour(t); int m=minute(t); int s=second(t); // get hours, minutes, and seconds
if (USE_12HR_FORMAT) { // adjust hours for 12 vs 24hr format:
  if (h==0) h=12; // 00:00 becomes 12:00
  if (h>12) h-=12; // 13:00 becomes 01:00
}

```

Do you prefer “05:00” or “5:00”? Let’s add an option to suppress the leading zero in the hour field. Add a define for this first, then modify the displayTime() routine. The trick to suppressing the leading zero is to print an “8” (which uses all 7 segments) in the background color, effectively erasing any digit that was there before:

```

if (h<10) { // is hour a single digit?
  if (!(USE_12HR_FORMAT)|| (LEADING_ZERO)) // 24hr format: always use leading 0
    x+= tft.drawChar('0',x,y,f); // show leading zero for hours
  else {
    tft.setTextColor(TFT_BLACK,TFT_BLACK); // black on black text
    x+=tft.drawChar('8',x,y,f); // will erase the old digit
    tft.setTextColor(TIMECOLOR,TFT_BLACK);
  }
}

```

The Step 6 clock presents local time and date, with automatic DST adjustment, in 12-hour format.

STEP 7: CLOCK STATUS

I put the clock aside one morning, then came back later in the day to check it. It looked fine, and the time was right, but I wondered: “How current is the data? Is the clock synchronizing with NTP every half-hour?” A status indicator would be nice. ezTime provides a function, lastNtpUpdateTime(), which returns the time of the last update. To get the elapsed time since the update, just subtract the result of this function from the current time. I am using a color-coded status indicator to visually show how stale the clock data is. For example, green means synchronization within the last hour, orange for synchronization with the last 24 hours, and red for anything more than a day:

```

#define SYNC_MARGINAL 3600 // orange status if no sync for 1 hour
#define SYNC_LOST 86400 // red status if no sync for 1 day

void showClockStatus() {
  const int x=290,y=1,w=28,h=29,f=2; // screen position & size
  int color;
  if (second()%10) return; // update every 10 seconds
  int syncAge = now()-lastNtpUpdateTime(); // how long since last sync?
  if (syncAge < SYNC_MARGINAL) // time is good & in sync
    color = TFT_GREEN;
  else if (syncAge < SYNC_LOST) // sync is 1-24 hours old
    color = TFT_ORANGE;
  else color = TFT_RED; // time is stale!
  tft.fillRoundRect(x,y,w,h,10,color); // show clock status as a color
}

```

Another helpful thing to know is the strength of your Wi-Fi signal. The WiFi object contains a function that returns the RSSI (“received signal strength indicator”), which is an estimate, in [dBm](#), of the received signal. For Wi-Fi, usable values range between -30 (maximum signal) to -80 (unreliable). -50 to -70 are typical, usable signal strengths. We can show it with only two extra lines of code:

```
tft.setTextColor(TFT_BLACK,color);  
tft.drawNumber(-WiFi.RSSI(),x+8,y+6,f);           // signal strength as positive value
```

The Step 7 clock is a full-featured clock with an NTP status indicator.

STEP 8: DUAL DISPLAY

The previous step created a perfectly usable clock. There is even a bit of unused space at the bottom of the display which you can use for own modifications, such as weather data, sunspot data, clock temperature, sunrise/sunset details, moon phase, etc. The ESP32, with its build-in Wi-Fi, is well suited for these applications and more.

I decided to use that extra space for a second time display. As a ham, it is important to know local time *and* UTC. And it is especially important to know the date for each, since here at W8BH they are often different. I will use the same format as I did for the [GPS clock](#): local time/date on top, and UTC time/date on bottom. Let’s get started.

First, to create enough screen space for both clocks, we can shrink the date so that it fits beside the time display, instead of underneath it. So the displayDate() method will need to be rewritten. Also, since we are going to use it in two different places on the screen, we will need to pass it the screen coordinates of where it should be displayed. Consider the following routine:

```
void showDate(time_t t, int x, int y) {  
    const int f=4,yspacing=30;           // screen font, spacing  
    const char* months[] = {"JAN","FEB","MAR",  
        "APR","MAY","JUN","JUL","AUG","SEP","OCT",  
        "NOV","DEC"};  
    tft.setTextColor(DATECOLOR,TFT_BLACK);  
    int m=month(t), d=day(t);           // get date components  
    tft.fillRect(x,y,50,60,TFT_BLACK); // erase previous date  
    tft.drawString(months[m-1],x,y,f); // show month on top  
    y += yspacing;                     // put day below month  
    if (d<10) x+=tft.drawNumber(0,x,y,f); // draw leading zero for day  
    tft.drawNumber(d,x,y,f);           // draw date  
}
```

The two important lines in this routine are highlighted: a drawString() to show the month, and a second drawString() to show to date. The routine is passed 3 variables: the time (which includes the date information) and the x,y screen coordinates. Months[] contains strings for the months. Astute readers will realize that ezTime already includes these strings, but I am not using them, to keep the code consistent with my other projects.

We will need a new routine to show the time zone. ezTime provides a function, getTimezoneName(), that returns the name of the time zone. The routine below shows how to display the local zone name,

with the important line highlighted. All the other lines are just for screen formatting. Notice that the zone name for UTC is just “UTC”, but you could call `UTC.getTimezoneName()` if you wanted.

```
void showTimeZone (int x, int y) {
    const int f=4; // text font
    tft.setTextColor(LABEL_FGCOLOR,LABEL_BGCOLOR); // set text colors
    tft.fillRect(x,y,80,28,LABEL_BGCOLOR); // erase previous TZ
    if (!useLocalTime)
        tft.drawString("UTC",x,y,f); // UTC time
    else
        tft.drawString(local.getTimezoneName(),x,y,f); // show local time zone
}
```

For both clocks, whenever we update the time, we must also check to see if the date or DST status changes. For convenience, let’s combine these functions in a “showTimeDate” routine:

```
void showTimeDate(time_t t, time_t oldT, bool hr12, int x, int y) {
    showTime(t,hr12,x,y); // display time HH:MM:SS
    if ((!oldT) || (hour(t)!=hour(oldT))) // did hour change?
        showTimeZone(x,y-42); // update time zone
    if (day(t)!=day(oldT)) // did date change?
        showDate(t,x+250,y); // update date
}
```

This routine is called at the top of each new second, and “showTime()” is called to update the time. But we don’t want to rewrite the date every second: this would cause unnecessary screen flicker as the old date is erased and the new date is written. The only time it needs to be updated is when the new day starts. Look at the highlighted line above. `day(t)` will return the day for the latest time, and `day(oldT)` will return the day for the previously displayed time. If they differ, it must be the first second of the new day – the perfect time to update the date display. Updating the time zone is a bit trickier, since that can happen at almost any time. I am taking a shortcut, and update it hourly. A more elegant method might be to pass the `timezone` object and check to see if the `local.isDST()` flag changed. It’s up to you.

Lastly, we might want the local and UTC clocks to use a different format. For instance, I like having my local clock in 12-hour mode, and my UTC clock in 24-hr mode. The following two defines will allow us to configure each one independently:

```
#define LOCAL_FORMAT_12HR true // local time format
#define UTC_FORMAT_12HR false // UTC time format
```

With all of above changes, circle back to the `updateDisplay` routine. Displaying a single clock just got simpler, with a single call to `showTimeDate()`:

```
void updateDisplay() {
    t = now(); // check latest time
    if (t!=oldT) { // are we in a new second yet?
        lt = local.now(); // keep local time current
        useLocalTime = true; // use local timezone
        showTimeDate(lt,oldLt,LOCAL_FORMAT_12HR,10,46); // show new local time
        showClockStatus(); // and clock status
        oldT=t; oldLt=lt; // remember currently displayed time
    }
}
```

To add a second clock, we only need two more lines in our `updateDisplay` routine:

```
void updateDisplay() {
    t = now(); // check latest time
    if (t!=oldT) { // are we in a new second yet?
        lt = local.now(); // keep local time current
        useLocalTime = true; // use local timezone
```

```

showTimeDate(lt,oldLt,LOCAL_FORMAT_12HR,10,46); // show new local time
useLocalTime = false; // use UTC timezone
showTimeDate(t,oldT,UTC_FORMAT_12HR,10,172); // show new UTC time
showClockStatus(); // and clock status
oldT=t; oldLt=lt; // remember currently displayed time
}
}

```

2026 Refresh: OTA Wi-Fi credentials

Four years ago, I put this project to bed with the following features:

- A startup screen that shows the progress of the Wifi and NTP connection
- A Wi-Fi status monitor, which will automatically reconnect if the Wi-Fi connection is lost.
- An option for displaying AM/PM in 12-hour mode.
- An option for sending debug information to the serial output port.
- An option for selecting the NTP server
- An option for sending the time, each second, to the serial output port. The timestamp format is completely customizable.

I keep a version of this clock on my bench. But I was recently reminded that time moves on: the Arduino IDE has evolved, software libraries have been updated, and our needs have changed. For instance, who hard-codes Wi-Fi credentials anymore? It's far more practical to let the end-users update them – without recompiling. In 2026 we can update Wi-Fi credentials using our mobile phones. It's time to add this must-have feature.

For any project using multiple libraries, it is important to keep track of version numbers. An older version of one library, for instance, may not be compatible with a new version of another library. I am using the Arduino IDE version 2.3.8 with the following software: Expressif ESP32 board 2.0.17, TFT_eSPI 2.5.43, and ezTime 0.8.3. Expressif offers a more recent ESP32 board package, 3.3.7, but there are issues: a) it requires significantly more FLASH memory and b) it has minor incompatibilities with the other libraries. So, I am using the 2.x board package for now. This is not an issue if you are using the ESP8266 microcontroller in your clock.

The critical library for OTA updates is the WiFi Manager by tzapu, hosted on [GitHub](#). It gives our project a Wi-Fi connection manager via mobile phone user interface: connect your phone to the “NTP clock” Wi-Fi network, open your phone’s web browser, and set your clock’s Wi-Fi credentials. These credentials are saved in the microcontroller’s FLASH memory and do not have to be reentered.

I added a normally-open pushbutton switch to force an update of the WiFi credentials. Pressing the switch on startup momentarily connects a microcontroller pin to ground, causing the software to erase the current credentials:

```

if (switchPressed()) // reset switch is pressed?
  wm.resetSettings(); // yes, erase WiFi credentials

```

The switchPressed() routine looks at the logic state of the pin and inverts it, so that a grounded pin (logic 0) returns true (1=pressed) and an ungrounded pin (logic 1) returns false (0=unpressed):

```

bool switchPressed() {
  return !digitalRead(SWITCH_PIN); // switch active low (gnd)
}

```

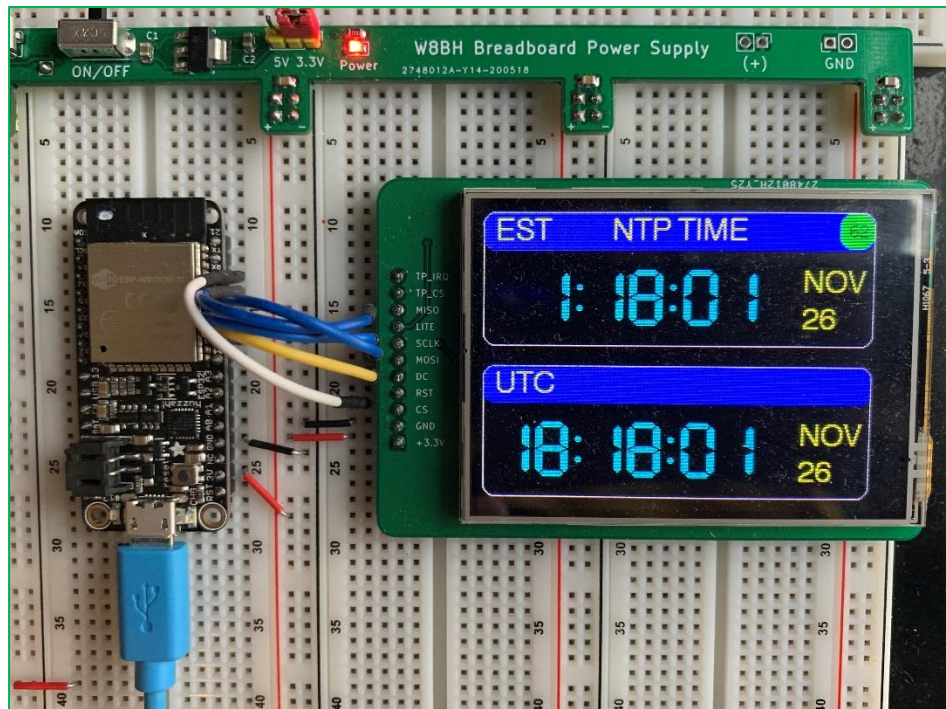
The pin connected to the pushbutton is assigned in the code by a #define statement. I used pin D1 in the ESP8266 Wemos D1 mini and pin GPIO22 in the ESP32. Note that we can use the same code for both, asking the compiler to choose based on the target microcontroller:

```
#if defined(ESP32) // The WiFi reset button:
#define SWITCH_PIN 22 // .. is GPIO 22 on ESP32
#elif defined(ESP8266)
#define SWITCH_PIN D1 // .. is D1 on ESP8266
#endif
```

See [Part 2](#) for a builder's guide to the completed project.

The source code for each step, and the final version, is on my [GitHub account](#). Drop me a line if you build your own NTP clock!

73,
Bruce.



Last updated: 13 Mar 2026

APPENDIX A: Using an ESP8266 in place of the ESP32

STEP A1: CONFIGURING YOUR MICROCONTROLLER

For a smaller and less expensive module, you may also use an ESP8266 module called the “D1 Mini”. Besides the cost and size advantage, its board layout is also more standardized between manufacturers. The D1 mini is widely available on [Amazon](#) and [eBay](#) for \$2 to \$6 each, depending on quantity. I used the D1 Mini for my completed NTP clock project, [described here](#).



Wemos D1 Mini

Here is a quick run-down of using the ESP8266 in the Arduino IDE:

1. Copy the following URL(s) into your Arduino Boards Manager list.

http://arduino.esp8266.com/stable/package_esp8266com_index.json

2. Next, configure the IDE. I am currently using IDE version 1.8.13.
 - a) Choose Tools-> Board -> ESP8266 Boards -> LOLIN(Wemos) D1 R2 and mini
 - b) Tools -> Port -> (choose the computer port attached to the ESP8266)

STEP A2: CONNECT THE DISPLAY

Connect the wires and apply power. Make sure the backlight is ON – if not, immediately disconnect and check your wiring. The most common failure at this point is improper wiring.

Please note that the display RST pin is not 5V tolerant, therefore connect it to the ESP8266 3.3V, or to 5V through a 10K resistor.

Display Pin	Display Label	Connects To ESP8266:
1	Vcc	“5V”
2	Gnd	“G”
3	CS	D8
4	RST	“3V3”
5	DC	D3
6	MOSI	D7
7	SCK	D5
8	LED	“5V”
9	MISO	n/c

STEP A3: INSTALL THE DISPLAY SOFTWARE

Install the TFT_eSPI library, as described in Step 3. Then edit your User_Setup.h file to include only the following DEFINES. Return to Step 3 and confirm the display works with the Hello World sketch.

```
#define ILI9341_DRIVER
#define TFT_CS    PIN_D8
#define TFT_DC    PIN_D3
#define TFT_RST   -1
#define LOAD_GLCD
#define LOAD_FONT2
#define LOAD_FONT4
#define LOAD_FONT6
#define LOAD_FONT7
#define LOAD_FONT8
#define LOAD_GFXFF
#define SPI_FREQUENCY 4000000
#define SPI_READ_FREQUENCY 2000000
#define SPI_TOUCH_FREQUENCY 2500000
```