# GPS clock

## Build a GPS-based clock using a microcontroller and LCD display.

Bruce E. Hall, W8BH


**Introduction.**

I recently built a WWVB clock, which is a lot of fun. As soon as I posted it, someone asked "What about a GPS clock?" I have made a few GPS clocks before, but none with a graphical LCD display. It is time to make one!

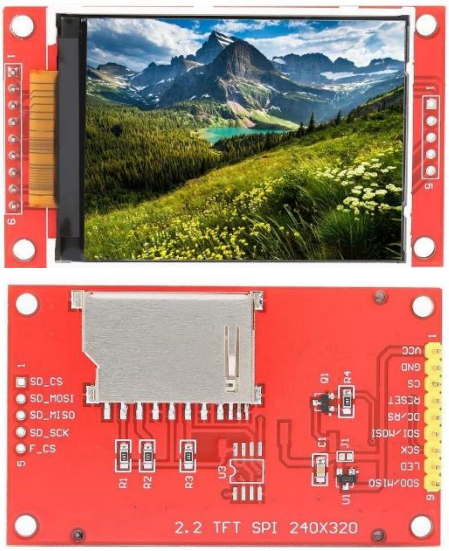
This article describes how GPS-controlled clocks work and gives you enough information to build your own precision timepiece, accurate to within a few microseconds of UTC(USNO)*. For my clock I chose a STM32 "Blue Pill" microcontroller, a 2.8" 320x240 pixel LCD display, and a GPS module that uses the MTK3339 chipset. 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.

*Your GPS module is accurate within tens of nanoseconds of official US time. The GPS information is captured by the microcontroller, and acts on this data within 3 microseconds of the GPS timing pulse. Additional time required to display the result.

The USNO Master clock, at left, produces the official time reference for the US Dept. of Defense. A second US time reference, time.gov, is maintained by the National Institute of Standards and Technology (NIST). Both are considered official US time.

**STEP 1: THE HARDWARE**

I assume that you have a suitable breadboard and 3.3V power supply. For my clock I am using a "Blue Pill" microcontroller and a 320x240 TFT display.




The Blue Pill is a low-cost, widely-available microcontroller board using a clone of the SMT32F103 microcontroller from ST Microelectronics. It is widely available on Amazon and eBay for about $3.00. I describe the Blue Pill and show its pinout in my Morse Tutor article.

The display is a 320x240 pixel TFT LCD on a carrier board, using the ILI9341 driver, and an SPI interface. It is a 3.3V device. 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 cost 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 5 pins on the display that connect to pins on the Blue Pill, and 3 pins that are power/ground related. The following table details the connections:

| Display Pin | Display Label | Connects To: | Function |
|---|---|---|---|
| **1** | Vcc | Vcc bus (3.3V) | Power |
| **2** | Gnd | Gnd bus | Ground |
| **3** | CS | Blue Pill, pin PA1 | Chip Select |
| **4** | RST | Vcc bus (3.3V) | Display Reset |
| **5** | DC | Blue Pill, pin PA0 | Data/Cmd Line |
| **6** | MOSI | Blue Pill, pin PA7 | SPI Data |
| **7** | SCK | Blue Pill, pin PA5 | SPI Clock |
| **8** | LED | Vcc bus (3.3V) | LED Backlight Power |
| **9** | MISO | Blue Pill, pin PA6 | SPI Data |

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.

Next, connect your GPS module. I am using the Adafruit Ultimate GPS module, but many other GPS modules will work. Only four connections to the GPS module are required: Connect power and ground first. Next, connect the data out or TX line from the GPS to the Blue Pill pin PA10. Finally, connect the 1 pulse-per-second timing output to the Blue Pill pin PA11. If your module does not have a 1PPS signal, don't worry: the clock will still work without it. It just won't have the microsecond accuracy that the timing signal provides.

**STEP 2: THE SOFTWARE**

I assume that you are comfortable with the Arduino IDE and know how to program a Blue Pill microcontroller. The Blue Pill was initially supported in the Arduino IDE with a fantastic package written by Roger Clarke and hosted at dan.drown.org. I had great success using this software but Roger no longer supports his package. In the meantime, STMicroelectronics, the makers of the microcontroller in the Blue Pill, now support the Arduino environment and have created their own software package. To use it, copy the following URL into your Arduino Boards Manager list.

https://github.com/stm32duino/BoardManagerFiles/raw/main/package_stmicroelectronics_index.json

The current version for this core is 2.2.0. For TFT support I am using "TFT_eSPI" by Bodmer, version 2.4.32. To install it, go to the Arduino library manager (Sketch->Include Libaries->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'd 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 the following DEFINEs:

```
#define STM32
#define ILI9341_DRIVER
#define TFT_SPI_PORT 1
#define TFT_MOSI PA7
#define TFT_MISO PA6
#define TFT_SCLK PA5
#define TOUCH_CS PA2
#define TFT_CS   PA1
#define TFT_DC   PA0
#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  40000000
#define SPI_READ_FREQUENCY  20000000
#define SPI_TOUCH_FREQUENCY  2500000
```

Next, configure the IDE for your Blue Pill. I am currently using IDE version 1.8.13.

a) Choose Tools-> Board -> STM32 boards (select from submenu) -> Generic STM32F1.
b) Tools -> Board -> Board Part Number -> Blue Pill F103CB (or C8 with 128k)
c) Upload method -> STM32CubeProgrammer (SWD)

For programming you will need an ST-LINK v2-compatible dongle, widely available on eBay and Amazon.

**STEP 3: HELLO, WORLD**

The following sketch will verify that your hardware is in working order, the STM32 package is correctly installed, the display library is correctly configured, and that you are able to upload code:

```
#include <TFT_eSPI.h>
#define TITLE "Hello, World!"
```

```
TFT_eSPI tft = TFT_eSPI();                          // display object

void setup() {
  tft.init();
  tft.setRotation(1);                               // portrait screen orientation
  tft.fillScreen(TFT_BLUE);                         // start with empty screen
  tft.setTextColor(TFT_YELLOW);                     // yellow on blue text
  tft.drawString(TITLE,50,50,4);                    // display text
}

void loop() {
}
```

If you see “Hello, World” on your display, you are ready to continue. If the display is upside-down, physically rotate the display or change the setRotation() parameter from 1 to 3.

**STEP 4: TESTING THE GPS 1PPS SIGNAL**

The GPS module needs a clear view of the sky to work well. Try to position yourself near a window, if possible. 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.

Apply power and wait for the module to get a satellite “fix”. On my Ultimate GPS board, the LED on the starts blinking immediately after power is applied. If not, check the power connections. A once-per-second blink means that the unit is searching for GPS satellites. A slower, once-per-15 seconds blink indicates that the unit is in fix. Be patient: it will take several minutes to get a fix. If you do not get a fix, try moving to a suitable location (or add an active antenna, such as Adafruit #960). Other GPS modules act differently. I have another GPS unit that indicates fix with a once-per-second LED, attached directly to its 1PPS signal. Your GPS module will not generate any 1pps signal until it has obtained satellite fix. Don’t proceed until the unit is ready.

The microcontroller will use a hardware interrupt to handle the GPS timing signal.

Interrupt code is as simple as this:

```
volatile byte pps  = 0;                             // GPS one-pulse-per-second flag

void ppsHandler() {                                 // 1pps interrupt handler:
  pps = 1;                                          // flag that signal was received
}
```

Interrupts must be enabled before they start. A special procedure called “attachInterrupt” is used:

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

This single line, placed in the setup() routine, calls the interrupt handler “ppsHandler” whenever the rising edge of a pulse is detected on the GPS 1pps signal line. Let’s add the interrupt code to our working display to see incoming 1pps pulses. Here is the entire sketch:

```
#include <TFT_eSPI.h>
#define GPS_PPS               PA11                  // GPS 1PPS signal pin

TFT_eSPI tft = TFT_eSPI();                          // display object
volatile byte pps  = 0;                             // GPS one-pulse-per-second flag

void ppsHandler() {                                 // 1pps interrupt handler:
  pps++;                                            // increment pulse count
}

void setup() {
  tft.init();
  tft.setRotation(1);                               // portrait screen orientation
  tft.fillScreen(TFT_BLACK);                        // start with blank display
  attachInterrupt(digitalPinToInterrupt(            // enable 1pps GPS time sync
    GPS_PPS), ppsHandler, RISING);
}

void loop() {
  tft.drawNumber(pps,50,50,7);                       // show pulse count on screen
  delay(100);                                        // wait 0.1s; no need to hurry!
}
```

Notice the two highlighted lines. In the interrupt handler, we increment the pulse counter. And in the loop() function, we display the counter. That's it. Run this sketch. If your GPS' pps signal is working, you should see the displayed value increment once per second. If it doesn't, make sure your GPS module has a good view of the sky and the unit has obtained a fix.

**STEP 5: TESTING THE GPS SERIAL DATA**

Let's create a sketch to test the serial connection. Most GPS modules transmits asynchronous serial data on their "Tx" line at a rate of 9600 baud. (You should check the specs of your module to be sure.) The microcontroller will receive this data on its serial Rx line, which is PA11. Here is a bare-bones sketch to see the data:

```
#include <TFT_eSPI.h>
TFT_eSPI tft = TFT_eSPI();                          // display object

void setup() {
  tft.init();
  tft.setRotation(1);                               // portrait screen orientation
  tft.fillScreen(TFT_BLACK);                        // start with blank display
  Serial1.begin(9600);                              // set baud rate of incoming data
}

void loop() {
  if (Serial1.available()) {                        // if a character is ready to read...
    char c = Serial1.read();                        // get it, and
    tft.print(c);                                   // show it on the display
  }
}
```

Setup() sets the baud rate. Then, in the program loop, characters are read and displayed. Run the sketch to see your GPS data. If successful, you should see a screenful of information! Notice that this simple sketch only displays the first 30 or so lines of data, and does not refresh. Press your microcontroller's Reset button if you want to see more.

**STEP 6: PARSING THE GPS DATA**

Take a look the data on your screen. 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 of an NMEA sentence:

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

| Name | Data | Description |
|---|---|---|
| **Sentence Identifier** | $GPGGA | Global Positioning System Fix Data |
| **Time** | 033757 | 03:37:57 UTC = 11:37:57 PM EDT |
| **Latitude** | 3942.9046,N | 39d 42.9046' N = 39.7151 N |
| **Longitude** | 08410.5099,W | 84d 10.5099 W = 84.1752 W |
| **Fix: 0 Invalid, 1 GPS, 2 DGPS** | 2 | Data is from a DGPS fix |
| **Number of Satellites** | 8 | 8 Satellites are in view |
| **Horizontal Dilution of Precision** | 1.05 | Relative accuracy of horizontal position |
| **Altitude** | 311.0, M | 311 meters above mean sea level |
| **Height above WGS84 ellipsoid** | -33.4, M | -33.4 meters |
| **Time since last DGPS update** | 0000 | No last update |
| **DGPS reference station id** | 0000 | No station id |
| **Checksum** | *61 | Used to check for transmission errors |

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

```
#include <TFT_eSPI.h>
#include <TinyGPS++.h>                          // gps functions - install within IDE
TFT_eSPI tft = TFT_eSPI();                      // display object
TinyGPSPlus gps;                                // gps object

void setup() {
  tft.init();
  tft.setRotation(1);                           // portrait screen orientation
  tft.fillScreen(TFT_BLACK);                    // start with blank display
  Serial1.begin(9600);                          // set baud rate of incoming data
}
```

In the loop() routine, send each character to the library as it is received. The function gps.encode() parses each character from the GPS module, and gps.time.isUpdated() returns true when a new time has been decoded. Finally, add a routine to print the time. The gps object "gps.time" contains methods that return the decoded hour, minute, and second.

```
void loop() {
  if (Serial1.available()) {                    // if a character is ready to read...
    char c = Serial1.read();                    // get the character
    gps.encode(c);                              // and feed it to the GPS parser.
    if (gps.time.isUpdated())                   // Wait until time has been updated
      displayTime();                            // then, display the time
  }
}
void displayTime() {
  int x=10,y=50,f=7;                            // screen position & font
  int hr = gps.time.hour();                     // get hour value
```

```
  int mn = gps.time.minute();                    // get minute value
  int sec = gps.time.second();                   // get second value
  tft.fillRect(x,y,250,60,TFT_BLACK);            // erase old time
  x+= tft.drawNumber(hr,x,y,f);                  // hours
  x+= tft.drawChar(':',x,y,f);                   // hour:min separator
  x+= tft.drawNumber(mn,x,y,f);                  // show minutes
  x+= tft.drawChar(':',x,y,f);                   // show ":"
  x+= tft.drawNumber(sec,x,y,f);                 // show seconds
}
```

Step 6 gives us a clock that displays UTC time. If your GPS has a satellite fix, you will see the time display about twice a second. The time is reported in coordinated universal time (UTC), which is 5 hours ahead of Eastern Standard Time.

**STEP 7: THE TIME LIBRARY**

So far, we have tested the LCD display, the 1pps interrupt, the GPS serial connection, and parsing the serial data. This is everything we need to set and display the time.

The Arduino time library, written by Michael Margolis and maintained by Paul Stoffregen, is a convenient collection of routines for timekeeping in the Arduino environment. The updated library is on GitHub here. You can install this library directly from within the Arduino IDE. Add its #include directive to the top of the sketch:

```
#include <TimeLib.h>                             // time/date functions
```

To set the system time, call the function setTime with time and date information obtained from the gps.time and gpd.date objects. The following will work nicely:

```
void syncWithGPS() {                             // set Arduino time from GPS
  if (!gps.time.isValid()) return;               // continue only if valid data present
  if (gps.time.age()>1000) return;               // don't use stale data
  int h = gps.time.hour();                       // get hour value
  int m = gps.time.minute();                     // get minute value
  int s = gps.time.second();                     // get second value
  int d = gps.date.day();                        // get day
  int mo= gps.date.month();                      // get month
  int y = gps.date.year();                       // get year
  setTime(h,m,s,d,mo,y);                         // set the system time
}
```

The display does not look any different with these code changes. However, setting the system time provides a convenient way of dealing with time and date from anywhere in the sketch.

**STEP 8: IMPROVING CLOCK ACCURACY**

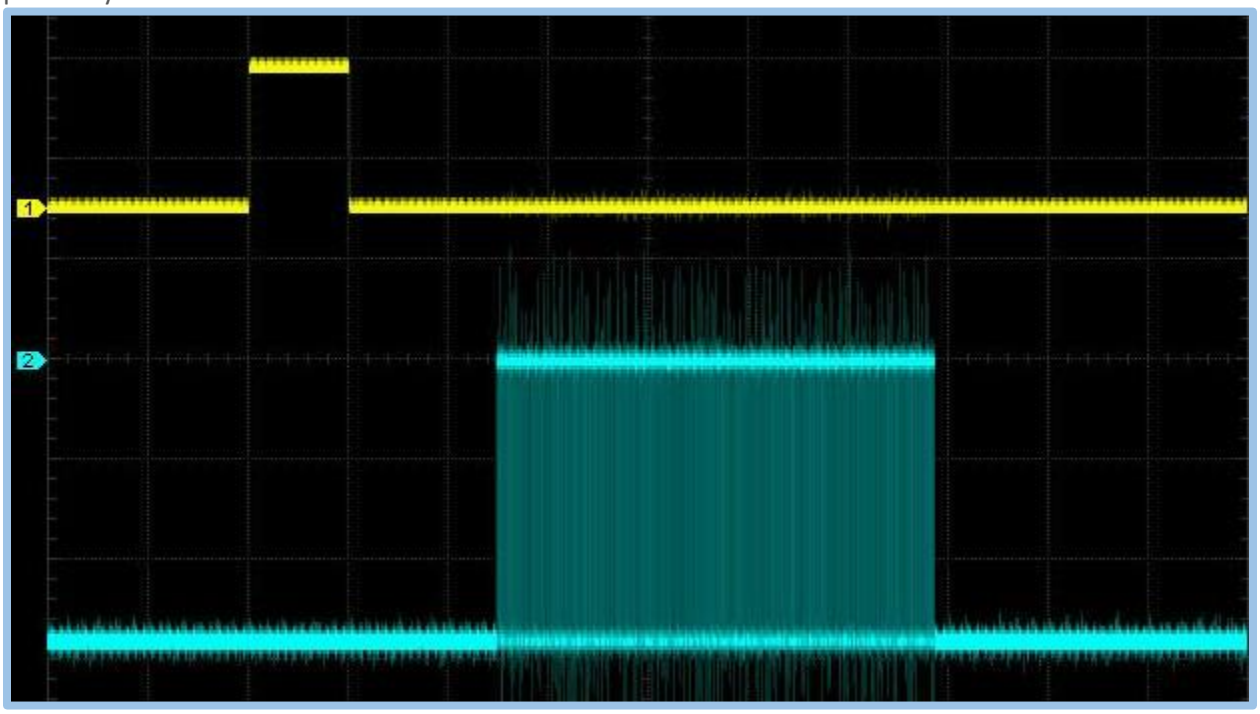
Compare the time on your clock to the US standard at time.gov, and you will notice that your clock is slightly…off. Mine is about half a second behind official time. This is not acceptable for a precision clock! Why does it happen?

The reason lies in how the GPS data is sent. At the start of each second, the GPS module sends time information for that second. Each NMEA sentence takes about 50-100 milliseconds to send over the serial connection at 9600 baud. Modules vary in terms of which sentences they send. The 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.

Note that the GPS software library provides a variable, 'age', that tells you how much time has passed since the data was decoded. But it does not tell you about the variable amount of time that passed between the beginning of the current 'second' and when the data sentence was fully received.

Fortunately, there is a better way. We can use the 1pps (1 pulse-per-second) timing 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 gives the corresponding time. The following digital oscilloscope image shows the relationship between the 1pps pulse in yellow and the serial data in blue.




*The Rigol 1054Z oscilloscope: at $399, one of the best bang-for-the-buck scopes you can buy!*

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.

Timing details vary from module to module. The pulse may be positve or negative. The pulse width is typically 100 mS, but may be 1 uS or shorter. Consult your GPS module's datasheet if you use a different module.

I have searched to see how others use the 1pps signal. There is very little information. If you find a better solution (I am sure there are many), I'd like to hear from you. The following method is simple and works. Here is the basic algorithm:

Main Loop:

a. Feed any incoming ASCII data to the GPS library
b. Synchronize clock (see below).
c. Update the display if system time has changed.

Synchronize Clock:

*As soon as a 1pps signal has just been received,* do the following:

a. Set the time to GPS time + 1 second

The most interesting aspect of this algorithm is adding one second to the time. We need to advance the clock one second, because the 1pps signal indicates the start of the *next* second. The GPS time on hand represents decoded serial data for the previous second.

We have all the components needed to implement the PPS algorithm:

Feed incoming data to the GPS library:

```
void feedGPS() {
  if (Serial1.available()) {                    // if a character is ready to read...
    char c = Serial1.read();                    // get the character
    gps.encode(c);                              // if input is complete, use it
  }
}
```

Display the time:

```
void displayTime() {
  int x=10, y=50, f=7;                          // screen position & font
  tft.fillRect(x,y,250,60,TFT_BLACK);           // erase old time
  x+= tft.drawNumber(hour(),x,y,f);             // hours
  x+= tft.drawChar(':',x,y,f);                  // hour:min separator
  x+= tft.drawNumber(minute(),x,y,f);           // show minutes
  x+= tft.drawChar(':',x,y,f);                  // show ":"
  x+= tft.drawNumber(second(),x,y,f);           // show seconds
}
```

And set the time from the GPS data:

```
void syncWithGPS() {                            // set Arduino time from GPS
  if (!gps.time.isValid()) return;              // continue only if valid data present
  if (gps.time.age()>1000) return;              // dont use stale data
  int h = gps.time.hour();                      // get hour value
  int m = gps.time.minute();                    // get minute value
  int s = gps.time.second();                    // get second value
  int d = gps.date.day();                       // get day
  int mo= gps.date.month();                     // get month
  int y = gps.date.year();                      // get year
  setTime(h,m,s,d,mo,y);                        // set the system time
  adjustTime(1);                                // and adjust forward 1 second
}
```

Notice the addition of adjustTime(), which sets the system time to GPS time + 1 second. The last thing needed is a way to trigger the sync when the 1pps signal is received. We do this by checking the pps flag, and doing the sync as soon as the flag is set:

```
void syncCheck() {
  if (pps) syncWithGPS();                       // is it time to sync with GPS?
  pps=0;                                        // reset flag, regardless
}
```

Step 8 gives us a clock, synchronized to UTC time, and accurate to within a few microseconds.

### STEP 9: 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). It requires a small modification to displayTime(). Add a DEFINE at the top of the sketch make it easy to change the color later.

```
#define TIMECOLOR TFT_CYAN
```

Does it bother you when 11:00:05 displayed as 11:0:5? Consider the following modifications:

```
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 no longer needs to be erased. And the display flicker caused this erasure is eliminated. The clock is starting to look nice, isn't it?

### STEP 10: TIME AND DATE

Time to add the date to our display. The time library give us access to the day, month, year, 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 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 10 clock displays UTC time and date. A screen border is also added to enhance the display.

**STEP 11: 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. Let's add the ability to display local time, taking daylight saving time into consideration.

Converting UTC to local standard time is straightforward. Most of the time zones on land are offset from UTC by a whole number of hours. I live in the Eastern US time zone, and my offset is – 5 hours. 11:00 UTC is 06:00 here. There are 3600 seconds in an hour, so this difference in time, expressed in seconds, -5*3600 = -18000 seconds. The Arduino time library keeps track of time in seconds. Therefore, converting UTC to local time is a matter of adding or subtracting seconds. For me, localStandardTime = utcTime-18000.

Dealing with daylight saving time is a bit more complicated, however. Some localities use it, others do not. Furthermore, summertime in the UK may begin and end at different times than other countries. In short, your clock will need different rules to follow than mine.

Calculating local time with DST is an interesting programming exercise. I did it for a previous clock. For this clock I am using a ready-made Timezone library by Jack Christenson. The code for this library is on GitHub, and can be installed from within the Arduino IDE. Install it and add the following lines to enable local time with DST. The library handles all of the calculations for you.

```
#include <Timezone.h>

TimeChangeRule EDT                                 // Local Timezone. Mine is EST/EDT.
  = {"EDT", Second, Sun, Mar, 2, -240};            // Set Daylight time here.  UTC-4hrs
TimeChangeRule EST                                 // For ex: "First Sun in Nov at 02:00"
  = {"EST", First, Sun, Nov, 2, -300};             // Set Standard time here.  UTC-5hrs
Timezone myTZ(EDT, EST);
```

You will need to adjust the rules, depending on your time zone. Notice that the offsets are expressed in minutes. In the Eastern US time zone, the standard time offset is -5 hours which is -300 minutes. The

offset during daylight saving hours is -4 hours or -240 minutes. DST in the US currently starts on the second Sunday in March at 02:00 AM and ends on the first Sunday in November at 02:00 AM.

The library gives us a function to convert UTC to local time: localTime = mtTZ.toLocal(utcTime). We add this call to our updateDisplay() routine.

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

We need to save the local time in global variable, lt, so that we can detect when the local date has changed (The local date changes at a different time than the UTC date). The clock now accurate 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
}
```

The Step 11 clock presents local time and date, with automatic DST adjustment, in 12-hour format.

**STEP 12: 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 it synching to GPS every second, or was the last sync hours ago?" A status indicator would be nice. First, create a global variable to store the time of the last successful GPS synchronization, then compare this value to the current time. A color-coded status bar will tell us 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:

```
time_t lastSync    = 0;                            // UTC time of last GPS sync

void showClockStatus () {
  int color,x=20,y=200,w=80,h=20,ft=2;             // screen position and size
  if (!lastSync) return;                           // haven't decoded time yet
  int minPassed = (now()-lastSync)/60;             // how long ago was last decode?
  if (minPassed<60) color=TFT_GREEN;               // green is < 1 hr old
  else if (minPassed<1440) color=TFT_ORANGE;       // orange is 1-24 hr old
  else color=TFT_RED;                              // red is >24 hr old
  tft.fillRoundRect(x,y,80,20,5,color);            // show status indicator
}
```

The number of satellites being received is a good indicator of reception quality. Let's add this as a second status indicator:

```
void showSatellites() {
  int x=200,y=200,w=50,h=28,ft=4;                  // screen position and size
  int sats = 0;                                    // number of satellites
  if (gps.satellites.isValid())
    sats = gps.satellites.value();                 // get # of satellites in view
  tft.setTextColor(TFT_YELLOW);
  tft.fillRect(x,y,w,h,TFT_BLACK);                 // erase previous count
  tft.drawNumber(sats,x,y,ft);                     // show latest satellite count
}
```

Here, gps.satellites.value() is used to get the most recent satellite count.

The two status indicators are complimentary. If the displayed satellite count is 0, there is probably no serial data transmission from the GPS module. On the other hand, if the satellite count is good but the data is stale, serial data is being received but there is no 1pps synchronization signal.

The Step 12 clock adds helpful status GPS status indicators.

**FINISHING TOUCHES**



The final sketch for this series, **GPS_CLOCK_single**, adds 12/24hr display, local/UTC display. Touch control is used to toggle the options. (To enable touch, connect your display pins as follows: T_DO to MISO, T_DIN to MOSI, T_CLK to SCK, and T_CS to Blue Pill PA2. T_IRQ remains unconnected. See my touch tutorial for more information.)

The **GPS_CLOCK_dual** sketch shows local and UTC simultaneously. No touch needed. The **GPS_CLOCK_triple** sketch gives you just about everything: time, dual time, location, altitude, speed, and direction.

Links to all my files are listed at right. Drop me a line if you build your own GPS clock!

73, Bruce. *Last Updated: 5/12/2022 2:56 PM*

**Project Files**

Part 1 (this document)
Part 2: Hardware
Schematic
Source Code
PCB Gerbers
Enclosure STL files
YouTube Video