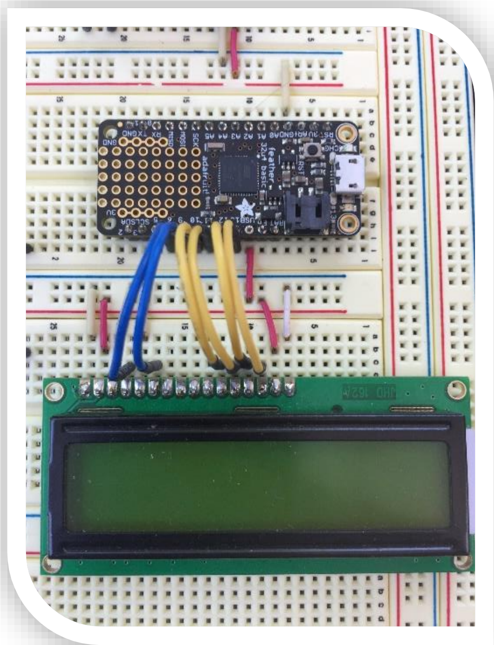


Build an Accurate GPS-Controlled Clock: Step by Step

Bruce E. Hall, W8BH

In the previous article, I described how to build a GPS-controlled clock with a seven-segment LED display, suitable for handing on the wall. And that's exactly how I use mine. It has worked flawlessly for one year, never gaining or losing a second, providing my ham radio shack with simultaneous local and UTC time. I am glad I built it!

And yet, there was one small detail that kept nagging at me: the time was *never* exactly correct! But how could this be? It gets its time from GPS, a very reliable and accurate standard. But sure enough, the displayed time was always about half a second slow. What is the big deal, you might say? Half a second! If you are not bothered by this at all, go enjoy life and read no further. But it you are the kind of person who likes to ring in the new year at exactly 12:00:00, and not a half-second later, this article is for you.

I will show you how to build an accurate GPS-based clock using an Arduino-type microcontroller and small 16x2 LCD display. Using the steps below, you can apply the same principles to other microcontrollers and displays.

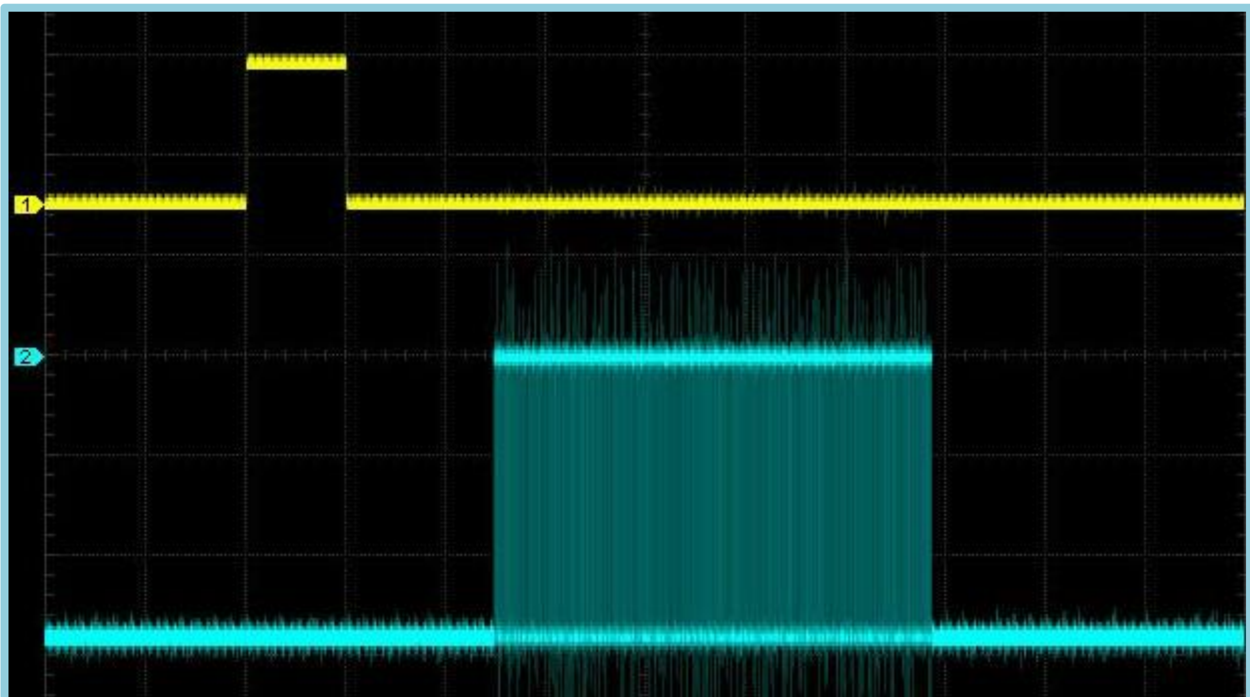
WHY SIMPLE GPS-BASED CLOCKS MAY BE INACCURATE

In my original clock, the microcontroller reads ascii data from the GPS module and decodes the time information from that data. Periodically, the microcontroller's time is reset according to the GPS information. Here is the problem: it takes time to read the 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. 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 from the beginning of the current 'second' until the data sentence was fully received.

Fortunately there is a better way. 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 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 as short as 1 uS. Consult your GPS module's datasheet if you use a different module.

By synchronizing the clock to the 1pps signal, we avoid the delay associated with the slow serial data transfer.

BASIC ALGORITHM

I have searched to see how others use the 1pps signal. Amazingly I cannot find a single article describing how to use 1pps in an Arduino-based clock. Please let me know if any of this information is helpful to you. And 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:

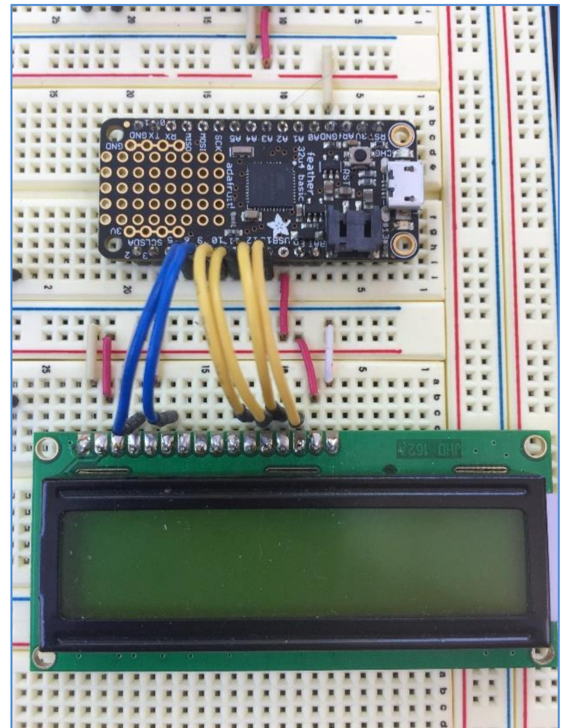
a. Feed any incoming ASCII data to the GPS library
b. Synchronize clock (see below).
c. Update the display if system time has changed.

Synchronize Clock:

a. *When a 1pps signal has just been received,* do the following:
b. Get the GPS time.
c. If data is current, set system time to GPS time + 1 second

The most interesting part 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.

STEP 1: STARTING THE BUILD



I will assume that you have a suitable breadboard and connecting wires. I am using the following hardware, all available at Adafruit.com:

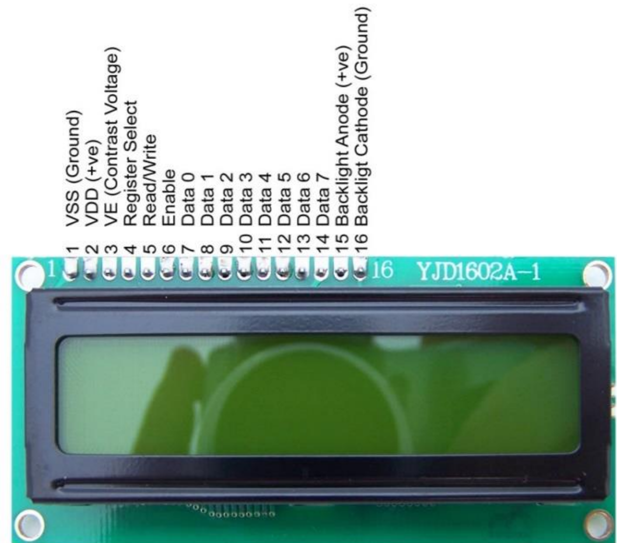
Ultimate GPS breakout	Adafruit #746	$40
Feather 32u4 basic proto	Adafruit #2771	$20
DS3231 Precision RTC	Adafruit #3013	$14
Standard 16x2 LCD	Adafruit #181	$10

As always, I like to start small and test as I go along. Let's breadboard the microcontroller board and a 16x2 LCD display. The display will be used to count 1pps pulses. The RTC is optional.

Place the micro and display on your breadboard as shown. We are going to power the project via the onboard USB connector, so connect the feather pin GND to the breadboard BLUE bus, and feather pin VBUS to the breadboard RED bus.

Next, connect the feather to the LCD display:

32u4 Feather	LCD Display
Digital Pin 6	pin 4
Digital Pin 9	pin 6
Digital Pin 10	pin 11
Digital Pin 11	pin 12
Digital Pin 12	pin 13
Digital Pin 13	pin 14



The LCD has a 16-pin interface. On the LCD, connect pins 2 & 15 to +5v power and pins 1, 5, and 16 to ground. Pin 3 is the contrast voltage. For some displays, you can connect this directly to ground. For others, a 1K resistor to ground works better. You can vary display contrast with a potentiometer, which is included in the Adafruit offering.

When everything is hooked up, the LCD will have pins 7-10 disconnected. On the feather, seven pins in a row are connected, plus GND on the opposite side of the board.

Apply power via a suitable 5V supply to the feather's USB port, and you should see the glow of your LCD backlight. If not, unplug and check all power connections. In addition, the top row of the LCD should display a line of solid-block characters. If not, the display contrast may need to be adjusted. Vary the voltage on pin3 to get good display contrast.

STEP 2: TESTING THE DISPLAY

Once you see the solid block characters on the display, it is time to verify the connections between the microcontroller and display. I described a small set of routines in a previous article that write to a 16x2 display. This time we will use a ready-made library called "LiquidCrystal".

Try running [this code] from your Arduino environment:

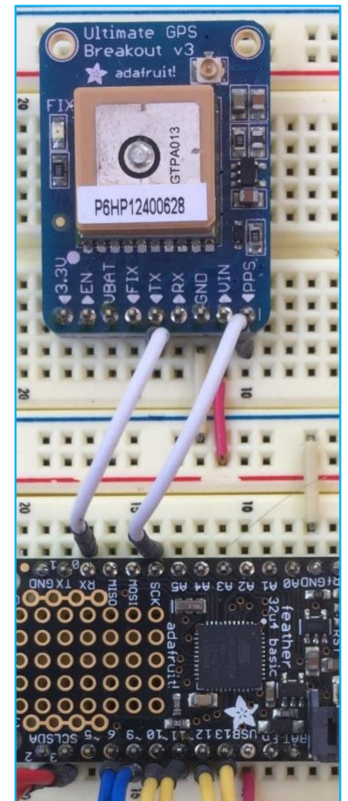
```
#include <LiquidCrystal.h>
LiquidCrystal lcd(6,9,10,11,12,13);

void setup()
{
  lcd.begin(16,2);
  lcd.print("Hello, World!");
}

void loop()
{
}
```

If you see "Hello, World!" on the display, the programming and hardware connections are in good working order.

STEP 3: TESTING THE GPS 1PPS SIGNAL



Time to add the GPS module. It is simpler to connect that the LCD: only power, ground, and two data lines. First, connect the GPS module power “Vin” and “GND” to your breadboards power bus. Connect the GPS pin “Tx” to the Feather “Rx” and the GPS pin “PPS” to the Feather pin “SCK” as shown here.

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”. An LED on the Ultimate GPS board should start blinking immediately after power is applied. If not, check the power connections. A once-per-second blink means that the unit is searching for a GPS satellite fix. 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, that can be placed in a suitable location).

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.

The GPS module will not generate any 1pps signal until it has obtained satellite fix. Don’t proceed until the unit is ready.

I think the easiest way to handle this input signal is to trigger an interrupt. Use a hardware interrupt if it is available. If not, the Arduino supports “pin-change” interrupts which will work in many cases. On my 32u4 board I use a pin-change interrupt and it works.

Code for an interrupt looks like this:

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

void isr()                                           // INTERRUPT SERVICE ROUTINE
{
  pps = 1;                                           // Flag the 1pps input signal
}
```

Interrupts must be enabled before they start. I used a library called “EnableInterrupt” which very nicely unifies the way hardware and pin-change interrupts are called. The following lines are needed to enable the above interrupt:

```
#include <EnableInterrupt.h>                     // for pin-change interrupt
#define PPS_PIN                SCK              // Pin on which 1PPS line is attached

void setup()
{
  enableInterrupt(PPS_PIN, isr, RISING);         // enable GPS 1pps interrupt input
}
```

That's all there is to an interrupt. Let's add the interrupt code to our working display code to display incoming 1pps pulses:

```
#define PPS_PIN    SCK     // digital Pin on which 1PPS line is attached

#include <LiquidCrystal.h>            // Library for LCD display
#include <EnableInterrupt.h>          // Library for pin-change interrupt

LiquidCrystal lcd(6,9,10,11,12,13);
volatile int pps = 0;

void isr()                            // INTERRUPT SERVICE ROUTINE
{
  pps += 1;                           // increment pulse count
}

void setup()
{
  lcd.begin(16,2);
  enableInterrupt(PPS_PIN, isr, RISING);
}

void loop()
{
  lcd.clear();                        // start at top of display
  lcd.print(pps);                     // show pulse count on display
  delay(100);                         // wait 0.1 sec; no need to hurry!
}
```

Run this sketch. If all goes well, the display will now increment once per second

STEP 4: TESTING THE GPS SERIAL DATA

Let's write some software to test the serial connection. The GPS module transmits asynchronous serial data on its "Tx" line at a rate of 9600 baud. (Some older modules use 4800, so you should check the specs of your module if you are using different hardware.) Our microcontroller receives this data on its "Rx" line. Here is a simple sketch to see the data:

```
#define SerialGPS Serial1                         // use hardware UART on 32u4

void setup()
{
  SerialGPS.begin(9600);                          // GPS communication at 9600 baud
  Serial.begin(9600);                             // set serial output to 9600 baud
  while (!Serial);                                // wait for serial output monitor
  Serial.println("Waiting for GPS data...");      // show something until GPS kicks in
}

void loop()
{
  if (SerialGPS.available())                      // a character is ready to read...
```

```
  {
    char c = SerialGPS.read();                    // so get it
    Serial.print(c);                              // and display it
  }
}
```

The first routine runs at the beginning of the sketch, initializing our serial connections. The second routine, loop(), runs as a continuous loop after setup() as finished. Loop() waits for characters to arrive on the serial input port, and echoes them one at a time on the output port. Simple! Run the sketch, and open the Arduino serial output monitor. If everything is configured correctly, you will see a flood of data from the GPS module. What does it all mean?

STEP 5: 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"

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", available at https://github.com/mikalhart/TinyGPS. 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 <TinyGPS.h>                              // arduiniana.org/libraries/TinyGPS/
#define SerialGPS Serial1                         // use hardware UART on 32u4
TinyGPS gps;                                      // the GPS parser object

void setup()
{
  SerialGPS.begin(9600);                          // GPS communication at 9600 baud
  Serial.begin(9600);                             // set serial output to 9600 baud
  while (!Serial);                                // wait for serial output monitor
  Serial.println("Waiting for GPS data...");      // show something until GPS kicks in
}
```

In the loop() routine, send each character to the library instead of serial.print(). The function gps.encode() accepts each character, and returns true when the NMEA sentence is complete. Finally, add a routine to print the time. The function gps.crack_datetime() returns the time as hours, minutes, seconds, etc.

```
void loop()
{
  if (SerialGPS.available())                    // if character is available
  {
    char c = SerialGPS.read();                  // read the next character
    if (gps.encode(c))                          // if input is complete, use it
      printTime();                              // to print the time
  }
}

void printTime()
{
  byte hr,mn,sec;
  gps.crack_datetime(NULL,NULL,NULL,            // get time info, ignore date
      &hr,&mn,&sec,NULL,NULL);
  Serial.print(hr); Serial.print(":");          // print out time (crudely!)
  Serial.print(mn); Serial.print(":");
  Serial.print(sec); Serial.println(" UTC");
}
```

Run the sketch. If your GPS has a satellite fix, you will see the time print about twice a second. The time is reported in coordinated universal time, 5 hours ahead of Eastern Standard Time.

STEP 6: DISPLAYING THE TIME

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 by Paul Stoffregen is a convenient collection of routines for timekeeping in the Arduino environment. A good description is found here, and the updated library is on GitHub here.

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

```
void SyncWithGPS()
{
  byte h, m, s, mon, d;
  int y;
  unsigned long age;
  gps.crack_datetime(&y,&mon,&d,&h,&m,&s,NULL,&age);   // get time data from GPS
  if (age<1000)                                        // don't use data older than 1 second
  {
    setTime(h,m,s,d,mon,y);                            // set the system time
  }
}
```

The previous sketch, modified to use system time, is here. The last thing to do is modify PrintTime, above, to display the time on the LCD instead of the sending it to the serial monitor:

```
void ShowTime()
{
  lcd.clear();
  lcd.print(hour());   lcd.print(":");
  lcd.print(minute()); lcd.print(":");
  lcd.print(second()); lcd.print(" UTC");
}
```

Remember to add the LCD library back in, and initialize it. The entire sketch is here.

STEP 7: IMPROVING TIME ACCURACY WITH 1PPS SYNCRONIZATION

It's finally time to modify our basic GPS clock, adding synchronization with the GPS 1pps signal. Let's modify the main program loop to match the 'Basic Algorithm' I described above. Here are the key changes:

```
volatile int pps = 0;                                  // 1 = 1pps signal has been received

void isr()                                             // INTERRUPT SERVICE ROUTINE
{
  pps = 1;                                             // Flag the 1pps input signal
}

void setup()
{
  SerialGPS.begin(9600);                              // GPS communication at 9600 baud
  lcd.begin(16,2);                                    // initialize 16x2 character LCD display
  enableInterrupt(PPS_PIN, isr, RISING);              // enable GPS 1pps interrupt input
}

void loop()
{
  FeedGpsParser();                                     // decode incoming GPS data
  SyncCheck();                                         // synchronize to GPS or RTC
  UpdateDisplay();                                     // if time has changed, display it
}

void FeedGpsParser()
{
  while (SerialGPS.available())                        // look for data from GPS module
  {
     char c = SerialGPS.read();                        // read in all available chars
     gps.encode(c);                                    // and feed chars to GPS parser
  }
}

void SyncCheck()
{
  if (pps)                                             // is it time to sync with GPS yet?
    SyncWithGPS();                                     // yes, so attempt it.
  pps = 0;                                             // reset 1-pulse-per-second flag
}
```

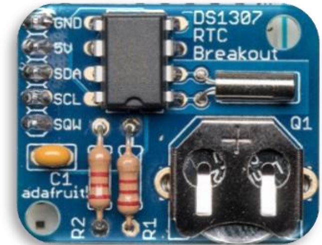
New code relating to our 1pps interrupt is highlighted in yellow. The clock now synchronizes with GPS only after the pps has been received. Finally, add one second to the time in SyncWithGPS() with the following call: adjustTime(1). Download the full sketch here.

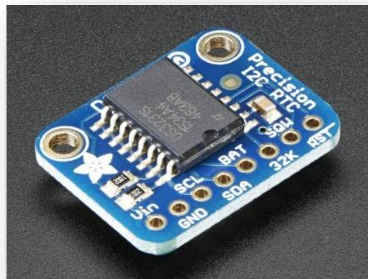
STEP 8: ADDING A REAL-TIME CLOCK

If the GPS is disconnected for a day or more, the time will start to drift.

The Arduino time library routines work well for short periods of time. Long-term accuracy depends on the precision and stability of the microcontroller's oscillator. A good crystal-based oscillator with an accuracy of 1 part per million may gain or lose a few seconds every month. In typical microcontroller environments, time drift can be minutes a day. Time is unreliable after a few weeks of use.

Enter the real-time clock. The most-common RTC is the DS1307, pictured here. There are many code libraries available for the '1307. Note the battery clip. With a coin cell will it will keep track of time for up to 5 years. The DS1307 uses a 32 kHz crystal (the silver cylinder-shaped component below the word 'Breakout') to improve timekeeping accuracy. Nevertheless, it is not a high-precision timekeeper, and will gain/lose up to 2 seconds a day depending on ambient temperature.

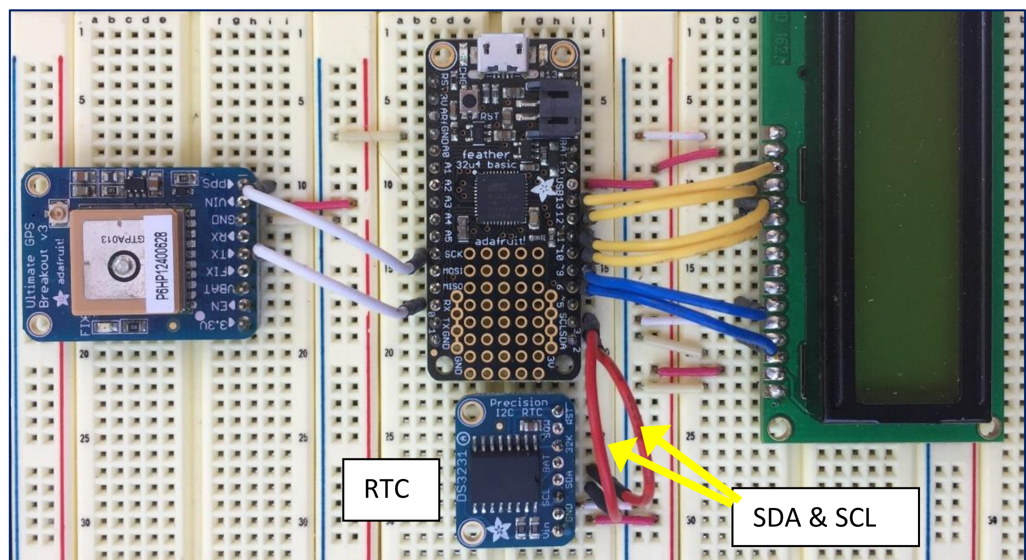




If a GPS signal is available on a fairly regular basis, say, once per day, the DS1307 is a good choice. It is inexpensive, readily available, and easy to use. However, time accuracy can be significantly improved for a few dollars more. The DS3231 chip, containing a temperature-controlled crystal oscillator (TXCO), limits drift to less than one minute *per year*. Cheap DS3231 modules are available on eBay for $1. But is highly unlikely that they contain a genuine Maxim DS3231. A better breadboarding choice would be the Adafruit #3103, pictured at left.

See my DS1307 article for a low-level programming interface. For simplicity's sake (and better code), I am using a code library for this project. Download and install Paul Stoffregen's DS1307RTC library from github. The DS1307 and DS3231 have the same interface and can use the same libraries.

Wiring up an RTC module requires four connections. First, connect the two power pins Vin and GND to the appropriate breadboard busses. Next, connect the RTC pins SCL and SDA to the same-named pins on the Feather.



The software library is equally simple.

There is a routine for setting time and one for getting the time. Time can be expressed in terms of unix time (an integer equaling the number of seconds elapsed since 00:00 UTC on 01 Jan 1970) or the traditional hours, minutes, seconds, etc. Unix time is helpful to use when trying to determine the amount of time between two events.

Let's incorporate the RTC into our code, in order to minimize drift when GPS is not available. The first step is to add the library:

```
#include <DS1307RTC.h>                                     // github.com/PaulStoffregen/DS1307RTC
```

Next, use the function RTC.set() to set the time in our real-time clock. We will add a routine that will periodically update the real-time clock with the current time. I have arbitrarily chosen an update interval of once per day:

```
#define RTC_UPDATE_INTERVAL    SECS_PER_DAY              // time(sec) between RTC SetTime events
time_t lastSetRTC  = 0;                                  // time that RTC was last setvoid UpdateRTC()

void UpdateRTC()
// keep the RTC time updated by setting it every (RTC_UPDATE_INTERVAL) seconds
// should only be called when system time is known to be good, such as in a GPS sync event
{
  time_t t = now();                                      // get current time
  if ((t-lastSetRTC) >= RTC_UPDATE_INTERVAL)             // is it time to update RTC internal clock?
  {
    RTC.set(t);                                          // set RTC with current time
    lastSetRTC = t;                                      // remember time of this event
  }
}
```

The correct time is now stored and updated in our real-time clock. We can use that time to update our clock if the GPS unit every loses fix (and stops providing data). If the GPS loses fix, how soon should we switch over to RTC time? An hour, maybe? I arbitrarily chose 60 seconds. To keep track, we will need a variable to store the time of last GPS synchronization. Check out the following changes to our synchronization routines, highlighted in yellow. The completed sketch is here.

```
#define SYNC_TIMEOUT           60                        // time(sec) without GPS input before errorime_t
syncTime    = 0;                                         // time of last GPS or RTC synchronization

void SyncCheck()
{
  unsigned long timeSinceSync = now()-syncTime;          // how long has it been since last sync?
  if (pps                                                // is it time to sync with GPS yet?
    SyncWithGPS();                                       // yes, so attempt it.
  pps = 0;                                               // reset 1-pulse-per-second flag, regardless
  if (timeSinceSync >= SYNC_TIMEOUT)                     // GPS sync has failed
    setTime(RTC.get());                                  // set system time from RTC
}

void SyncWithGPS()
{
  byte h, m, s, mon, d;    int y;
  unsigned long age;
  gps.crack_datetime(&y,&mon,&d,&h,&m,&s,NULL,&age);   // get time data from GPS
  if (age<1000)                                        // dont use data older than 1 second
  {
    setTime(h,m,s,d,mon,y);                            // set system time
    adjustTime(1);                                     // 1pps signal = start of next second
    syncTime = now();                                  // remember time of this sync
    UpdateRTC();                                       // update RTC periodically
  }
}
```

STEP 9: FINISHING TOUCHES

The logic for the clock is now complete. But if you watch the display for a while, you will notice that it doesn't display leading zeroes yet (12:01:04 is displayed as '12:1:4'). Its doesn't display the date. It doesn't even display the local time! These details are "a simple matter of programming." The complete source code is printed below. I couldn't resist adding a few bells and whistles:

- routines for 12hr local and 24hr UTC time display
- automatic daylight-saving time correction.
- diagnostic output on the USB serial port
- Indicator for GPS lock

There is a lot more you could add. How about:

- A timeserver with Wi-Fi or Ethernet connectivity
- Bluetooth
- Infrared control
- Some pushbuttons for display mode, brightness, etc
- Alarms
- Temperature monitoring and display

Now go build your own clock!

DOWNLOADS:

step 2: verify LCD display

step 3: verify GPS 1pps signal

step 4: verify GPS serial data

step 5: parse GPS serial data

step 6: a simple clock

step 7: add 1pps synchronization

step 8: add an RTC

step 9: finishing touches (final build)

SOURCE CODE:

```
//------------------------------------------------------------------------------------------------------
// clock2: a GPS-based clock using 16x2 character LCD display      >>> COMPLETE <<<<<
//
// The time is displayed in local time or UTC, supplied by GPS and/or RTC when GPS is unavailable.
// Local time automatically tracks daylight saving time for selected time zone.
// Time and GPS/RTC status messages and are copied to the serial monitor
//
// Author    : Bruce E. Hall, W8BH <bhall66@gmail.com>
// Website   : w8bh.net
// Version   : 1.0
// Date      : 31 May 2017
// Target    : 32u4 microcontroller on Adafruit "Feather" board
// Platform  : Arduino 1.8
// Size      : 14394 bytes (50% used)
//
// NOTE      : This is Final part of series on building a GPS-based LCD clock.
//             Please refer to http://w8bh.net/avr/clock2.pdf for the entire series
//
// Hardware  : 32u4 feather basic proto  - Adafruit #2771
//             Standard 16x2 LCD module  - Adafruit #181 or similar
//             Ultimate GPS breakout     - Adafruit #746
//             DS3231 Precision RTC      - Adafruit #3013
//
// Wiring    : Feather "VUSB", LCD pins 2,15, GPS "Vin", RTC "Vin" to +V bus
//             Feather "GND", LCD pins 1,3,5,16, GPS "GND", RTC "GND" to GND bus
//             Feather "6" to LCD pin 4
//             Feather "9" to LCD pin 6
//             Feather "10" to LCD pin 11
//             Feather "11" to LCD pin 12
//             Feather "12" to LCD pin 13
//             Feather "13" to LCD pin 14
//             Feather "Rx" to GPS "Tx"
//             Feather "SCK" to GPS "PPS"
//             Feather "SDA" to RTC "SDA"
//             Feather "SCL" to RTC "SCL"
// -----------------------------------------------------------------------------------------------------

// GLOBAL DEFINES
#define PPS_PIN               SCK                        // Pin on which 1PPS line is attached
#define SYNC_INTERVAL         10                         // time, in seconds, between GPS sync attempts
#define SYNC_TIMEOUT          60                         // time(sec) without GPS input before error
#define RTC_UPDATE_INTERVAL   SECS_PER_DAY               // time(sec) between RTC SetTime events

// INCLUDES
#include <LiquidCrystal.h>     // https://github.com/arduino/Arduino/tree/master/libraries/LiquidCrystal
#include <TimeLib.h>           // Time functions  https://github.com/PaulStoffregen/Time
#include <TinyGPS.h>           // GPS parsing     https://github.com/mikalhart/TinyGPS
#include <EnableInterrupt.h>   // Pin-change Int  https://github.com/GreyGnome/EnableInterrupt
#include <DS1307RTC.h>         // DS1307/DS3231   https://github.com/PaulStoffregen/DS1307RTC

//  GLOBAL VARS & OBJECTS
LiquidCrystal   lcd(6,9,10,11,12,13);
TinyGPS         gps;
time_t displayTime = 0;                                  // time that is currently displayed
time_t syncTime    = 0;                                  // time of last GPS or RTC synchronization
time_t lastSetRTC  = 0;                                  // time that RTC was last set
volatile int pps   = 0;                                  // GPS one-pulse-per-second flag
time_t dstStart    = 0;                                  // start of DST in unix time
time_t dstEnd      = 0;                                  // end of DST in unix time
bool gpsLocked     = false;                              // indicates recent sync with GPS
int currentYear    = 0;                                  // used for DST


// -----------------------------------------------------------------------------------------------------
// SERIAL MONITOR ROUTINES
// These routines print the date/time information to the serial monitor
```

```
// Serial monitor must be initialized in setup() before calling


void PrintDigit(int d)
{
  if (d<10) Serial.print('0');
  Serial.print(d);
}


void PrintTime(time_t t)
// display time and date to serial monitor
{
    PrintDigit(month(t));  Serial.print("-");
    PrintDigit(day(t));    Serial.print("-");
    PrintDigit(year(t));   Serial.print(" ");
    PrintDigit(hour(t));   Serial.print(":");
    PrintDigit(minute(t)); Serial.print(":");
    PrintDigit(second(t)); Serial.println(" UTC");
}


// ---------------------------------------------------------------------------------------------
// DST & LOCAL TIME ROUTINES
// The following routines support automatic daylight saving time adjustment
//
// Daylight Saving Time constants:
// Please set as UTC values according to current DST transition times
//
// Select timeZone offset according to your local time zone:


const int timeZone = -5;                             // Eastern Standard Time (EST)
//const int offset = -6;                             // Central Standard Time (CST)
//const int offset = -7;                             // Mountain Standard Time (MST)
//const int offset = -8;                             // Pacific Standard Time (PST)

#define DST_START_WEEK      2                        // Second Sunday
#define DST_START_MONTH     3                        // in March
#define DST_START_HOUR      7                        // at 2AM EST = 0700 UTC

#define DST_END_WEEK        1                        // First Sunday
#define DST_END_MONTH      11                        // in November
#define DST_END_HOUR        6                        // at 2AM EDT = 0600 UTC


time_t timeChange(int hr, int wk, int mo, int yr)
// returns unix time of DST transition according to hr, wk, mo, and yr
// Routine first calculates time on first day of month, at specified time
// then adds number of days to first Sunday, then number of additional weeks
{
  tmElements_t tm;                                   // set up time elements struct
  tm.Year   = yr - 1970;                             // need unix year, not calendar year
  tm.Month  = mo;
  tm.Day    = 1;                                     // start on first day of month
  tm.Hour   = hr;                                    // use UTC hour, not local hour
  tm.Minute = 0;
  tm.Second = 0;
  time_t t = makeTime(tm);                           // convert to unix time
  int daysTillSunday = (8 - weekday(t)) % 7;         // how many days until first Sunday?
  t += (daysTillSunday + (wk-1)*7)*SECS_PER_DAY;     // adjust time for additional days
  return t;
}


void PrintDST()
// debug function to show DST start & end times
{
    Serial.print("DST starts at ");
```

```
    PrintTime(dstStart);
    Serial.print("DST ends at ");
    PrintTime(dstEnd);
}


void initDST(int yr)
// establish start and end times for DST in a given year.  Call at least once a year!
{
  dstStart = timeChange(DST_START_HOUR, DST_START_WEEK, DST_START_MONTH, yr);
  dstEnd   = timeChange(DST_END_HOUR,   DST_END_WEEK,   DST_END_MONTH,   yr);
}


void UpdateDST()
// sets start/end times for DST
// assumes that current time is set in UTC
{
  int yr = year();                                        // what year is it?
  if (yr != currentYear)                                  // a new year started, need new DST info
  {
    initDST(yr);                                          // find DST times for new year
    currentYear = yr;                                     // save current year
  }
}


bool isDST(time_t t)                                      // returns TRUE if time is in DST window
{
  return ((t >= dstStart) && (t < dstEnd));
}


time_t LocalTime()                                        // returns local time, adjusted for DST
{
  time_t t = now();                                       // get UTC time
  if (isDST(t)) t += SECS_PER_HOUR;                       // add DST correction
  t += (timeZone * SECS_PER_HOUR);                        // add timeZone correction
  return t;
}


// ---------------------------------------------------------------------------------------------------
//  RTC SUPPORT
//  These routines add the ability to get and/or set the time from an attached real-time-clock module
//  such as the DS1307 or the DS3231.  The module should be connected to the I2C pins (SDA/SCL).


void PrintRTCstatus()
// send current RTC information to serial monitor
{
    time_t t = RTC.get();
    if (t)
    {
      Serial.print("The RTC is OK, Time = ");
      PrintTime(t);
    }
    else if (RTC.chipPresent())
      Serial.println("The RTC is stopped.  Please set the time.");
    else
      Serial.println("ERROR: cannot read the RTC.");
}


void SetRTC(time_t t)
{
  if (RTC.set(t))
  {
    Serial.print("RTC time set to ");
```

```
    PrintTime(t);
  }
  else Serial.print("ERROR: cannot set RTC time");
}


void ManuallySetRTC()
// Use this routine to manually set the RTC to a specific UTC time.
// Since time is automatically set from GPS, this routine is mainly for
// debugging purposes.  Change numeric constants to the time desired.
{
  tmElements_t tm;
  tm.Year   = 2017 - 1970;                              // Year in unix years
  tm.Month  = 5;
  tm.Day    = 31;
  tm.Hour   = 5;
  tm.Minute = 59;
  tm.Second = 30;
  SetRTC(makeTime(tm));                                 // set RTC to desired time
}


void UpdateRTC()
// keep the RTC time updated by setting it every (RTC_UPDATE_INTERVAL) seconds
// should only be called when system time is known to be good, such as in a GPS sync event
{
  time_t t = now();                                     // get current time
  if ((t-lastSetRTC) >= RTC_UPDATE_INTERVAL)            // is it time to update RTC internal clock?
  {
    SetRTC(t);                                          // set RTC with current time
    lastSetRTC = t;                                     // remember time of this event
  }
}


// ---------------------------------------------------------------------------------------------------
// LCD SPECIFIC ROUTINES
// These routines are used to display time and/or date information on the LCD display
// Assumes the presence of a global object "lcd" of the type "LiquidCrystal" like this:
//    LiquidCrystal   lcd(6,9,10,11,12,13);
// where the six numbers represent the digital pin numbers for RS,Enable,D4,D5,D6,and D7 LCD pins


void ShowDate(time_t t)
{
  int d = day(t);
  if (d<10) lcd.print("0");
  lcd.print(d);

  int m = month(t);
  lcd.print(monthShortStr(m));

  int y = year(t)%100;
  if (y<10) lcd.print("0");
  lcd.print(y);
}


void ShowTime(time_t t)
{
  int h = hour(t);
  if (h<10) lcd.print("0");
  lcd.print(h);
  lcd.print(":");

  int m = minute(t);
  if (m<10) lcd.print("0");
  lcd.print(m);
  lcd.print(":");
```

```
  int s = second(t);
  if (s<10) lcd.print("0");
  lcd.print(s);
}


void ShowDateTime(time_t t)
{
   lcd.setCursor(0, 1);
   ShowDate(t);
   lcd.print(" ");
   ShowTime(t);
}


void ShowSyncFlag()
{
  lcd.setCursor(10,0);
  if (gpsLocked) lcd.print('.');
  else lcd.print(' ');
}


void InitLCD()
{
  lcd.begin(16,2);                                      // initialize LCD
  lcd.print("W8BH clock");                              // write to top line
}


// ----------------------------------------------------------------------------------------------------
// TIME SYNCHONIZATION ROUTINES
// These routines will synchonize time with GPS and/or RTC as necessary
// Sync with GPS occur when the 1pps interrupt signal from the GPS goes high.
// GPS synchonization events are attempted every (SYNC_INTERVAL) seconds.
// If a valid GPS signal is not received within (SYNC_TIMEOUT) seconds, the clock with synchonized
// with RTC instead.  The RTC time is updated with GPS data once every 24 hours.


void SyncWithGPS()
{
  int y; byte h,m,s,mon,d; unsigned long age;
  gps.crack_datetime(&y,&mon,&d,&h,&m,&s,NULL,&age);     // get time from GPS
  if (age<1000)                                          // dont use data older than 1 second
  {
    setTime(h,m,s,d,mon,y);                              // copy GPS time to system time
    adjustTime(1);                                       // 1pps signal = start of next second
    syncTime = now();                                    // remember time of this sync
    gpsLocked = true;                                    // set flag that time is reflects GPS time
    UpdateRTC();                                         // update internal RTC clock periodically
    Serial.println("GPS synchronized");                  // send message to serial monitor
  }
}


void SyncWithRTC()
{
    setTime(RTC.get());                                 // set system time from RTC
    syncTime = now();                                   // and remember time of this sync event
    Serial.println("RTC synchronized");                 // send message to serial monitor
}


void SyncCheck()
// Manage synchonization of clock to GPS module
// First, check to see if it is time to synchonize
// Do time synchonization on the 1pps signal
// This call must be made frequently (keep in main loop)
```

```
{
  unsigned long timeSinceSync = now()-syncTime;         // how long has it been since last sync?
  if (pps && (timeSinceSync >= SYNC_INTERVAL))          // is it time to sync with GPS yet?
    SyncWithGPS();                                      // yes, so attempt it.
  pps = 0;                                              // reset 1-pulse-per-second flag, regardless
  if (timeSinceSync >= SYNC_TIMEOUT)                    // GPS sync has failed
  {
    gpsLocked = false;                                  // flag that clock is no longer in GPS sync
    SyncWithRTC();                                      // sync with RTC instead
  }
}



// -----------------------------------------------------------------------------------------------
// MAIN PROGRAM


void setup()
{
  InitLCD();                                           // initialize LCD display
  Serial1.begin(9600);                                 // set GPS baud rate to 9600 bps
  Serial.begin(9600);                                  // set serial monitor rate to 9600 bps
  PrintRTCstatus();                                    // show RTC diagnostics
  SyncWithRTC();                                       // start clock with RTC data
  enableInterrupt(PPS_PIN, isr, RISING);               // enable GPS 1pps interrupt input
}


void loop()
{
  FeedGpsParser();                                      // decode incoming GPS data
  SyncCheck();                                          // synchronize to GPS or RTC
  UpdateDisplay();                                      // if time has changed, display it
}


void isr()                                              // INTERRUPT SERVICE REQUEST
{
  pps = 1;                                              // Flag the 1pps input signal
}


void FeedGpsParser()
// feed currently available data from GPS module into tinyGPS parser
{
  while (Serial1.available())                           // look for data from GPS module
  {
     char c = Serial1.read();                           // read in all available chars
     gps.encode(c);                                     // and feed chars to GPS parser
  }
}


void UpdateDisplay()
//  Call this from the main loop
//  Updates display if time has changed
{
  time_t t = now();                                     // get current time
  if (t != displayTime)                                 // has time changed?
  {
      UpdateDST();                                      // check DST status
      ShowDateTime(LocalTime());                        // Display the new local time
      // ShowDateTime(t);                               // Display the new UTC time
      ShowSyncFlag();                                   // show if display is in GPS sync
      displayTime = t;                                  // save current display value
      PrintTime(t);                                     // copy time to serial monitor
  }
}
```