Build a GPS-controlled Clock

Bruce E. Hall, W8BH

Many of us have a fascination with time and time-keeping. Search the internet for clock projects and you will find thousands of interesting articles! I've always wanted a clock for my ham radio shack, using seven-segment LEDs. I wanted a clock that would simultaneously display local time and coordinated universal time (UTC). One can buy or build two clocks and put them side by side, but what's the fun in that? Here is an easy-to-build clock project that you can configure to your own needs. I'll show you, *step by step*, how to put the hardware and software together.

To keep things simple, I used breadboard-friendly hardware modules from one of my favorite online stores, adafruit.com:

#2771	Feather 32u4 Basic (Atmega32u4 microcontroller)	$20
#3028	DS3231 Precision RTC featherwing	$14
#2940	Female Headers, #3002 Male Headers	$2
#0878	Four Digit, 7-segment Red LED w/ I2C Backpack	$9 each
#0747	Ultimate GPS Breakout v3	$40
#1606	Full-size PCB Breadboard	$7

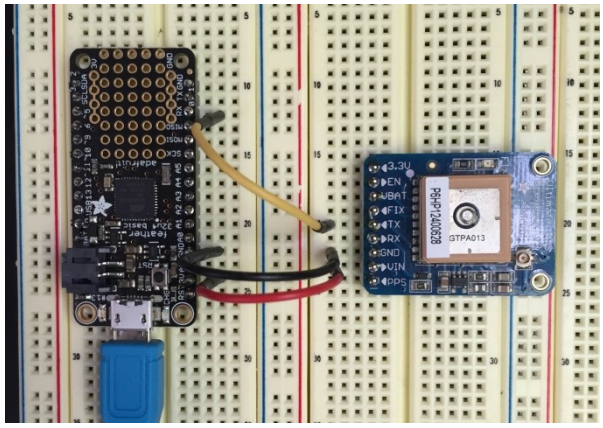
You will need a few other odds and ends: female/male headers, wire, solder, enclosure materials, and a power source. The LED display can be whatever color you like. A full display needs 6 of the display modules, but you can make a smaller display using 4, 3, 2, or even a single module.

A note about GPS modules: I like the Adafruit module, and keep a breadboard-ready unit for all of my testing. It's rugged and dependable. For my working clock I am using a Ublox NEO-6m module, about $10 on eBay. There are many GPS choices, and most will provide the time data that this project needs. Older units send their data at 4800 rather than 9600 baud – something to check if the unit doesn't seem to work.

STEP 1: ADDING A GPS MODULE

Start with the excellent tutorial on the microcontroller board, found at: https://learn.adafruit.com/adafruit-feather-32u4-basic-proto. This tutorial explains how to install Arduino on your computer and configure it for Adafruit's AVR boards. Try the 'blink' sketch to make sure everything is working. (Speaking of tutorials, there is also a tutorial at adafruit for another version of a GPS-enabled clock. Check it out here: https://learn.adafruit.com/arduino-clock)

Let's add some hardware. Connect the GPS module to the micro according the table below:



32u4 microcontroller	GPS module
3v3	Vin
GND	GND
Rx	Tx

That's it: only 3 wires. If you are breadboarding the modules, your setup should look something like the photo on the left.

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.

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 2: PARSING THE GPS DATA

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

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

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, we'll 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:

```
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
  }
}
```

The last thing to do is write a function for printing the time. The function gps.crack_datetime() returns the time as hours, minutes, seconds, etc:

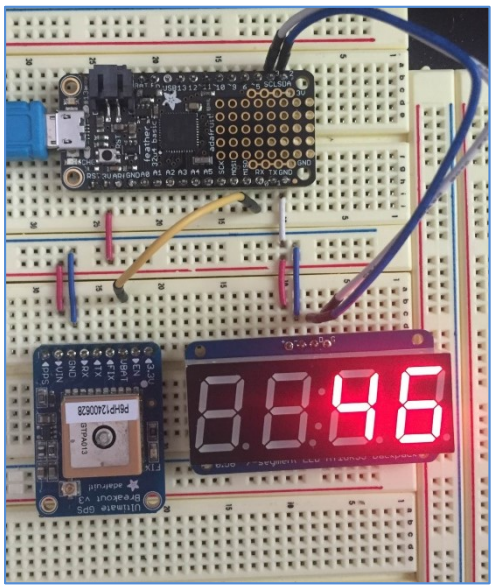
```
void printTime()
{
  byte hr,mn,sec;
  gps.crack_datetime(NULL,NULL,NULL,            // get time info, ignore date
      &hr,&mn,&sec,NULL,NULL);
  Serial.print(hr); Serial.print(":");          // print out time (crudely!)
  Serial.print(mn); Serial.print(":");
  Serial.print(sec); Serial.println(" 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 3: ADDING A DISPLAY

Once the GPS module is working, it is time to add a display module. This module from adafruit (https://www.adafruit.com/products/878) is a 4 digit display with an I2C interface. Connect it to the feather according to the table below:

32u4 microcontroller	LED module
3v3	"+"
GND	"-"
SDA	"D"ata
SCL	"C"lock



The LED module is a 5V unit, but will work on 3.3 volts with reduced brightness. I breadboarded mine alongside the GPS module, shown here. I've moved all of the power connections to a horizontal power bus in the middle of the photo. The blue and white wires at top right are the SDA and SCL lines, respectively.

The I2C backpack really simplifies the wiring: only two pins are required to drive all 33 LED segments (4 digits x 8 segments/digit… and a colon). On the software side, we will need to transform numbers into segments. You can program this yourself, which I did for my for LED interface project. But this project is all about simplicity, so I'm using the Adafruit library instead. Here is a quick sketch to test the setup:

```
#include <Wire.h>
#include <Adafruit_GFX.h>                       // needed for LEDBackpack library
#include "Adafruit_LEDBackpack.h"               // supports 7-segment LED Backpack

Adafruit_7segment  d1  = Adafruit_7segment();   // display object
int i = 0;


void setup()
{
  d1.begin(0x70);                               // initialize display object w/ I2C addr
}

void loop()
{
  d1.print(i++,DEC);                            // send count to display, then increment
  d1.writeDisplay();                            // show value on the LEDs
  delay(1000);                                  // and wait a sec before repeating
}
```

Run the sketch, and the LED display will count upwards from 0. The #includes are necessary for the backpack library. In setup(), specify the I2C address of the backpack module. This address is 0x70 by factory default. You can physically change the address in anything in the range 0x70 – 0x77 by soldering 1 or more pads together on the backside of the module. Loop() shows the incrementing value on the display. Change 'DEC' to 'HEX' if you'd prefer counting in hexadecimal!

STEP 4: DISPLAYING THE TIME

At this point, the GPS is connected, the display is connected, and we have code to control both. The only item left is to merge the code. Check out the following sketch:

```
#include <Wire.h>
#include <Adafruit_GFX.h>                       // needed for LEDBackpack library
#include "Adafruit_LEDBackpack.h"               // supports 7-segment LED Backpack
#include <TinyGPS.h>                            // arduiniana.org/libraries/TinyGPS/
#define SerialGPS Serial1                       // use hardware UART on 32u4

TinyGPS gps;                                    // the GPS parser object
Adafruit_7segment  d1  = Adafruit_7segment();   // display object w/ I2C addr


void setup()
{
  d1.begin(0x71);                               // initialize display object
  SerialGPS.begin(9600);                        // GPS communication at 9600 baud
}

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);
  int i= (hr*100) + mn;                         // convert hrs/mins to hhmm number
  d1.print(i,DEC);                              // send time value to displa
  d1.drawColon(sec % 2);                        // flash colon for seconds
  d1.writeDisplay();                            // show value on the LEDs
}
```

Remove all calls to the USB serial port, since the time will now display on LEDs. Setup() is otherwise just a combination of the two code pieces. And loop() is entirely unchanged. In printTime(), I added a line to turn the colon on and off. Using the modulus "%" operator on the seconds value is a tricky way of giving us a 0.5 Hz flash rate. Voilá, a working GPS clock!

I used this setup for a couple days, enjoying perfect timekeeping. But it's not quite perfect. For example, there are no leading zeros, so 00:05 UTC will confusingly display as "5". It takes a minute or more to get a GPS fix when you turn on the power. And if you use this clock indoors, a GPS signal may not be available. No GPS, no time (or even worse, wrong time)! To fix these issues, keep reading.

STEP 5: REMOVING GPS DEPENDENCE

Temporarily remove the wire from the Tx pin of the GPS module, and notice that the time display stops. Nothing happens in loop() if there isn't a steady stream of meaningful GPS data. We must be able to refresh the display without GPS. Start with the following modification:

```
#include <TimeLib.h>                            // github.com/PaulStoffregen/Time
prevDisplay = 0;                                // when the display was last updated

void loop()
{
  while (SerialGPS.available())                 // look for GPS data on serial port
  {
    int c=SerialGPS.read();                     // get one character at a time
    gps.encode(c);                              // and feed it to gps parser
  }
  if (now() != prevDisplay)                     // dont update display until time changes
  {
    prevDisplay = now();                        // save current time
    printTime();                                // display time
  }
}
```

This code uses the now() function of the time library, which returns the current unix time in seconds. Loop continues to send GPS data to the parser, but display updates no longer depend on a steady data stream. With this new code, printTime() will be called every second. One more change is

needed, since printTime() uses parsed GPS data internally. Let's remove that, and get the time information from the time library instead:

```
#include <TimeLib.h>                            // github.com/PaulStoffregen/Time
#include <Wire.h>
#include <Adafruit_GFX.h>                       // needed for LEDBackpack library
#include "Adafruit_LEDBackpack.h"               // supports 7-segment LED Backpack
#include <TinyGPS.h>                            // arduiniana.org/libraries/TinyGPS/
#define SerialGPS Serial1                       // use hardware UART on 32u4

TinyGPS gps;                                    // the GPS parser object
time_t prevDisplay = 0;                         // when the digital clock was displayed
Adafruit_7segment  d1  = Adafruit_7segment();   // display object w/ I2C addr


void setup()
{
  d1.begin(0x71);                               // initialize display object
  SerialGPS.begin(9600);                        // GPS communication at 9600 baud
}

void loop()
{
  while (SerialGPS.available())                 // look for GPS data on serial port
  {
    int c=SerialGPS.read();                     // get one character at a time
    gps.encode(c);                              // and feed it to gps parser
  }
  if (now() != prevDisplay)                     // dont update display until time changes
  {
    prevDisplay = now();                        // save current time
    printTime();                                // send time to USB serial port
  }
}

void printTime()
{
  int i = hour()*100 + minute();                // "11:32" converted to number "1132"
  d1.print(i,DEC);                              // send time value to display
  d1.drawColon(second() % 2);                   // flash colon for seconds
  d1.writeDisplay();                            // show value on the LEDs
}
```

What happens? Time is displayed with or without GPS input, but time is never synchronized with GPS either! When you apply power, time starts at 0 (00:00 on 1/1/1970). At first glance, this doesn't seem helpful at all. But all we need is a way to synchronize the time library with external sources. Fortunately there is a way: the built-in routine setSyncProvider(). Call this routine with your external time source, and the system time is periodically updated. The default update interval is once every 5 minutes. Try the following code:

```
#include <TimeLib.h>                            // github.com/PaulStoffregen/Time
#include <Wire.h>
#include <Adafruit_GFX.h>                       // needed for LEDBackpack library
#include "Adafruit_LEDBackpack.h"               // supports 7-segment LED Backpack
#include <TinyGPS.h>                            // arduiniana.org/libraries/TinyGPS/
#define SerialGPS Serial1                       // use hardware UART on 32u4

TinyGPS gps;                                    // the GPS parser object
time_t prevDisplay = 0;                         // when the digital clock was displayed
Adafruit_7segment  d1  = Adafruit_7segment();   // display object w/ I2C addr


void setup()
{
```

```
  d1.begin(0x71);                                 // initialize display object
  SerialGPS.begin(9600);                          // GPS communication at 9600 baud
  setSyncProvider(timeSync);                      // specify time setting routine
}

void loop()
{
  while (SerialGPS.available())                   // look for GPS data on serial port
  {
    int c=SerialGPS.read();                       // get one character at a time
    gps.encode(c);                                // and feed it to gps parser
  }
  if (now() != prevDisplay)                       // dont update display until time changes
  {
    prevDisplay = now();                          // save current time
    printTime();                                  // send time to USB serial port
  }
}

void printTime()
{
  int i = hour()*100 + minute();                  // "11:32" converted to number "1132"
  d1.print(i,DEC);                                // send time value to display
  d1.drawColon(second() % 2);                     // flash colon for seconds
  d1.writeDisplay();                              // show value on the LEDs
}

time_t timeSync()
{
  tmElements_t tm;
  time_t t = 0;
  int yr;
  unsigned long fixAge;
  gps.crack_datetime(&yr, &tm.Month, &tm.Day,     // get UTC time from GPS
     &tm.Hour, &tm.Minute, &tm.Second,
     NULL, &fixAge);
  if (fixAge<2000)                                // GPS has current data
  {
    tm.Year = yr-1970;                            // convert calendar years to unix years
    t = makeTime(tm);                             // convert to time_t
  }
  return t;                                       // return unix time
}
```

The code is getting a bit longer, but not too complicated. Look at the new timeSync() routine. It is called once every 5 minutes to synchronize the time library with GPS data. gps.crack_datetime() gives us all the time elements we need. If this data was collected within the last 2000 milliseconds, it is used to update the system time. The routine makeTime() is used to convert the time elements into unix time. MakeTime expects the year to be a unix year (0=1970, 1=1971, etc) instead of the calendar year, so we must do a simple conversion before calling it.

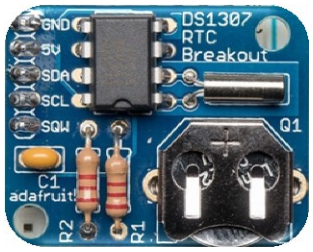
Try running the sketch. After 5 minutes, if GPS has a fix, the time will be updated. Disconnect the GPS for a while, and the clock still works! We now have a clock that can function without GPS, but will update with GPS when available.

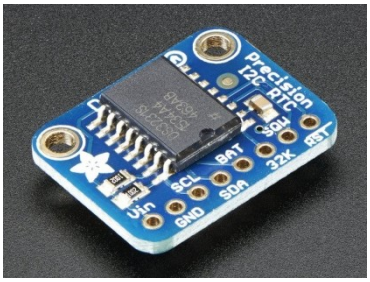
STEP 6: ADDING A REAL-TIME CLOCK

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.

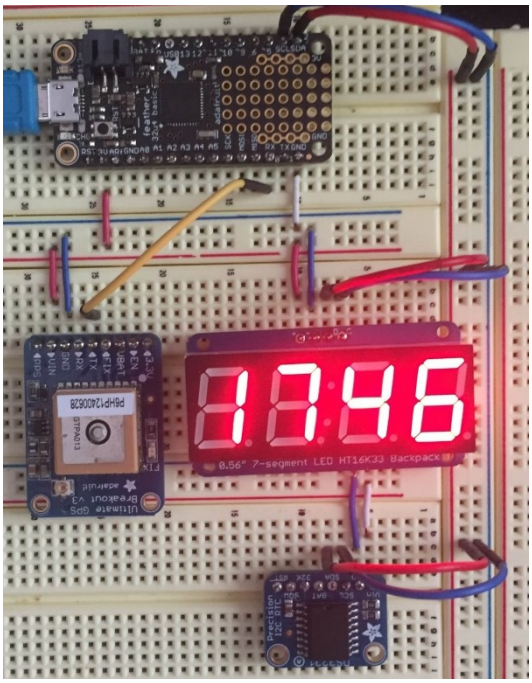
Enter the real-time clock. The most-common RTC is the DS1307, pictured here. There are many code libraries available for the '1307. Note the battery clip. With a coin cell will it will keep track of time for up to 5 years. The DS1307 uses a 32 kHz crystal (the silver cylinder-shaped component below the 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 is just like the LED display: two wires for I2C communications and two wires for power. On my breadboard (see photo at right) I've added the DS3231 module below the display. The two I2C lines are connected to a vertical bus strip on the right.

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, modify the timeSync() routine to read the RTC if GPS is not available. But if GPS is available, use it and update the RTC time:

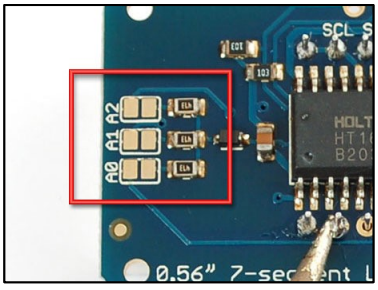
```
time_t timeSync()
{
  tmElements_t tm;
  time_t t = 0;
  int yr;
  unsigned long fixAge;
  gps.crack_datetime(&yr, &tm.Month, &tm.Day,     // get UTC time from GPS
     &tm.Hour, &tm.Minute, &tm.Second,
     NULL, &fixAge);
  if (fixAge>2000)                                // no good GPS data
    t = RTC.get();                                // ..so use RTC time instead
  else
  {                                               // GPS has good data
    tm.Year = yr-1970;                            // convert calendar years to unix years
    RTC.write(tm);                                // save GPS time to RTC
    t = makeTime(tm);                             // convert to unix time
  }
  return t;                                       // return unix time
}
```

This small modification is all that's required to add a real-time clock. Now, when you apply power, time is immediately taken from the RTC, and the clock will continue be synchronized by the RTC until GPS is available.

STEP 7: SUPERSIZE THE DISPLAY

With the confidence gained from a working prototype, it was time to build what I wanted: a full display of the time and date for local time and UTC. How many modules would that require?

Module 1: Local HH:MM	Module 2: Local SS + AM/PM	Module 3: Local DD.YY
Module 4: UTC HH:MM	Module 5: UTC SS	Module 6: UTC DD.YY



That's one big display! I decided to arrange my 6 modules as two rows of three, as above. All of the modules are connected in parallel. You may connect up to 8 modules, depending on what data you want to display. Each module must have a different I2C address, which is configured by

soldering one or more of the address pads, marked A0/A1/A2, on the back of the module. For a module with no solder bridges, the I2C address is 0x70. The table at left shows the I2c addresses and solder bridges needed for 6 different modules.

I2C address	A0	A1	A2
0x70			
0x71	X		
0x72		X	
0x73	X	X	
0x74			X
0x75	X		X

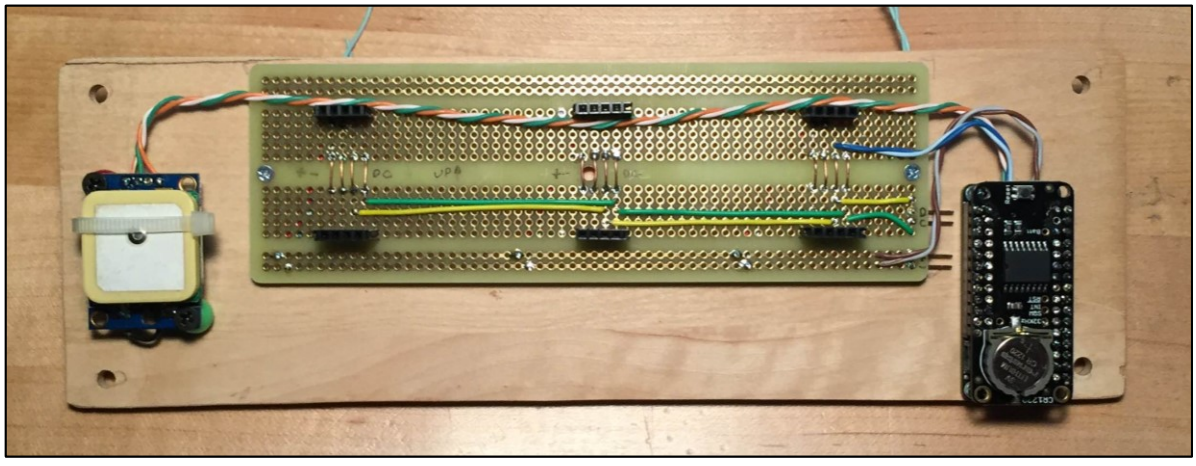
The method of wiring the modules depends on the size of the display and how you want them arranged. For my display, I found that 6 modules fit perfectly on a full size protoboard. I soldered female headers to the protoboard, and then plugged in the modules. If you prefer, you can solder the modules directly to the board. The protoboard makes it easy to run parallel connections between the modules.

You must consider the power required to run your display. In step 2, when we added a single display module, we attached it the 3.3 volt supply from the feather board. The on-board regulator is capable of supplying 500mA for short periods of time. If each segment is at full power, drawing 20mA, a single module could draw up to 33 x 20 = 660mA of current. Six modules, drawing up to 4 amps total, would quickly overwhelm the regulator and most power supplies. (USB power from a computer is typically only 500mA.)

There are several ways to reduce power consumption, and reduce the load on the 3.3V regulator. First, bypass the regulator, and supply 5V power directly from the USB jack. The feather board has a pin marked Vusb specifically for this purpose. Next, reduce the display brightness in software. At 5V full brightness, these modules are unnecessarily bright. They are perfectly readable at even minimum brightness. Lastly, make sure your power supply can supply sufficient current. My 5V/2A supply works well.

STEP 8: PUTTING IT ALL TOGETHER

Once you have your display module built on a protoboard, there isn't much left to do. My enclosure is super-simple: a used a 3" x 10" piece of this wood for the back, and a matching



piece of 1/8” acylic sheet for the front. I screwed down the feather, display protoboard, and GSP module to the wood back using #2 wood screws. I mounted a DS3231 RTC “featherwing” on top of the feather. The RTC featherwing provides an additional set of through-holes, to which I soldered solid-wire jumpers to the GPS and display modules. After verifying the clock, I used four long 8-32 bolts to secure the front and back together, so that the display modules were snug against the back of the acrylic. Two small screw eyes and a wire hold the clock to my pegboard wall. Current consumption at moderate display brightness is less than 250mA at 5V.

The complete source code is printed below. I couldn’t resist adding a few bells and whistles:

- startup screen, to verify microcontroller and display
- setting the display brightness
- routines for 12hr local and 24hr UTC time display
- automatic daylight saving time correction.
- diagnostic output on the USB serial port
- led indicator for GPS lock
- decreased time-to-sync & RTC writes

My goal was to create a “plug-and-forget” wall clock. 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!

SOURCE CODE:

```
/*
 *  MyFeatherGPS.ino
 *
 *  GPS-driven, digital clock with local and UTC time on 7-segment LED display.
 *
 *  Author:     Bruce E. Hall, W8BH
 *  Date:       01 May 2016
 *  Hardware:   ATMEL 32u4 microcontroller on Adafruit "Feather" board
 *              Maxim DS 3212 real-time-clock on Adafruit Featherwing
 *              4-digit 7-segment displays mounted on Adafruit I2C backpacks
 *              GPS module (Adafruit Ultimate GPS or UBLOX Neo-6M module)
 *  Software:   Arduino 1.6.5
 *  Code size:  about 17200 bytes
 *
 *  The system keeps time in UTC, supplied by GPS and/or RTC when GPS is unavailable.
 *  Local time automatically tracks daylight saving time for selected time zone.
 *  USB serial port outputs time and GPS/RTC status messages.
 *  Dual display (local & UTC) very handy for a ham radio shack.
 *
 */

#include <TimeLib.h>                            // github.com/PaulStoffregen/Time
#include <TinyGPS.h>                            // arduiniana.org/libraries/TinyGPS/
#include <DS1307RTC.h>                          // github.com/PaulStoffregen/DS1307RTC
#include <Wire.h>                               // needed for I2C
#include <Adafruit_GFX.h>                       // needed for LEDBackpack library
#include "Adafruit_LEDBackpack.h"               // supports 7-segment LED Backpack


#define TIME_SYNC_INTERVAL  180                 // 180 seconds = 3 min between GPS time sync attempts
#define CYCLES_PER_RTC_SYNC  20                 // 20 cycles = 1 hr between gps->rtc sync


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

                                                // Select timeZone offset from UTC:
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)


// I2C address of each display unit.  The default address is 0x70
#define MIN_BRIGHTNESS    0x01                  // suitable for dark room
#define MAX_BRIGHTNESS    0x0F                  // really, really bright
#define HHMM_ADDRESS      0x70                  // local time Hour:Minute display
#define SECONDS_ADDRESS   0x71                  // local time Seconds & AM/PM display
#define DATE_ADDRESS      0x72                  // local date Month.Day display
#define UTCTIME_ADDRESS   0x73                  // UTC time Hour:Minute display
#define INFO_ADDRESS      0x74                  // UTC time units = "utc"
#define UTCDATE_ADDRESS   0x75                  // UTC date Month.Day display

// define some Alpha characters on the seven segment displays
#define smallB            0x7C                  // 'b'
#define smallR            0x50                  // 'r'
#define smallU            0x1C                  // 'u'
#define smallT            0x78                  // 't'
#define smallC            0x58                  // 'c'
```

```
#define largeA            0x77                  // 'A'
#define largeH            0x76                  // 'H'
#define largeE            0x79                  // 'E'
#define largeP            0x73                  // 'P'
#define largeL            0x38                  // 'L'
#define dash              0x40                  // '-'
#define blank             0x00                  // ' '
#define DP                0x80                  // '.'

// Create display objects.
Adafruit_7segment  d1  = Adafruit_7segment();
Adafruit_7segment  d2  = Adafruit_7segment();
Adafruit_7segment  d3  = Adafruit_7segment();
Adafruit_7segment  d4  = Adafruit_7segment();
Adafruit_7segment  d5  = Adafruit_7segment();
Adafruit_7segment  d6  = Adafruit_7segment();

TinyGPS gps;                                    // gps string-parser object
time_t prevDisplay = 0;                         // when the digital clock was displayed
int  cyclesUntilSync = 0;                       // counts number of cycles until RTC sync
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


void setDisplayBrightness (int b)               // 1=min, 15=max
{
  d1.setBrightness(b);
  d2.setBrightness(b);
  d3.setBrightness(b);
  d4.setBrightness(b);
  d5.setBrightness(b);
  d6.setBrightness(b);
}

void updateDisplay()
{
  d1.writeDisplay();
  d2.writeDisplay();
  d3.writeDisplay();
  d4.writeDisplay();
  d5.writeDisplay();
  d6.writeDisplay();
}

void clearDisplay()
{
  d1.clear();
  d2.clear();
  d3.clear();
  d4.clear();
  d5.clear();
  d6.clear();
  updateDisplay();
}

void initDisplay()
{
  d1.begin(HHMM_ADDRESS);                       // init I2C backpack objects
```

```
  d2.begin(SECONDS_ADDRESS);
  d3.begin(DATE_ADDRESS);
  d4.begin(UTCTIME_ADDRESS);
  d5.begin(INFO_ADDRESS);
  d6.begin(UTCDATE_ADDRESS);
  clearDisplay();
                                          // put startup message on display...
  d2.writeDigitRaw(0,smallB);             // 'b'
  d2.writeDigitRaw(1,smallR);             // 'r'
  d2.writeDigitRaw(3,smallU);             // 'u'
  d2.writeDigitRaw(4,smallC);             // 'c'
  d3.writeDigitRaw(0,largeE);             // 'E'

  d5.writeDigitRaw(0,largeH);             // 'H'
  d5.writeDigitRaw(1,largeA);             // 'A'
  d5.writeDigitRaw(3,largeL);             // 'L'
  d5.writeDigitRaw(4,largeL);             // 'L'
  updateDisplay();
  delay(5000);                            // let me admire my work
  clearDisplay();
}

void setup()
{
  initDisplay();                          // initialize 7segment display objects
  setDisplayBrightness(MIN_BRIGHTNESS+3); // conserve display power consumption
  Serial.begin(9600);                     // start USB serial output
  SerialGPS.begin(9600);                  // open UART connection to GPS
  //manualTimeSet();                      // allow temporary time setting before gps kicks in
  setSyncProvider(timeSync);              // specify time setting routine
  setSyncInterval(TIME_SYNC_INTERVAL);    // periodically get time from gps/rtc
}



//
//  Here are the basic clock routines:
//  "Loop" updates the display when a new time is available
//  "TimeSync" keeps system time in UTC, and updates it with GPS data.
//  If GPS is unavailable, time is updated from RTC chip instead.
//

void loop()
{
  while (SerialGPS.available())           // look for GPS data on serial port
  {
    int c=SerialGPS.read();               // get one character at a time
    gps.encode(c);                        // and feed it to gps parser
  }
  if (timeStatus()!= timeNotSet)          // wait until time is set by sync routine
  {
    if (now() != prevDisplay)             // dont update display until time changes
    {
      prevDisplay = now();                // save current time
      updateDST();                        // check DST status
      displayLocalTime();                 // show local time on LEDs
      displayUTC();                       // show UTC time on LEDs
      printTime(prevDisplay);             // send time to USB serial port
    }
  }
```

```
}

time_t timeSync()
// will change global variable "gpsLocked" according to ability to sync with GPS
// returns time as UTC
{
  tmElements_t tm;
  time_t t = 0;
  int yr;
  unsigned long fixAge;

  gpsLocked = false;                              // assume failure to read GPS
  gps.crack_datetime(&yr, &tm.Month, &tm.Day,     // get UTC time from GPS
     &tm.Hour, &tm.Minute, &tm.Second,
     NULL, &fixAge);

  if (fixAge==TinyGPS::GPS_INVALID_FIX_TIME)      // GPS online but no satellite fix
  {
    Serial.println("\nNo GPS fix, using RTC");
    t = RTC.get();                                // ...so get from RTC instead
  }
  else if (fixAge > 2000)                         // GPS is offline
  {
    Serial.println("\nGPS offline, using RTC");
    t = RTC.get();                                // ...so get from RTC instead
  }
  else                                            // GPS time is good
  {
    gpsLocked = true;                             // flag success
    tm.Year = yr-1970;                            // convert calendar years to unix years
    Serial.print("\nGPS time OK, ");
    Serial.print(gps.satellites());
    Serial.print(" sats. ");
    if (cyclesUntilSync <=0)                      // time to update RTC yet?
    {
      if (RTC.write(tm))                          // update real-time clock chip
        Serial.print("RTC updated.");
      else Serial.print("RTC update failed.");
      cyclesUntilSync = CYCLES_PER_RTC_SYNC;      // reset counter
    }
    Serial.println();
    t = makeTime(tm);                             // convert to time_t
  }
  cyclesUntilSync --;                             // count-down another cycle
  return t;                                       // return unix time in UTC
}

void manualTimeSet()
// Use this routine to manually set the clock to a specific UTC time.
// Set each tm element to desired value
// Since time is automatic set from gps, this routine is mainly for debugging
{
  tmElements_t tm;
  tm.Year   = 2016 - 1970;                       // Year in unix years
  tm.Month  = 11;
  tm.Day    = 6;
  tm.Hour   = 5;
  tm.Minute = 59;
  tm.Second = 30;
  RTC.write(tm);                                 // set RTC to desired time
```

```
}



//
// The following routines support automatic daylight saving time adjustment
//
// Daylight Saving Time constants:
// Please set according to current DST transition times
//

#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()
{
  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;
}



//
//  The following routine support time display
//

void printTime(time_t t)
// send time string to serial monitor
{
  Serial.print(hour(t));
  printDigits(minute(t));
  printDigits(second(t));
  Serial.print(" ");
  Serial.print(month(t));
  Serial.print("/");
  Serial.print(day(t));
  Serial.print("/");
  Serial.print(year(t));
  Serial.println(" UTC");
}

void printDigits(int digits)
// utility function for digital clock display: prints preceding colon and leading 0
{
  Serial.print(":");
  if(digits < 10)
    Serial.print('0');
  Serial.print(digits);
}

void displayUTC()
{
  int displayValue = hour()*100 + minute();       // "11:32" converted to number "1132"
  int dateValue = month()*100 + day();            // "04/08" converted to number "408"

  d4.print(displayValue, DEC);                    // Send time/date value to display buffers
  d6.print(dateValue, DEC);

  if (displayValue<1000)                          // Add zero padding
    d4.writeDigitNum(0, 0);                       // Ex: 123 --> "01:23"
  if (displayValue<100)
    d4.writeDigitNum(1, 0);                       // Ex:  23 --> "00:23"
  if (displayValue<10)
```

```
    d4.writeDigitNum(3, 0);                    // Ex"   3 --> "00:03"
// Add static indicators
  d5.writeDigitRaw(0,smallU);                  // 'u'
  d5.writeDigitRaw(1,smallT);                  // 't'
  d5.writeDigitRaw(3,smallC);                  // 'c'
  d4.drawColon(1);                             // add colon to time
  d6.displaybuffer[1] += DP;                   // put decimal point between month.day
  if (gpsLocked)                               // time sync with GPS recently?
    d5.displaybuffer[3] += DP;                 // ... put Dp after utc: "utc."
  updateDisplay();                             // show new data on display
}

void displayLocalTime()
{
  time_t t = localTime();                      // convert system time (UTC) to local time
  int hours = hourFormat12(t);                 // use hour() for 24HR time
  int seconds = second(t);
  int displayValue = hours*100 + minute(t);    // "11:32" converted to number "1132"
  int dateValue = month(t)*100 + day(t);       // "04/08" converted to number "408"

  d1.print(displayValue, DEC);                 // Send time/date value to display buffers
  d2.print(seconds*100, DEC);
  d3.print(dateValue, DEC);

  if (seconds<10)                              // add zero padding to seconds display
     d2.writeDigitNum(0,0);
  if (seconds==0)
     d2.writeDigitNum(1,0);
  d1.drawColon(1);                             // to blink colon, use: '(seconds % 2 == 0)'
  d2.writeDigitRaw(4,blank);
  if (isAM(t))
      d2.writeDigitRaw(3,largeA);              // 'A'
  else d2.writeDigitRaw(3,largeP);             // 'P'
  d3.displaybuffer[1] += DP;                   // put decimal point between month.day
  updateDisplay;                               // show new data on display
}
```