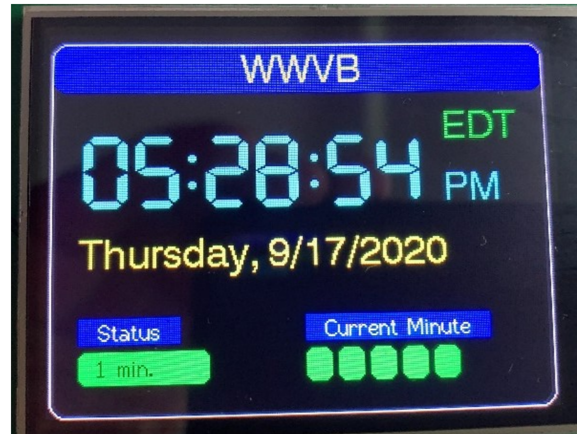
WWVB clock

Build a radio-controlled clock using a microcontroller and LCD display.

Bruce E. Hall, W8BH



Introduction.

Do you have a radio-controlled clock at home? You know, the ones that set themselves? If you live in the eastern US like me, you might also know they don't always work. Mine won't update unless I take it off the wall and put it by a window overnight. Which defeats the purpose of a self-setting clock…

So why would I want to *build* one? There are other choices for precision timekeeping, like GPS and NTP. For some locations, WWVB indoor reception is significantly better than GPS. No clear view of the sky is required. WWVB is well suited for low-cost, low-power, battery operation. And, unlike NTP, no internet access is required. Finally, for a dyed-in-the-wool ham operator like me, getting information by radio has a certain appeal that's difficult to describe.

This article describes how radio-controlled clocks work and gives you enough information to build your own. For my clock I chose a STM32 "Blue Pill" microcontroller, a 2.8" 320x240 pixel LCD display, and a WWVB radio module. 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.

I will start by creating small, easily-understandable, and usable bits of code. Each step builds on the preceding steps until the application is complete. This is how I learn best. The source code for each step and final application is on my GitHub account.

WWVB.

These clocks receive their signals from the [WWVB radio station] near Fort Collins, Colorado. WWVB began service on the 4th of July, 1963 and has been broadcasting time signals continuously since then on a frequency of 60 kHz. This frequency Is considered at the low end of “long wave” radio (compared to AM radio in “medium wave” and international broadcasting in “short wave”). Long wave uses ground-wave propagation for communications up to 1200 miles from the transmitter. Time stations around the world use long-wave, as do marine and aeronautical beacons. A list of time signal stations is found here.

The WWVB signal has not changed since 1965. Time information is sent as a string of bits, at a rate of one bit per second, such that the time, date, and other related data, such as daylight savings information, is transmitted once every minute. Each data bit can be a ‘1’, a ‘0’, or a special marker bit. The marker bits, which help the receiver synchronize and verify the data, are sent on the 1st, 10th, 20th, 30th, 40th, 50th, and 60th second of each minute.

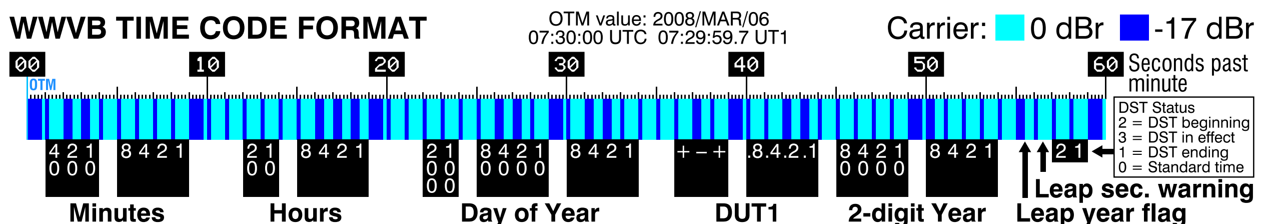
Each data field is sent in binary-coded decimal format, with zeroes separating the digits. For example, suppose the time is 11:42. The minute field “42” is encoded as a binary four = “100”, followed by a zero, followed by a binary two = “0020”. The full minute is encoded as follows according to the table at right.

The bits are sent by modulating the carrier. Think of the full carrier is as unmodulated signal. The carrier is modulated by temporarily reducing the output power from 70kW to 1.4kW, a 17 decibel reduction: 10 * log(1.4/70) = -17. The amount of time spent in reduced carrier determines each bit’s value:

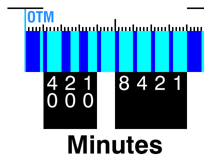
Modulation Time	Bit Value
200 mS	0
500 mS	1
800 mS	Marker

Bit Number	Description
0	Marker
1-3	Minutes, tens place
4-8	Minutes, ones place
9	Marker
10-13	Hours, tens place
14-18	Hours, ones place
19	Marker
20-23	Day of year, hundreds place
24-28	Day of year, tens place
29	Marker
30-33	Day of year, ones place
34-38	UT1 offset sign
39	Marker
40-43	UT1 offset value
44-48	Year, tens place
49	Marker
50-53	Year, ones place
54-55	Leap year indicator
56	Leap second indicator
57-58	Daylight savings indicator
59	Marker

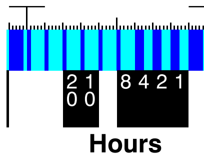
Consider the following example minute, taken from the WWVB Wikipedia page:



It's confusing at first. The dark blue areas indicate when the WWVB carrier is modulated/reduced. Notice that the width of the first dark blue line is wide, denoting the first marker bit. Find this wide line, and then notice similar wide lines at 10 second intervals. All minutes begin and end with the marker.



Let's decode the minutes. Look at the example, and notice that the data for "minutes" is located between two marker bits. The thinnest blue lines have the value 0 and the medium-thickness lines have the value 1. From left to right I see the value "01100000". The first three bits "011" = decimal 3, give us the tens place and the bottom 4 bits "0000" = decimal 0 give us the ones place. The minutes value is 3 and 0 = "30". This method of encoding numbers is called binary-coded decimal, or BCD.



Just for fun let's decode the hours, too. Count the dark blue lines between the markers, left to right, substituting 0 for thin lines and 1 for medium lines. Remember, don't include the fat marker bits. I count "000000111". Two of the first five bits, binary"00" = 0 represent the tens place. The bottom four bits, binary "0111" = 7 represent the ones place. Hours is therefore equal to 07, and the time is 07:30

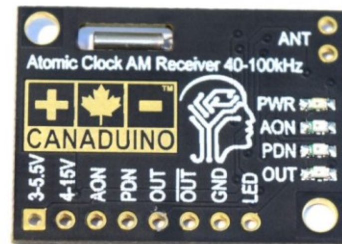
Seconds are not encoded; the top of the minute ":00" corresponds to the start of the first marker bit. There are 60 bits of data at 1 bit per second, therefore it takes 60 seconds to transmit each minute.

But wait a "second": if it takes 60 seconds to send the data, aren't you a minute late by the time you've received all the data? Answer: yes, you are. Your software must correct for the delay.

The WWVB receiver.

Radio-controlled clocks are common, but receiver modules are much more difficult to find. Items such as the Sparkfun NIST Receiver kit, a repackaged CMax CMMR-6 module, are no longer for sale. Some have resorted to buying a cheap clock and 'harvesting' the receiver. But a few options do exist. Consider the inexpensive 2-pack receiver module from LStech for $13, and their clock module for $16.

Currently, the best choice seems to be the "Canaduino" 60kHz Atomic Clock Receiver Module, available at Amazon or directly from Universal-Solder for around $15. This superb, recently redesigned module, based on the MAS6180C chip, comes with its own onboard voltage regulator and diagnostic LEDs, perfect for the experimenter and hobbyist. If you are serious about building a WWVB clock, this is the one to buy.



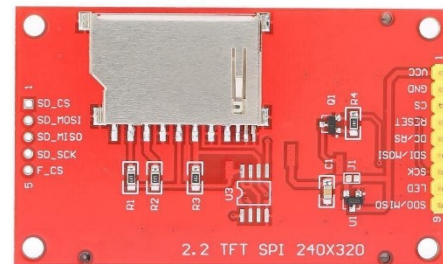
To use the receiver, orient the antenna just that it is horizontal and broadside (perpendicular) to the direction from your location to the transmitter in Fort Collins CO. Next, make sure that there are no electronic devices nearby. For example, my receiver does not work when placed near my computer monitor. Use a battery or low-noise supply for your power. Finally, use the receiver later at night or early in the morning, avoiding the afternoon. If you live near Ft. Collins these steps may not be necessary, but they'll help avoid frustration and failure here in the Eastern US.

Hook up the receiver, as indicated, without any other hardware. The Power and AON LED’s should immediately light. If they don’t recheck your power connections. After a few minutes, the OUT light should flicker, signaling that the device has finished its power-up sequence and is now sending data.

Sometimes you will notice the OUT light is steady on. This indicates that the receiver is working but not finding any modulated signal. Try reorienting your antenna, removing nearby electronics, etc, to improve the signal output. The OUT light should irregularly blink off roughly once a second. Very quick flashes (<100mS) represent noise, not data.

The Hardware.

Time to build! I assume that you are familiar with breadboarding and can supply your board with 3.3V. For my clock I am using a “Blue Pill” microcontroller and a 320x240 TFT display.



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.

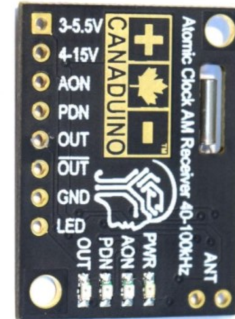
There are 5 pins on the display that connect to pins on the Blue Pill, and 3 pins that are power/ground related. The following table details the connections:

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

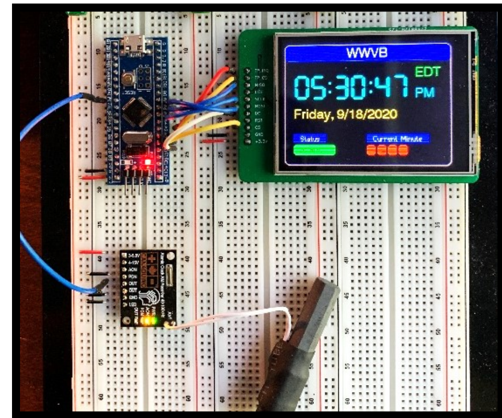
Connect the wires and apply power. Make sure the backlight is ON – if not, immediately disconnect and check your wiring. The most common failure at this point is improper wiring.

Next, hook up the 8-pin Canaduino Module as follows:

Label	Connects To:	Function
3-5.5V	Vcc bus (3.3V)	Power
4-15V	(no connection)	Unregulated Power
AON	(no connection)	AGC enable
PDN	Ground	Power Down
OUT	Blue Pill, pin PA12	Receiver Data output
~OUT	(no connection)	Inverted Data output
GND	Ground	Device Ground
LED	Vcc bus (3.3V)	LED Enable



The illustration at right shows the author's breadboard setup. Note the gray ferrite rod antenna at the bottom, which attaches to the receiver module. A single blue wire carries data from the receiver module to the Blue Pill microcontroller above it. Several wires carry data and control signals from the microcontroller to the display.



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/master/STM32/package_stm_index.json

For TFT support I am using "TFT_eSPI" by Bodmer, version 2.2.14. 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 0: 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.

Step 1: Counting pulses.

As noted above, WWVB sends time information by varying the length of its carrier modulation. So, to decode the data we will need a way of measuring the length of time the receiver’s OUT line is low. Amazingly, the Arduino environment provides a routine tailor-made for this job: “pulseIn”. This routine monitors a microcontroller pin and returns the amount of time, in microseconds, that the line is pulsed (high or low). An additional parameter is provided to allow for a timeout, in case a pulse does not occur within a specific amount of time:

```
pulseWidth = pulseIn(RADIO_OUT,LOW,2000000);    // get width of low-going pulse in uS
```

In the code above, the microcontroller pin called “RADIO_OUT” is monitored for a low-going pulse, and the duration of the pulse is returned to the variable pulseWidth. The function returns with a 0 if there is no pulse within 2 seconds (2,000,000 microseconds).

Using pulseIn, here is a simple sketch to monitor Canaduino output and display the width of each pulse:

```
#include <TFT_eSPI.h>
#define RADIO_OUT          PA12
#define TITLE "WWVB Step #1: SHOW PULSE WIDTH"

TFT_eSPI tft = TFT_eSPI();

void setup() {
  tft.init();
  tft.setRotation(1);                                    // portrait screen orientation
  tft.fillScreen(TFT_BLACK);                             // start with blank screen
  tft.setTextColor(TFT_YELLOW);
  tft.drawString(TITLE,10,10,2);                         // show title at top of screen
}

void loop() {
  int pulseWidth = pulseIn(RADIO_OUT,LOW,2000000);       // get width of pulse in uS
  pulseWidth /= 1000;                                    // convert uS to mS
  if (pulseWidth>90) {                                   // ignore noise
     tft.fillRect(50,50,100,20,TFT_BLACK);               // erase previous value
     tft.drawNumber(pulseWidth,50,50,4);                 // show pulse width on screen
  }
}
```

Make sure your Canaduino is receiving a signal (the OUT LED is flashing) and run this sketch. The width of each pulse should be displayed. Remember that 0 bits are around 200 mS in duration, ‘1’ bits are 500 mS long, and marker bits are 800 mS. Do you see values similar to this? On my system, actual values are around 180 mS for zero bits, 480 for one bits, and 770 for markers. Bits received on my system are typically tens of milliseconds shorter than the specified value. If pulse widths are displayed on your screen, continue to step 2.

Step 2: Decoding Bits

If we can display the pulse width, we can also display the bit value. For example, consider this:

```
If (pulseWidth==200) bit = 0;
else if (pulseWidth==500) bit = 1;
else if (pulseWidth==800) bit = marker;
else bit = error;
```

Would it work? All bits would likely be errors, since the received pulse width is not going to be exactly the right length. We need to give some leeway. The following code works quite nicely:

```
#include <TFT_eSPI.h>
#define RADIO_OUT          PA12
#define TITLE "WWVB Step #2: DECODING BITS"

TFT_eSPI tft = TFT_eSPI();

void setup() {
  tft.init();
  tft.setRotation(1);                                    // portrait screen orientation
  tft.fillScreen(TFT_BLACK);                             // start with blank screen
  tft.setTextColor(TFT_YELLOW);
  tft.drawString(TITLE,10,10,2);                         // show title at top of screen
}
```

```
void loop() {
  int pulseWidth = pulseIn(RADIO_OUT,LOW,2000000);      // get width of pulse in uS
  pulseWidth /= 1000;                                   // convert uS to mS
  if (pulseWidth>90) {                                  // ignore noise
     tft.fillRect(50,50,140,20,TFT_BLACK);              // erase previous value
     tft.drawNumber(pulseWidth,50,50,4);                // show pulse width on screen
     if (pulseWidth<300)                                // bit 0?
       tft.drawString("'0'",100,50,4);
     else if ((pulseWidth>330) && (pulseWidth<600))     // bit 1?
       tft.drawString("'1'",100,50,4);
     else if ((pulseWidth>630) && (pulseWidth<900))     // marker?
       tft.drawString("Mrk",100,50,4);
     else tft.drawString("Err",100,50,4);               // none of the above
  }
}
```

Try this simple sketch to see the receiver output and decode the incoming bits. The rest of the clock is just 'a simple matter of programming'! There are two drawbacks to using a pulseIn routine like this. First, pulseIn is blocking. In other words, the microcontroller can't do anything else until the incoming pulse is completed. A 0.8 sec. marker pulse creates a substantial delay.

The second issue is susceptibility to receiver noise. Imagine that a 500 mS '1' bit is being sent, and a short pulse of static occurs in the middle of the bit. Our receiver might decode this as two, separate ~250 mS pulses or two '0' bits. Or worse yet, a 320 mS error bit and a 180 mS '0' bit. Noise/Static and low signal are issues at my location, so I ditched pulseIn for another method: intermittent sampling.

With the sampling approach, the input signal is sampled at a high rate, say every 1-10 milliseconds, and if the signal is low-active, a counter is incremented. At the end of the timing period (one second), the number of low-signal intervals are added up for the pulse count. If a small amount of noise alters a few samples, the final result will not be altered. For example, consider a 180 mS '0' bit. If we are sampling that pulse at 10mS intervals, the result should be 18 samples of low output:

Sampler lo-lo-lo-lo-lo-lo-lo-lo-lo-lo-lo-lo-lo-lo-lo-lo-lo-lo (18 samples, all 'lo');

Now let's add two separate 10mS noise spikes during the pulse. The result would be:

Sampler lo-lo-lo-lo-lo-lo-hi-lo-lo-lo-lo-lo-lo-hi-lo-lo-lo-lo-lo-lo (18 samples, 16 'lo' and 2 'hi')

Where the 'hi' samples are from added noise in the signal. The total number of low samples is 16 * 10mS each, which equals 160mS and is still decoded as '0' bit. The two noise spikes have not affected the decoding. On the other hand, pulseIn would have returned three separate pulses, each measuring 60mS (noise), with no valid bit result.

Step 3: Sampling with Interrupt.

Sampling is best accomplished by an interrupt. We set a counter on the microcontroller to interrupt the main program, check the radio signal, then return to the program. By using interrupts, we avoid blocking calls, allowing many samples each second without stopping display updates, touch sensing, or other activities.

In the STM32 environment, interrupts are handled by creating a pointer to a timer object, then controlling that timer through various function calls. Here is code to create the timer object:

```
TIM_TypeDef   *instance = TIM3;
HardwareTimer *timer = new HardwareTimer(instance);
```

These two lines establish an object for hardware timer #3. What does this timer do? It counts. We can set the rate that it counts, and we can change pin states or generate interrupts when the counter reaches a certain value. In our case, we will use an interrupt to run a special, short “interrupt handler” routine that samples the receiver output.

Here is a timer that generates an interrupt 100 times per second (10mS intervals):

```
void initTimer() {                                  // set up 100Hz interrupt
  timer->pause();                                   // pause the timer
  timer->setCount(0);                               // start count at 0
  timer->setOverflow(100,HERTZ_FORMAT);             // set counter for 100Hz overflow rate
  timer->attachInterrupt(timerHandler);             // overflow results in interrupt
  timer->resume();                                  // restart counting
}
```

The hardware timer is normally free-running, so we must stop it first and reset its value to zero. Then, change the threshold value to that its counter overflows 100 times per second. (The counter’s overflow value depends on how fast the microcontroller is running, but the details are handled internally by the timer object.) All we need to do is specify how often the counter should overflow. Next, specify the name of the interrupt handler, which will be called when the counter overflows. Finally, restart the timer with our new specifications.

To demonstrate the interrupt in action, we need a handler that counts to number of times it is called.

```
void timerHandler()
{
  sampleCounter--;
  if (!sampleCounter)                              // full second of sampling?
   {                                               // if so,

    sampleComplete = true;                          // flag it, and
    sampleCounter = 100;                            // reset sampleCounter
  }
}
```

This routine requires two variables: a countdown that indicates the number of intervals remaining in the sample, and a flag, “sampleComplete” which indicates that a full second (100 * 10mS per sample = 1 sec) of sampling has occurred. Each time the handler is called, the value of sampleCounter is decremented. When it gets to zero, the sampleCounter is reset back 100 and the flag is set.

Here is a complete sketch that pulls all of these interrupt elements together:

```
#include <TFT_eSPI.h>
#define TITLE "WWVB Test #3: INTERRUPTS"

TFT_eSPI tft = TFT_eSPI();
TIM_TypeDef   *instance = TIM3;
HardwareTimer *timer = new HardwareTimer(instance);

volatile byte sampleCounter = 100;                  // remaining samples in current second
volatile bool sampleComplete = false;               // 1 second sample flag
int displayCount = 0;                               // something to show on screen

void initTimer() {                                  // set up 100Hz interrupt
```

```
  timer->pause();                                    // pause the timer
  timer->setCount(0);                                // start count at 0
  timer->setOverflow(100,HERTZ_FORMAT);              // set counter for 100Hz overflow rate
  timer->attachInterrupt(timerHandler);              // overflow results in interrupt
  timer->resume();                                   // restart counting
}

void timerHandler()                                  // called every 10mS
{
  sampleCounter--;                                   // count down remaining samples
  if (!sampleCounter)                                // full second of sampling?
   {                                                 // if so,
    sampleComplete = true;                           // flag it, and
    sampleCounter = 100;                             // reset sampleCounter
  }
}

void setup() {
  tft.init();
  tft.setRotation(1);                                 // portrait screen orientation
  tft.fillScreen(TFT_BLACK);                          // start with blank screen
  tft.setTextColor(TFT_YELLOW);
  tft.drawString(TITLE,10,10,2);                      // show title at top of screen
  initTimer();                                        // start the counter
}

void loop() {
  if (sampleComplete) {                               // has a full second passed?
    displayCount++;                                   // yes, so increment counter
    tft.fillRect(50,50,140,20,TFT_BLACK);             // erase previous value
    tft.drawNumber(displayCount,50,50,4);             // show count on screen
    sampleComplete = false;                           // wait until next second is flagged
  }
}
```

The sketch displays a number, displayCounter, which increments once per second. The loop() function only does something useful if sampleComplete is true. But there is absolutely nothing in loop() that sets it true! How does it work? By now you realize that the interrupt handler is working in the background, and it sets the flag as soon as 100 samples have been obtained. One final detail: any time you have a variable that can be modified both inside and outside of an interrupt handler, you must declare it using the **volatile** qualifier. This is a special compiler directive that specifies where to place this variable. See the Arduino reference for more information.

Step 4: Interrupt-driven radio sampler.

In Step 2 we created a simple radio sampler based on the pulseIn function, and in Step 3 we created an interrupt handler for the purpose of sampling noisier signals. Let's put both ideas together so that we can decode incoming bits. The sketch is getting longer now, but we have seen all of its elements in the previous sketches:

```
#include <TFT_eSPI.h>
#define TITLE "WWVB Step #4: INTERRUPT-DRIVEN BIT DECODER"

#define RADIO_OUT PA12                               // microcontoller pin for radio output
#define ERRBIT 4                                     // value for an error bit
#define MARKER 3                                     // value for a marker bit
#define NOBIT  2                                     // value for no bit
#define HIBIT  1                                     // value for bit '1'
#define LOBIT  0                                     // value for bit '0'

TFT_eSPI tft = TFT_eSPI();                           // display object
TIM_TypeDef *instance = TIM3;
```

```
HardwareTimer *timer=new HardwareTimer(instance);  // timer object

volatile byte sampleCounter = 100;                 // remaining samples in current second
volatile byte newBit = NOBIT;                      // decoded bit
volatile byte pulseWidth = 0;                      // width (in 10mS units) of radio
pulse
int displayCount = 0;                              // something to show on screen

void initTimer() {                                 // set up 100Hz interrupt
  timer->pause();                                  // pause the timer
  timer->setCount(0);                              // start count at 0
  timer->setOverflow(100,HERTZ_FORMAT);            // set counter for 100Hz overflow rate
  timer->attachInterrupt(timerHandler);            // overflow results in interrupt
  timer->resume();                                 // restart counter with these
parameters
}

void timerHandler()                                // called every 10mS
{
  pulseWidth += !digitalRead(RADIO_OUT);           // if output line low, add to pulse
width
  sampleCounter--;                                 // count down remaining samples
  if (!sampleCounter)                              // full second of sampling?
   {                                               // if so,
    if ((pulseWidth>63)&&(pulseWidth<90))          // long pulses are markers
      newBit = MARKER;
    else if ((pulseWidth>33)&&(pulseWidth<60))     // intermediate pulses are bit=1
      newBit = HIBIT;
    else if ((pulseWidth>5)&&(pulseWidth<30))      // short pulses are bit=0
      newBit = LOBIT;
    else newBit = ERRBIT;
    sampleCounter = 100;                           // reset sampleCounter
    pulseWidth = 0;                                // start new pulse
  }
}

void setup() {
  tft.init();
  tft.setRotation(1);                               // portrait screen orientation
  tft.fillScreen(TFT_BLACK);                        // start with blank screen
  tft.setTextColor(TFT_YELLOW);
  tft.drawString(TITLE,10,10,2);                    // show title at top of screen
  initTimer();                                      // start the counter
}

void loop() {
  if (newBit!=NOBIT) {                              // bit received?
    int x=50, y=50;                                 // screen position
    tft.fillRect(x,y,200,20,TFT_BLACK);             // erase previous value
    displayCount++;
    x+=tft.drawNumber(displayCount,x,y,4);          // count the bits
    x+=tft.drawString(": ",x,y,4);
    switch (newBit) {                               // put decoded bit on screen
       case LOBIT: {tft.drawString("Bit 0",x,y,4);
          break; }
       case HIBIT: {tft.drawString("Bit 1",x,y,4);
          break; }
       case MARKER: {tft.drawString("Mark",x,y,4);
          break; }
       case ERRBIT: tft.drawString("Err",x,y,4);
    }
    newBit = NOBIT;                                 // wait for next bit
  }
}
```

The output will be a counter, updated every second (step #3), displaying a decoded bit (step #4). Notice that the interrupt handler now sets variable ‘newBit’ with the decoded bit instead of the simple Boolean flag. The bit is displayed in loop().

Step 5: Signal synchronization (start of the second).

Study the preceding code carefully and you will discover a small problem: how do we know that our sampling is beginning at the start of the second? If we happen to start our code in the middle of the second, we will sample the last half of one second and the first half of the next. We need a way to start our sampling such that the 1 second sampling window coincides with the start of the second from the incoming radio signal. As it turns out, pulseIn has a use here! We can measure any incoming pulse, then restart our timer when next second starts. For example, if we receive a marker pulse, we know that it is 800 mS wide and therefore the next second starts 200mS after the marker is finished.

The following code halts the timer (and your entire program) until a marker pulse is received:

```
void sync() {
  int pw;                                           // pulse width, in milliseconds
  timer->pause();                                   // stop the timer
  do                                                // listen to radio
    pw = pulseIn(RADIO_OUT,LOW,2000000)/1000;       // get width of low-going pulse in mS
  while ((pw<650)||(pw>900));                       // and wait for a Mark
  delay(200);                                       // mark is done; wait to start new bit
  sampleCounter = 100;                              // reset sampleCounter for 1 second
  timer->setCount(0);                               // when timer resumes, do full 10mS
  timer->resume();                                  // restart sampling every 10mS
}
```

Add this code to the end of the setup() routine, and loop() will not run until the timer is synchronized with the incoming radio signal. Remember that this is a blocking routine: at my location, I cannot synchronize until late in the evening.

Step 6: Display those bits!

It's time to spruce up the display a 'bit'. We will store and decode 60 bits each minute, so let's add a global array for storing them and an index to the current bit position:

```
byte frame[60];                                     // space to hold full minute of bits
int frameIndex = 0;                                 // current position in frame (0..59)
```

Next, let's create a loop that adds each bit into the frame array:

```
void checkRadioData() {
  if (newBit!=NOBIT){                               // yes! we got a bit
   if (frameIndex>59) startNewFrame();              // time to start new frame
   frame[frameIndex] = newBit;                      // save this bit!
   showBit(frameIndex,newBit);                      // yes, so show it
   frameIndex++;                                    // advance to next bit position
   newBit = NOBIT;                                  // time to get another bit
  }
}
```

Notice that this routine does not care about the display particulars. It calls showBit() to place the bit on the screen. The startNewFrame() will begin a new frame and clear the currently displayed bits.

With my first version of the clock, I displaying the bits in a circular pattern, attempting to emulate the beautiful DCF77 clock by Erik Deruiter. However, fitting 60 bits in a circle on a 320x240 pixel display results in teeny-tiny bits. The bits are larger and more readable in a rectangular pattern of 6 rows with 10 bits in a row. If each column is 20 pixels wide and each row is 20 pixels tall, the entire 60-bit display fits in 200 x 120 pixels – just right. The position of any bit is determined by its index. Each row is 20 pixels high, and the row is equal to index/10. For example, bit 52 is in row 52/10 = 5. So, the y coordinate is 20*(index/10) plus an offset. Similarly, the column position is the remainder of index/10 (In the same example, bit 52 is in the column 2, since the remainder of 52/10 is 2). Arduino has a built-in remainder function, called Modulo and written '%'. The y coordinate, therefore, is 20*(index % 10) plus an offset. Here is a routine to display the bit number and the bit itself. Note that the last line does all of the important work:

```
void showBit(int frameIndex, int bitType) {         // display a bit on screen
  const int x=100,y=50,w=15,h=15;                   // screen position & bit size
  int color;
  int xpos = 20;
  tft.setTextColor(TFT_YELLOW,TFT_BLACK);
  xpos += tft.drawString("Bit ",xpos,80,4);
  tft.drawNumber(frameIndex,xpos,80,4);             // show the bit number
  switch (bitType)                                  // color code the bit
  {
    case HIBIT:  color = TFT_CYAN; break;
    case LOBIT:  color = TFT_WHITE;  break;
    case MARKER: color = TFT_YELLOW; break;
    case ERRBIT: color = TFT_RED; break;
    default:     color = TFT_BLACK; break;
  }
  tft.fillRoundRect(x+20*(frameIndex%10),           // now draw the bit
      y+20*(frameIndex/10),w,h,4,color);
}
```

Run Step 6 to see a colorful representation of the received data. You can comment out the call to sync() if you don't want to wait for proper receiver synchronization.

Step 7: More synchronization (start of the minute).

In step 5 we synchronized our signal sampler to the beginning of each second. And in step 6 we put each of the sampled bits into a frame buffer. But bit 0 may not be at the beginning of the minute. It could be anywhere in the minute, depending on when we started the sketch and got signal synchronization. Looking back at the table on page 2, we need to know the position of each bit in the current minute. But how can we tell when the minute starts? There is no special 'start of minute' marker.

The WWVB timecode gives us a way. Notice that the minute starts and ends with marker bits. In other words, two marker bits in a row (bit 59 of the first minute and bit 0 of the next) will tell us that a new minute is started.

To look for two consecutive marker bits we will need another variable, 'oldBit' to keep track of the preceding bit. Then, whenever our newBit is a marker and the oldBit was a marker, we know that a new minute has started. The relevant changes to checkRadioData() are highlighted in yellow:

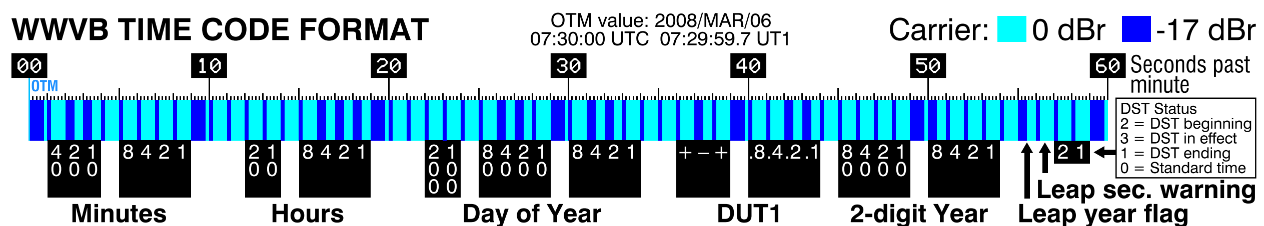
```
byte oldBit = NOBIT;                                  // previous bit


void checkRadioData() {
  if (newBit!=NOBIT){                                 // yes! we got a bit
   if (frameIndex>59) startNewFrame();                // time to start new frame
   if ((newBit==MARKER) && (oldBit==MARKER))          // is this the start of a new minute?
     startNewFrame();                                 // yes, start a new cycle
   frame[frameIndex] = newBit;                        // save this bit!
   showBit(frameIndex,newBit);                        // yes, so show it
   frameIndex++;                                      // advance to next bit position
   oldBit = newBit;                                   // remember current bit
   newBit = NOBIT;                                    // time to get another bit
  }
}
```

For this and subsequent sketches, wait until your Canaduino module has good reception. Once synchronized, you will notice that the first bit in the displayed frame is a marker bit, and that each row of the display ends in a marker bit.

Step 8: Decoding the frame

The preceding code didn't change much, and yet it was an incredibly important step. We now have a full, synchronized frame of WWVB data, with frame[0] containing the first bit of data and frame[59] containing the last. We have *everything* we need to decode the time.

The data is in binary-coded decimal format. Search the internet and you will find several good ways of converting BCD to binary. Feel free to try them out. In the end, I decided on a simple, brute-force approach that is sure to offend the programming crowd. It has all the elegance of a muddy boot, and yet I like how easy it is to understand and how it resembles to the WWVB time example:




```
void getRadioTime()                                   // decode time from current frame
{
   const byte daysInMonth[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
                             //JanFebMarAprMayJunJulAugSepOctNovDec
   const int century=2000;
   int yr,mo,dy,hr,mn,leap,dst;
   leap=dy=hr=dst=leap=mn=0;
   yr=century;

   if (frame[1]==HIBIT) mn+=40;                       // decode minutes
   if (frame[2]==HIBIT) mn+=20;
   if (frame[3]==HIBIT) mn+=10;
   if (frame[5]==HIBIT) mn+=8;
   if (frame[6]==HIBIT) mn+=4;
   if (frame[7]==HIBIT) mn+=2;
   if (frame[8]==HIBIT) mn+=1;
```

```
    if (frame[12]==HIBIT) hr+=20;                   // decode hours
    if (frame[13]==HIBIT) hr+=10;
    if (frame[15]==HIBIT) hr+=8;
    if (frame[16]==HIBIT) hr+=4;
    if (frame[17]==HIBIT) hr+=2;
    if (frame[18]==HIBIT) hr+=1;

    if (frame[22]==HIBIT) dy+=200;                  // decode days
    if (frame[23]==HIBIT) dy+=100;
    if (frame[25]==HIBIT) dy+=80;
    if (frame[26]==HIBIT) dy+=40;
    if (frame[27]==HIBIT) dy+=20;
    if (frame[28]==HIBIT) dy+=10;
    if (frame[30]==HIBIT) dy+=8;
    if (frame[31]==HIBIT) dy+=4;
    if (frame[32]==HIBIT) dy+=2;
    if (frame[33]==HIBIT) dy+=1;

    if (frame[45]==HIBIT) yr+=80;                   // decode years
    if (frame[46]==HIBIT) yr+=40;
    if (frame[47]==HIBIT) yr+=20;
    if (frame[48]==HIBIT) yr+=10;
    if (frame[50]==HIBIT) yr+=8;
    if (frame[51]==HIBIT) yr+=4;
    if (frame[52]==HIBIT) yr+=2;
    if (frame[53]==HIBIT) yr+=1;
    if (frame[55]==HIBIT) leap+=1;                  // get leapyear indicator
    if (frame[58]==HIBIT) dst+=1;                   // get DST indicator
          // more to follow…
}
```

At this point the minutes, hours, days, and year have been decoded, as well as flags for leap year and daylight savings. Notice there is no data for months. That is because the days' field represents the day of the year, not day of the month. For example, the day of the year for 9/11 is 255 in a regular year and 256 in a leap year. We need to convert DOY to month and day. Here is the code:

```
    const byte daysInMonth[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
                              //JanFebMarAprMayJunJulAugSepOctNovDec

    mo=1;                                           // convert day of year to month/day
    while (1) {                                     // for each mon, starting with Jan
       byte dim = daysInMonth[mo];                  // get # of days in this month
       if (mo == 2 && leap == 1) dim += 1;          // adjust for leap year, if necessary
       if (dy <= dim) break;                        // have we reached right month yet?
       dy -= dim;  mo += 1;                         // no, subtract all days in this month
    }
```

Step 8 provides fully-decoded time data from WWVB, which you can now use in your own projects. Continue reading to see how to incorporate this data into a working clock.

Step 9: The Time Library

The time library, written by Paul Stoffregen, gives us a handy way to deal with timekeeping in the Arduino environment. It allows us to set the time periodically via an external source, like our Canduino module, and keep that time updated in the background via the microcontroller's system clock. It is based on the standard Unit "time_t", which represents the number of seconds since 1/1/1970.

Using the Arduino Library Manager, install "Time by Michael Margolis". Then include it in your sketch:

```
#include <TimeLib.h>
```

To set the time, it is a simple one-line call added to the end of getRadioTime():

```
setTime(hr,mn,0,dy,mo,yr);                    // set the arduino time
```

This will set the time according to the decoded values. Try it and you will notice that this time lags that actual time by a minute. Why? The decoded values represent the minute that *just finished*. To correct for this, we could call another function, adjustTime(60), to add back the minute. Try that and you will be much closer. We are still lagging by a second, however. Why? Remember that we decode *after* the first second of the next minute. We need to add another second:

```
adjustTime(61);                               // adjust for 61 seconds of delay
```

Now is a good time to consider Time Zones. Most of the time zones on land are offset from Coordinated Universal Time (UTC) by a whole number of house. Here in the Eastern US, the time zone offset is -5, meaning that the display time is five hours earlier in the day than the zone at 0 degrees longitude. In the summer, the offset decreases to -4 on account of daylight saving time.

Let's add US time zone information in the form of several defines:

```
#define UTC  0                                // Coordinated Universal Time
#define EST -5                                // Eastern Standard Time
#define CST -6                                // Central Standard Time
#define MST -7                                // Mountain Standard Time
#define PST -8                                // Pacific Standard Time
#define LOCALTIMEZONE EST                     // Set to your own time zone!
```

WWVB broadcasts time as UTC. The local receiver must convert this time to local time, if desired, according to the time zone and, if needed, daylight savings time. This requires only two more lines of code. The adjustments are in seconds, so multiply the number of hours by 3600:

```
adjustTime(3600*LOCALTIMEZONE);
if (dst) adjustTime(3600);
```

Our time is now set according to local time Zone and DST. It is ready to be displayed. Let's show it every time a new frame is started:

```
void startNewFrame()
{
  frameIndex = 0;
  tft.fillRect(20,50,290,130,TFT_BLACK);         // clear bits on screen
  tft.fillRect(20,200,200,40,TFT_BLACK);         // clear previous time
  showTime();
  showDate();
}
```

Step 9 displays the time and date! See the source code for full details.

Test 10: Segments.

The previous sketch decodes and displays the time, which is exactly what we wanted. But watch it for a while and you will eventually notice an *occasional* time or date that is *incorrect*. This is not acceptable for a precision clock! We need to do perform error checking on our received bits. In other words, make sure the entire frame is good before using it to update the time. The microcontroller itself, via the time library, keeps reasonable time while waiting for good data.

There are no checksum or parity bits to confirm the frame is good. But, we do have marker bits at known locations in the frame. If all of the marker bits are received in their proper positions, we have some assurance of a good signal. So, let's start with marker verification.

The frame is punctuated with 6 marker bits at equal intervals, plus the start bit. Let's divide the frame into 6 corresponding segments, such that each segment contains 10 bits and ends with the marker bit, like this. In the illustration below, each row represents a segment.

M									M
									M
									M
									M
									M
									M

Not so coincidentally, this is the layout of our bit display. We create an byte array to hold the status of each of the segments, with seg[0] representing the top row and seg[5] representing the bottom. If the segment ends in a marker bit, the segment is considered valid. Using the remainder/modulo function, we can determine if we are at the end of a segment. For example, if we add the following lines to the getRadioData routine, by the time the frame is complete,

```
if ((frameIndex%10)==9)                              // are we at end of segment?
seg[frameIndex/10] = (newBit==MARKER);               // seg OK if ends in a Marker bit
```

the segment array will contain a 1=true for each segment ending in a marker, and 0=false otherwise. All we need to do is check the segments, and if any are bad then the frame is bad, too.

```
bool validFrame() {                                  // evaluate data in current frame
  for (int i=0; i<6; i++)                            // look at each segment
    if (seg[i]<1)) return false;                     // and return false if any are bad
  return true;
}
```

Finally, it would be helpful to display the segment status. Here are a pair of routines for displaying good/bad segment indicators.

```
void clearSegments() {
  tft.fillRect(160,200,120,20,TFT_BLACK);            // erase indicators
   for (int i=0; i<6; i++)                           // and wipe segment data
     seg[i] = -1;
}

void showSegments() {
  const int x=160,y=200,w=16,h=20,r=5;               // indicator position & size
  int color;
  for (int i=0; i<6; i++) {                          // for each segment:
    if (seg[i]<0) color = TFT_BLACK;                 // unevaluated segments are black
    else if (seg[i]>0) color = TFT_GREEN;            // good segments are green
    else color = TFT_RED;                            // and bad segments are red
    tft.fillRoundRect(x+i*20,y,w,h,r,color);         // display segment indicator
  }
}
```

Test 11: More Error Checking

The previous test significantly decreases the chance of displaying an incorrect time or date. If all of the markers are in their correct position, the signal is good and the data is likely correct.

But consider the frame on the right. The markers are in correct position, but there are two error (red) bits. We should obviously not use this frame of data, either. We need to modify our segment evaluation to look for error bits, too:

M	-	-	e	-	-	-	-	-	M
-	-	-	-	-	-	-	-	-	M
-	-	-	-	-	-	-	-	-	M
-	-	-	-	-	e	-	-	-	M
-	-	-	-	-	-	-	-	-	M
-	-	-	-	-	-	-	-	-	M

```
bool validSegment() {                          // true if current segment is good
  bool goodData = true;                        // assume everything is good :)
  byte start = frameIndex-9;                   // start at beginning of segment
  for (int i=start; i<start+8; i++)            // look at all bits in segment
    if (frame[i]==ERRBIT) goodData = false;    // if error found, segment is bad
  return goodData && (newBit==MARKER);         // seg must also end on marker
}
```

This function is inserted into the getRadioData routine so that the segment is evaluated after the 9th and final bit of each segment has been received:

```
if ((frameIndex%10)==9) {                      // are we at end of segment?
  seg[frameIndex/10] = validSegment();         // validate segment & save result
  showSegments();                              // show segment evaluation
}
```

Interestingly, it's getting more difficult to exercise the new code. You'll need marginal reception: good enough to produce good segments, but bad enough to introduce occasional errors. I get this type of reception in the morning and at dusk.

We aren't catching all of possible errors. For instance, a flipped bit (an erroneous '1' bit instead of a '0' bit, or vice versa) would never get flagged. But I am quite satisfied with the result.

The clock is almost finished. The last few steps are dedicated to improving the display.

Step 12: 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 in showTime(). Unfortunately, the small corner of the display we've used until now is not large enough to show these big glowy digits. Besides, Time belongs front and center, right where our bit display is. So, say goodbye to the bits for now. Bit display is turned off by creating and setting a Boolean flag, bool doingBits = false. Then, at the top of our showBits() routine, add

```
if (!doingBits) return;                      // if in 'bit' mode, continue
```

Our code runs as usual, evaluating segments and frames, but the bits aren't displayed. Now there is plenty of space to display the time instead.

While we are at it, add lines to specify the time and date colors. DEFINEs at the top of the sketch make it easy to change these colors later.

```
#define TIMECOLOR TFT_CYAN
#define DATECOLOR TFT_YELLOW
```

Run this sketch to see a nice VFD-like clock with time and date.

Step 13: Second time around

No precision clock is complete without displaying seconds. Let's add a routine to display them:

```
void showSeconds() {
  int x=162,y=65,f=7;                              // screen position & font
  int s=second();                                  // get current seconds
  tft.setTextColor(TIMECOLOR, TFT_BLACK);          // set time color
  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
}
```

At the moment, our showTime() routine is called whenever a new frame starts, which is only once a minute. But we need a way to update showSeconds each *second*. Worse yet, if signal is poor and we cannot synchronize, the time might not be updated for hours. A better place put calls to time display is in the sketch's main loop().

One possible method for updating each second uses the Arduino millis() counter, comparing it to a previously saved value. When the difference reaches 1000, you know a second has passed. A better way is to save the Arduino time variable, t, which increments every second, and wait until it differs from the saved value. I like this method better: the display is updated exactly when time has changed.

```
void updateTimeDisplay() {
  if (doingBits) return;                           // bit vs. time display
  time_t timeNow = now();                          // check the time now
  if (timeNow!=t) {                                // are we in a new second yet?
    if (minute() != minute(t)) {                   // are we in a new minute?
      if (hour() != hour(t))                       // are we in a new hour?
         showDate();                               // new hour, so update date
      showTime();                                  // new minute, so update time
    }
    showSeconds();                                 // new second, so show it
    t = timeNow;                                   // remember the displayed time
  }
}
```

The routine uses a global variable, t, to hold the most recently displayed time. The seconds are updated whenever the time changes. To minimize screen flicker, the time and date are only updated on the minute and hour, respectively. Finally, the main program loop is simplicity itself:

```
void loop() {
   updateTimeDisplay();                            // keep display current
   checkRadioData();                               // collect data & update time
```

```
}
```

Run this sketch and watch those seconds come to life.

Step 14: Clock Status

It took me 13 iterations to get a fully working clock, as documented above. I put it aside one morning, then came back later in the day to check it. It looked fine, but I wondered: "How current is the data? Has it been updating every minute, or is the last reception hours old?" A status indicator light would be a nice. First, create a global variable to store the time of last receiver decode, then update this variable when the receiver data is decoded:

```
time_t goodTime = 0;                              // time of last receiver decode

// in getRadioData insert the following:
goodTime = now();                                 // remember when time last decoded.
```

A color-coded status light will tell us how stale the radio data is. For example, green means a decode in the last hour; orange is a decode in the last 12 hours, and red is anything longer than that:

```
void showClockStatus () {
  int color,x=20,y=200,w=80,h=20,ft=2;            // screen position and size
  if (!goodTime) return;                          // haven't decoded time yet
  int minPassed = timeSinceDecode();              // how long ago was last decode?
  tft.setTextColor(TFT_BLACK);
  tft.fillRect(x,y,w,h,TFT_BLACK);                // erase previous status
  if (minPassed<60) color=TFT_GREEN;              // green is < 1 hr old
  else if (minPassed<720) color=TFT_ORANGE;       // orange is 1-12 hr old
  else color=TFT_RED;                             // red is >12 hr old
  tft.fillRoundRect(x,y,80,20,5,color);           // show status indicator
}
```

We determined the elapsed time, in minutes, since the last decode. We can also show this value inside the status indicator:

```
void showClockStatus () {
  int color,x=20,y=200,w=80,h=20,ft=2;            // screen position and size
  char st[20];                                    // string buffer
  if (!goodTime) return;                          // haven't decoded time yet
  int minPassed = timeSinceDecode();              // how long ago was last decode?
  itoa(minPassed,st,10);                          // convert number to a string
  strcat(st," min.");                             // like this: "10 min."
  tft.setTextColor(TFT_BLACK);
  tft.fillRect(x,y,w,h,TFT_BLACK);                // erase previous status
  if (minPassed<60) color=TFT_GREEN;              // green is < 1 hr old
  else if (minPassed<720) color=TFT_ORANGE;       // orange is 1-12 hr old
  else color=TFT_RED;                             // red is >12 hr old
  tft.fillRoundRect(x,y,80,20,5,color);           // show status indicator
  tft.drawString(st,x+10,y+2,ft);                 // and time since last good
}
```

Finally, it is time to "rethink the sync". If the clock loses its original time synchronization, decoding will fail, even for strong signals. It is possible, and maybe desirable, to keep track of how close we are tracking the incoming signal, and to adjust synchronization on the fly. But I settled on a simpler approach: if a long time has passed since we've successfully decoded a full frame of data, try a round of synchronization before continuing. Regardless of the method, it is important to remember that the incoming signal may severely degrade or entirely vanish, making synchronization impossible.

I didn't have a good idea how to do this at first, so I programmed top-down instead of bottom up:

```
void loop() {
   updateTimeDisplay();                              // keep display current
   if (needSync()) doSync();                         // sync if data is stale
   checkRadioData();                                 // collect data & update time
}
```

We should resync when significant time has passed since the last good data. If we want to resync once per hour, which seems a reasonable amount to time, the modulo function will work. `timeSinceDecode()%60` will return the same value once per hour:

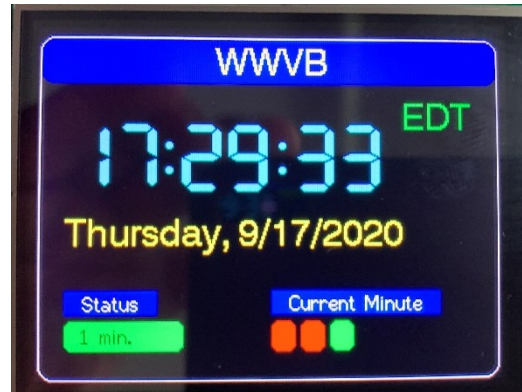
```
bool needSync() {
  int flag = timeSinceDecode()%60;                  // haven't sync'd for a while?
  return flag==30;                                  // do sync @ 30 min mark
}
```

I modified the sync code slightly to display its status and timeout after 5 minutes. See source code for details.

Finishing Touches.

Step 15 adds 12/24hr display, local/UTC display, and day of the week. The final source code adds touch control and bit/time display. Drop me a line if you build your own WWVB clock!

Local Time, 12hr format



Local Time, 24hr format

UTC time, 24hr format

Received Bits