Arduino-based GPSDO

Design based on the one posted on the time-nuts list by Lars Walenius

J. Harman 3/17//2016

Hardware notes

Overview: The system uses a 74HC390 chip to divide a 10 MHz oscillator to 5 MHz and then to 1 MHz. The 1 MHz and the 1 pps from the GPS drive a 74HC4046 phase comparator and integrator, which produces a pulse with a height proportional to the time difference between the two rising edges. An A/D converter reads the pulse height and the processor uses this data to derive the control voltage for the oscillator.

Detailed description: This iteration uses a 32u4-based Arduino Micro. Adafruit sells it under their part #1086. Beware of low-cost counterfeits. Note that this is different from the Sparkfun Pro Micro. The Micro has several advantages compared to the more common Arduino Uno. Because the 32u4 processor’s USB communication is built in to the processor and supports a virtual serial port, communication with the PC does not tie up the processor’s hardware serial port. The software uses the processor's serial port to communicate with the GPS using NMEA. This Serial1 port lets you send setup commands to the GPS, monitor GPS status, and include the actual time and date in the logging data.

Another advantage of the Micro is that unlike the traditional Arduino boards, its pins are on a uniform 0.1” grid, so you can mount it on a standard breadboard with that spacing.

I use the Adafruit GPS module with an external antenna. This is not a specialized timing receiver but its 1 pps output is specified at a stability of 100 nsec RMS. My performance is generally considerably better than this. Solder on the battery clip and install the button cell so that the GPS will remember its almanac data between power cycles. This makes startup much faster. If you use a different source for the 1 pps, make sure that the rising edge of the pps pulse marks the time. Often the width of the pps pulse is not controlled accurately.

The oscillator is a C-Mac STP-2322 10 MHz OCXO that I got on ebay. It is simplest to use an oscillator that runs off 5 V and produces HCMOS compatible output. It helps the stability if you choose an oscillator that provides a voltage reference output, because that provides a stable supply voltage for the VFC circuitry. Otherwise you will be depending on the stability of the Arduino’s 5 V regulator, or you will need to generate the reference voltage separately.

If your oscillator has an oven, chances are that during warm-up it will need more current than the USB or the on-board regulator can supply, so you will need an external power supply. Of course you will also need an external supply if you want to run the system stand-alone. Because OCXOs take days to stabilize, you will want to leave the system powered up as much as possible.

The power supply uses a 5V 2A wall-wart style supply. I found that the ground noise and thus the system noise is reduced if you use a low leakage “medical” rated supply. The 5V supply powers the oscillator and also feeds a DC-DC converter (5V to 9V) that drives the Vin pin to power the rest of the circuitry via the Micro’s 5V regulator. The converter I used is a muRata MEE1S0509SC (DigiKey 811-2731-ND) but there may be better choices. What I found with the muRata part is that its output voltage is unregulated and goes way above 9 V if it has much less than its rated load current of 110 ma. I added an 810 ohm 1/4W resistor across its output to correct for this. It also needs 10 uF tantalum capacitors with short leads at its input and output to avoid large voltage spikes.

The 74HC4046 phase comparator produces a pulse once per second with a width (0-1 usec) equal to the time difference between the PPS from the GPS and a 1 MHz pulse derived from the OCXO. The output at pin 15 goes High at the rising edge of the 1 MHz derived from the oscillator and is reset by the rising edge of the 1 pps pulse from the GPS. The diode/RC network at the phase comparator output converts this to a pulse with a height (0 – 2.56 V for the 32u4 processor and 0 – 1.1 V for the 328p-based Uno) proportional to the time difference. This pulse feeds ADC 0 of the processor. With the ADC’s 10 bit resolution, this provides a timing resolution of 1 nsec. The 1 Meg resistor discharges the capacitor between pulses, but slowly enough that the A/D reading is not affected. A real current source with steering diodes would certainly be more linear, but this circuit has proven to be good enough for this application.

In place of the pseudo-16bit PWM DAC used in Lars’ original design, I use an MCP4725 12-bit I2C DAC. This has the advantage that it can be powered by the oscillator’s reference output, eliminating any dependency on the stability of the main +5 supply. The resolution is marginal but it seems to be OK for this application. Maybe someday I will experiment with the original design or a true 16 bit DAC, but I have not found one in a DIP package or breakout board at a reasonable price.

A 5 MHz clock derived from the OCXO drives the processor’s Timer 1, which interrupts every 64K counts. The interrupt routine simply counts overflows. The timer value is captured by the processor hardware at the time of the PPS interrupt and along with the overflow count is used to produce a coarse timer with 200 nsec resolution.

An LM35 temperature detector feeds ADC port 1. This reads the ambient temperature with a resolution of 0.1 degrees C. This data is currently logged but not otherwise used.

Operation notes

The recommended startup sequence for loading software and communicating with a PC is as follows. This is for the Arduino Micro but should be similar for the Uno.

* Apply power to the system and plug in the USB cable.
* If you are using the Enhanced Serial Monitor (ESM – see below), start it up. Close the About box and click on Window/Monitor to open the Monitor window. Select the port for communication from the list. Leave the speed at 115200 but do not click on Open yet. Note that due to a bug in the ESM, when the PC software is first started the Plot-Sweep scale is not set correctly. Open the Plot-Sweep window and change the setting in the drop-down at the top right to 1000.
* In the Arduino IDE, make sure the correct board and port are displayed at the bottom right. If not, select them on the Tools menu.
* Go through the normal process of compiling and loading the software into the Arduino. When the loading is complete you will hear the sound of the port closing and then re-opening. You now have 10 seconds to open a serial monitor.
* If you are using the ESM, click Open in the Monitor window. Logging data should start spewing out.
* If you are not using the ESM, open the Arduino serial monitor.
* If you waited more than 10 seconds before opening a serial monitor, the system assumes it is running in stand-alone mode and data transmission is turned off, In this case you must press the reset button on the processor or send an X command to start it up.

Note that with the Uno, the processor will normally reset and the sketch will restart when you connect the PC. To avoid these resets, connect a 4.7 uF 10 V capacitor between the Reset pin and Ground. Make sure the + side of the cap goes to the Reset pin. You will need to remove this capacitor to reset the processor and download new software.

If you want to disconnect the computer, first send the X command to stop data transmission, then close the serial port and unplug the cable. Reverse these steps to reconnect the computer.

While the system is running it blinks the processor LED to show its status. A long blink indicates Run mode; any other mode produces a short blink.

Interpreting the log

The log contains detailed information about the past and current operation of the system. The data is sent one line per second on a 5 minute cycle, in blocks of 300 lines. Current data is at the start of each line. The last few columns of each line have either one of the 5 minute averages from the past 12 hours or 3 hour averages from the past 18 days, depending on which section of the block is being displayed. You can pause the log at any time to review the data.

The data layout is:

* Current local time and date. You can change time zone and DST settings by modifying constants in the software. With the Micro processor, this data comes from the GPS each second. With the Uno, the GPS data is read once at startup and incremented every second.
* Extended TIC value, including wrap-arounds if the error is more than 1 usec. Nominal is 500, resets if it exceeds +/- 20,000.
* Raw TIC value. Nominal is 500
* Current low pass filtered TIC error (average TIC value – 500)
* Current raw DAC value at output of PI filter
* Current DAC output, after aging compensation and dithering
* Current ambient temperature, in 1/10 degree C
* Current time constant
* Cumulative time offset in 200 nsec ticks. Should be stable at 4 or 5 if the loop is locked and there have been no wrap-arounds, resets if it exceeds +/- 100 (20 usec)
* Time offset in the past second in 200 nsec ticks. Nominal is 0.
* Number of missed PPS interrupts since startup.
* Current operating mode – Warmup, Settle, Run, or Hold
* Average number index – 0-143 are 5 minute averages and 1000-1143 are 3 hour averages. At the start of each section it shows which one it is currently collecting. Wraps after 143.
* Average values: TIC and DAC for 5 minute averages, TIC, DAC, and temperature for 3 hour averages.

Enhanced Serial Monitor

On the PC side I have incorporated an enhanced Arduino serial monitor that replaces the Arduino serial monitor and generates real-time plots on the PC so you can monitor the system’s operation. The monitor software is available on request from its author and is described in this thread in the Arduino forum: [forum.arduino.cc/index.php?topic=185740.0](http://forum.arduino.cc/index.php?topic=185740.0)

All the calls to the ESM are contained in one function so you don't have to use it if you don't care about the plots and other data.

The ESM data is displayed in several windows:

Monitor is like the standard Arduino serial monitor and displays all the logging data.

Sender is where you can enter commands as described below

Log displays one line of average DAC and temperature data every 3 hours. I copy this data to Excel to monitor oscillator drift

.

Alert shows system status and error messages

Histogram shows a histogram of 5-minute average TIC values

Plot-Sweep produces an oscilloscope-like trace with (normally) the past 24 hours of raw TIC (black), average TIC (red), DAC (blue), and temperature (green) data, You can set the sweep rate with the S command.

Plot-XY shows the past 15 days of 5-minute average TIC values with the days stacked vertically, to show day-day correlations

Sending Commands

To send commands using the ESM, use the Sender window. Make sure Loopback is un-checked, You can click on Less to make the window smaller. Enter your command then click Send.

To send commands with the standard Arduino monitor, make sure Newline is selected at the bottom right. Enter your command at the top and click Send.

Commands can be entered in either upper or lower case. Supported commands are:

* Hxxxx – enter Hold mode and force the DAC to xxxx decimal
* R – enter Run mode. The system goes to Settle mode first, then switches to Run after the loop has stabilized.
* Txxxx – set time constant in seconds (default is 1024)
* Sxxxx– set interval for the ESM Plot-Sweep graph in 1/10 sec. Default is 864 which produces one 1000-point sweep per 24 hours
* X – start/stop sending data to PC.
* jxxxx – used to inject a step in the TIC value, for loop tuning

Processing overview

At startup, the system performs some initialization then waits for the first PPS interrupt. After the interrupt it initializes the timers, gets an initial temperature reading, and retrieves the last stored DAC setting, which it uses as the initial DAC value. If the stored value is invalid as it most likely will be when starting for the very first time, it sets the DAC to half-scale. After these initializations it enters the main loop.

Operation starts in Warmup mode. In this mode the DAC remains at its initial setting. The system remains in Warmup mode until the oscillator frequency error is less than 2 parts in 10^8 (1,200 TIC counts in 60 sec). With my oscillator this takes a few minutes after a power down of up to a few hours. If you are using the ESM, the Plot-Sweep scale is set to 2 seconds per point. While the oscillator is warming up you should see a flattening sawtooth on the black trace.

When the oscillator frequency is close to nominal, the system switches to Settle mode. The time constant is set to 128 seconds and the control loop is closed. It remains in Settle mode until the system has stabilized further, then switches to Run mode.

In Run mode the time constant is set to its standard 1024 seconds and the ESM Plot-Sweep scale is set to 86.4 seconds per point, so it does a complete sweep in 24 hours. If the average TIC value drifts below 300 or above 700, the system switches back to Settle mode in an attempt to keep the loop locked.

Software notes

I am using Arduino IDE 1.6.8 but it should work with any version 1.0 or above that supports your board.

The code is conditionally compiled so it \*should\* also work with the Uno’s 328p processor, but I have not tested that recently. If you are using the 328p it uses the Software Serial library to get the time from the GPS at startup but it cannot listen to the NMEA while running.

You will find that the vast majority of the code deals with monitoring and logging the system's operation, which you may not need at all once you have the system running.

The main components of the code are in the following routines:

setup() – Initializes the hardware and software

loop() – The main software loop. This runs once per second, triggered by the PPS interrupt. Accumulates and decodes NMEA data from the GPS while it is waiting for an interrupt.

Timer 1 overflow interrupt handler – 16 bit Timer 1 is driven by a 5 MHz signal derived from the oscillator. This overflows and generates an interrupt approximately every 13.1 msec. The interrupt handler simply increments an overflow counter.

Timer 1 capture interrupt handler - triggered once per second by the PPS pulse from the GPS, at the same time as the peak of as the analog pulse generated by the phase comparator. The processor hardware grabs a snapshot of the Timer 1 value at the time of the interrupt request, minimizing any latency. In the interrupt routine it reads ADC0 to get the current phase offset (TIC\_Value). The interrupt routine also combines the captured Timer 1 value with the timer overflow count to make a 32-bit timer with a resolution of 200 nsec. This provides a measure of the oscillator frequency which is used for error detection and during warmup, to detect when the oscillator frequency has stabilized. Just before exiting, the interrupt routine sets PPS\_ReadFlag to signal that an interrupt has occurred.

calculation() – uses the TIC value to update the PI filter, and determines what value to send to the DAC that drives the oscillator’s VFC pin. Different algorithms are used depending on the operating mode: Warmup, Settle, Run, or Hold. This routine also updates the 5-minute and 3-hour averages of key variables that are displayed in the log.

PI\_Control(err) – this implements a Proportional-Integral (PI) controller with a low pass filter to remove short term variations in the TIC value. The filtered TIC value controls a P term which is proportional to the current phase error and an I term which integrates the phase error to zero out any long term error. You can adjust the filter characteristics by adjusting the time constant, which controls how fast the system responds to changes in the TIC value, and the Damping Factor, which controls the relative strength of the P and I terms to adjust the system's overshoot and stability. You can also adjust the gain, which determines the magnitude of the response to TIC value changes. You can change the time constant on the fly with the T command.

setDAC(rawDAC) – this applies corrections for aging and temperature and sends the desired value to the DAC. The temperature correction is currently disabled. It also dithers the DAC value, which increases the effective resolution. The dithering and aging compensation help to reduce errors at very long time constants.

printDataToSerial() – sends one line of logging data to the PC. The log includes current data, 5-minute averages for the past 12 hours, and 3-hour averages for the past 18 days. The averages repeat on a 5 minute cycle.

PrintDataToNewSerial() – sends additional logging data to the PC for plotting via the Enhanced Serial Monitor.

getCommand() – gets commands from the PC:

Construction notes

I initially hooked the whole system up on a solderless breadboard.  Anything with an SMD is on a breakout board that spreads its pins to 0.1" centers, so it all plugs right in. I migrated this to a solder-type breadboard that mimics the layout of the solderless board. The chips are in sockets and the processor board, GPS, and DAC are plugged into headers. Note that due to the fast rise times on the digital signals you will see more ringing and ground bounce on the solderless version. This noise may cause spurious PPS interrupts and noise or strange readings on the A/D converter. To avoid these problems, pay close attention to the lengths of the leads on your bypass caps and the total distance from the chips’ +5 pins through the bypass caps to their ground pins.

On the soldered version I suspended a ground wire about 0.5 in. below the bottom side of the board down the centerline of the chips and tied all the grounds to that. This helped minimize the noise and ground bounce.

If your oscillator does not have a separate ground pin for the oven, it is important to make sure that the VFC circuitry does not share a current path with the oscillator power, to avoid frequency shifts when the oven current changes.

It is important to shield the oscillator and GPS modules from air currents to avoid short-term timing variability. Enclosing the system in a small box reduces these variations.

Initial setup and checkout

Setting up and debugging the system is simplified if you have a storage scope with 10x probes, but with a little luck and patience you can get it working with just a DMM. The ESM software is also very helpful during setup, because of the real-time plots it produces.

To set the coarse oscillator frequency, it will be easiest to temporarily replace R1 and R2 with a 20 K 10 turn pot so that you can adjust the VFC voltage manually.

After wiring it all up and checking your connections, disconnect the DAC (you can simply unplug it) and power up the system. Disconnecting the DAC forces the control loop to be open, and it does not matter what code is running in the processor. The processor does need to be plugged in, however, because it generates the 5V for the GPS and the logic. Check that you have 5 V at the oscillator power pin, about 9 V at Vin, and 5 V at the processor’s +5V pin. The blue power light on the processor should be on and the oscillator should start to warm up.

The LED on the GPS will blink once per second while it is searching for satellites. When it has a fix, it will start to produce 1 pps pulses and the LED will flash every 15 seconds. Depending on your antenna location, it may take several minutes for the GPS to get a fix.

If you have a scope, make sure you are using 10x probes to avoid loading the signals you are trying to observe. Check that the noise on the power supplies is less than 100 mV p-p. If not, fix the bypassing. On the processor’s D4 pin you should see 1 pps, with minimal ringing. On D12 you should see 5 MHz, also with minimal ringing. On the D0 pin you should see the serial data from the GPS. If there is excessive overshoot or ringing on any of these pins, improve the grounding and/or shorten the wires that feed these pins. If there is still excessive ringing on the 1 pps, add a 100 ohm resistor in series with pin D4.

Continuing with a scope, pin 3 of the ‘HC4046 should have 1 MHz and pin 14 should have 1 pps. Pin 15 should have 1 pulse per second with a width of 0 – 1 usec. The width corresponds to the time difference between the 1 pps pulses from the GPS and the 1 MHz derived from your oscillator. Unless the oscillator frequency is exactly correct, the width of this pulse will be varying. It is easiest to observe this pulse if you can trigger the scope with the 1 pps from the GPS.

The processor’s A0 pin should have a 1 pps pulse that varies in height from about 0 to 2.5V (1.1V for the Uno processor), corresponding to the width of the pulse at pin 15 of the ‘HC4046. If this signal is noisy or voltage does not return to 0 between pulses, it is probably due to noise on the +5 from the processor or poor grounding between the processor and the rest of the circuitry. If the maximum voltage is too low or high, adjust C1 slightly, keeping in mind that the capacitance decreases and thus the amplitude will increase when you disconnect the scope.

Wait several minutes for the oscillator to warm up, then adjust the pot on the oscillator VFC until the pulse width is stable. Leave it running for several hours for the oscillator to stabilize, then adjust the pot again to stabilize the pulse width. You can leave the pot in place permanently, but the system may be more stable if you replace it with fixed resistors. The VFC voltage should be 2-3 V, but for an older surplus oscillator it is OK if it out of this range as long as you have some range left to compensate for aging.

If you don’t have a scope, adjust the pot for about 2 V on the VFC pin and we will use the software for the above adjustments.

Remove power, plug in the DAC, repower, and download the software as described above.

When the system starts up, it should start spewing out logging data. This procedure assumes that you have the ESM, but if not you can observe the same data in the text log.

Change the vertical interval in Plot-Sweep to 256 and the midpoint to 512 so that you can see the full TIC range of 0 – 1023. As the oscillator warms up you should see a sawtooth on the black TIC trace. The sawtooth should go from just above 0 to almost full scale. It is ok if there is a small amount of noise on the TIC data, but if it is more than 50 counts or so, you should look for and resolve power supply noise and/or ground bounce. If the sawtooth hangs at 1024 or resets before it gets to 900, adjust C1. More capacitance will reduce the amplitude.

If the oscillator is already close to the correct frequency, the sawtooth will flatten out as the oscillator warms up and the system will proceed to Settle mode. If necessary, give the oscillator several minutes to warm up and then adjust the pot until the sawtooth period is greater than 60 seconds, and the system should switch to Settle mode. If all is well, it will eventually switch to Run mode.

The next step is to adjust the gain of the control loop. Because this usually is a one-time setting you must change a constant in the code to do this. The idea is to set the gain to the number that represents a timing error of 1 TIC count per second.

* Give the oscillator several hours to warm up before adjusting this setting.
* Use the T100 command to set the time constant to 100 seconds and S100 to set the Plot-Sweep rate to 10 seconds per point.
* Note the current DAC value from the log (1000 say) and send H1000 or whatever to lock the DAC at that value
* Observe that the TIC is stable, not drifting up or down. If it is drifting, Change the Hold value until it stays stable for several minutes.
* Change the Hold value maybe 100 counts from the stable one and observe the slope of the TIC values.
* Find the number of DAC counts it takes to make the TIC move by 100 counts in 100 sec and set the gain constant in the code to this value.
* This should be close enough for good operation, but if you want to experiment further, you can use the j command while the system is running to inject offsets into the TIC and observe how the system responds as you vary the gain and damping factor.

Performance notes

With this design, short term TIC variations should be less than 20 counts from one second to the next. The short term variations are caused mainly by the uncertainty in the timing of the 1 PPS pulse from the GPS, which is synchronized to its internal clock. Using a timing grade GPS and modifying the code to use its sawtooth correction data would largely eliminate these short term variations. If you see larger short term variations than this, they are probably due to noise on the ground or +5 V lines. Also look for excessive ringing on the logic signals at the processor board, especially the 5 MHz line to the timer. A good oscilloscope is very helpful for tracking down these problems.

I have found that having a good antenna location makes a big difference in performance. With a good antenna location, the long term peak to peak variation in the TIC value is about 60 counts when the time constant is at its default of 1024 seconds. I do not have an independent reference, but most of this variation appears to be due to variations in the GPS signals as the satellites move overhead. As displayed on the ESM’s XY plot, the data is highly correlated with the value 24 hours ago. This is consistent with the 24-hour repetition of the GPS constellation.

With the time constant set to 10,000 sec, the TIC value wanders by about 175 counts when the heat comes on in my lab. Enabling and adjusting the temperature compensation should reduce this effect. Putting the whole system in a small picnic cooler with some water bottles helps to stabilize the temperature variations and give the loop time to respond.

.