Stratum Ten

An Arduino-based Stratum-1 NTP Server

–

aweatherguy

.

March, 2016

Chapter 1

Introduction

Figure 1.1: Assembled NTP Server

This document describes the construction of stratum-1 NTP servers with GPS-based time references and Arduino Megas. This project has been given the nickname *Stratum Ten* by the author. *Ten* refers to the 10MHz disciplined clock used in one of the variants, not the actual stratum designation of the server.

Two variants have been developed, each having excellent accuracy. Using the PPS output from a GPS receiver to track time, the two variants differ in the method used to measure time in between successive PPS pulses. One technique makes use of the Arduino Mega's crystal controlled oscillator; the other uses a precise 10MHz

clock output from the GPS receiver.

These servers support client/server and interleaved symmetric peer modes of operation within the NTP protocol. Interleaved symmetric peer mode can provide accuracies on Linux systems comparable to that of a local GPS reference clock. Both modes provide similar levels of performance on Windows 7.

An NTP server assembled in a plastic box with the cover removed can be seen in figure 1.1. On the right side are the BNC connector for PPS signal, and DB-9 RS232 connection for the time-of-day port on a HP55300A GPS receiver. The Ethernet Shield (version 2) can be seen on top of the Mega prototpye sheild (with red solder mask). On the back wall wires going to the five status LEDs which are visible. The LEDs are simply glued into five small holes drilled in the wall of the box.

An example of each variant has been built; one uses an HP55300A reference clock, while the other uses an inexpensive GPS module made by Trimble. Here is a list of the major parts required to build this project:

- (1) Arduino Mega1280 or Mega2560.
- (1) Ethernet shield (version 2).
- (1) GPS-based reference clock, one of the following:
  - HP55300A Primary Telecom Reference Source
  - Trimble ICM SMT 360 GPS receiver module
  - Other inexpensive GPS modules will also work but are not supported by the sketch supplied as part of this project.
- (1) Arduino Mega prototype shield or custom shield PCB.
- A few miscellaneous inexpensive parts

Appendix F contains a list of some of the acronyms used in this report. The reader can also try an Internet search of other acronyms which are unfamiliar.

1.1 What is ***Stratum-1***?

There is sometimes a misconception that in order for an NTP server to be classified as stratum-1, it must meet certain accuracy criteria. The stratum-1 moniker only means that the server obtains its time directly from a reference clock. There is no stated or implied accuracy criteria for either the reference clock or the NTP server itself.

This project qualifies as a stratum-1 NTP server because it obtains time directly from a GPS-based reference clock. It also provides excellent accuracy, but that fact is not connected with the stratum-1 classification.

1.2 Accuracy

With NTP's client/server mode of operation, the limiting factor in performance is variable network timing. Transmit timestamps must be loaded into UDP packets *before* they are transmitted. Packets are then entrusted to the NIC for transmission. There may be a delay before the NIC can start transmission however. If the amount of delay is constant, it can be incorporated into the transmit timesatmp. On a busy network this delay becomes unpredictable with a large range of variability. The result is that accuracy is determined largely by network delay variations.

NTP's interleaved symmetric peer mode solves this problem. Hardware timestamps captured *after* a packet is transmitted are sent in a subsequent packet. These timestamps are precise and not subject to network delay variations for devices sharing a local network hub. Performance on par with a local reference clock becomes possible when running a Linux variant which includes kernel discipline and nano-second resolution (the so-called nano-kernel).

Users of Windows 7 and earlier are limited to milli-second accuracy by the the operating system's clock management capabilities. Here, interleaved symmetric mode is of less value.

Figures 1.2 and 1.3 detail the estimated accuracy of the two variants in each NTP mode of operation. The estimate for client/server mode does not include variable network delays and as such is only realizable on a lightly-loaded network with predictable delays. Network delay variations can add errors of 50μs or more, totally dominating the overall accuracy. Individual contributors are combined in root-sum-square fashion, and totals are the approximate result.

	Basic Variant	Advanced Variant
PPS Accuracy	±50ns	±50ns
PPS Interpolation Error	±500ns	n/a
Rx timestamp	±250ns	±50ns
Tx timestamp uncertainty	±250ns	±400ns
Total accuracy (SWAG)	±700ns	±500ns

Figure 1.2: Estimated accuracy in client/server mode

This does not represent a meticulous accuracy analysis, but the author feels comfortable in claiming the bottom line SWAGs[1] are not too far from the truth.

[1]SWAG: a Somewhat Wild-Ass Guess

	Basic Variant	Advanced Variant
PPS Accuracy	±50ns	±50ns
PPS Interpolation Error	±500ns	n/a
Rx timestamps	±250ns	±50ns
Tx timestamps	±250ns	±50ns
Total accuracy (SWAG)	±700ns	±100ns

Figure 1.3: Estimated accuracy in interleaved symmetric mode

Read the Fine Print

These accuracy estimates are based on the assumption that hardware interrupt signals from the NIC on the Etherenet Shield are true trailing timestamps. Inquiries with the manufacturer for verification on this point were not answered. To the extent this assumption is incorrect, the above accuracy estimates will be incorrect.

1.3 Limitations

The sketch provided with this project only supports only a limited subset of the NTP version 4 protocol. It only listens for UDP packets on port 123; packets sent to any other port (e.g. ping requests) are ignored.

Incoming NTP packets contain a 3-byte mode field in the first byte of the NTP packet. The sketch only responds to two settings of the mode field. Packets containing any other mode value are ignored.

Mode 1: Symmetric active mode is supported; a symmetric active packet is returned.

Mode 3: Client mode is supported; a server mode packet is returned.

Modes 0,2,4,5,6,7: Incoming packets with these modes are ignored.

This means that broadcast modes, symmetric passive modes and control messages are not supported.

Packet extension fields and message authentication are likewise not supported. Although the sketch has not been tested under these conditions, it is designed to ignore anything after the first 48 bytes and should return a normal response without extension fields or authentication hash.

1.4 Project Variants

The two variants are referred to as *basic* and *advanced*. The advanced variant has the potential to be several times as accurate as the basic. Both variants should be more accurate than what can be achieved with a PC running `ntpd` on Linux with a local reference clock.

Each variant requires at least one minor hardware modification to Arduino boards. The advanced variant also requires a new bootloader in the Arduino Mega board. Figure 1.4 more detail on hardware and modifications required to build the two variants.

	Variant	
	Basic	Advanced
Mega processor required	1280 or 2560	2560
Arduino Mega Modified	No	Yes
Mega System Clock	No ceramic resonators	GPS disciplined 10MHz
Arduino bootloader	Standard	Custom
Arduino IDE	Standard	Modified `boards.txt` file
Ethernet Shield Modified	Yes	Yes
GPS PPS output required	Yes	Yes
GPS 10MHz output required	No	Yes

Figure 1.4: Hardware and modification requirements for the two variants

The basic variant requires an Arduino Mega board with crystal-controlled processor clock. Many Arduino boards are supplied with ceramic resonators nowadays, and this won't work for the basic version. The ceramic resonator must replaced with a crystal, or the advanced variant can be built instead.

1.5 Origins

Many years ago the author acquired an HP55300A network time standard. The idea was to use it on a Windows PC to create a local NTP reference clock. This turns out to be extremely difficult however. It can be accomplished on older PC hardware if an RS232 port serviced by the standard Windows driver is available, but finding

such hardware on a PC these days is near impossible. PCIe cards are available with RS232 ports but the supplied kernel drivers do not provide interrupt-driven callbacks necessary for an NTP reference clock driver. The project stalled, and the HP55300A just sat around collecting dust for years.

A couple of years later, the project came to life again with the discovery of a web page describing the construction of a Stratum-1 NTP server using the Arduino platform. This information appears to have been presented at a conference on the Perl language[2], of all places. This discovery changed everything. After years of stalled progress, that started the ball rolling and led to the result described herein.

[2]http://cleverdomain.org/yapc2012

Chapter 2

Reference Clocks

The project examples that were built use two different GPS-based reference clocks. The choice of variant is independent from the choice of reference clock. The project firmware (sketch) supports both reference clocks. Combined with the two NTP server variants, there are actually four different configurations. Other GPS-based clocks can also work in this project if the sketch is modified to provide proper communications with the clock.

2.1 Communications and Control

This project assumes the reference clock provides a serial port of some kind for the purposes of obtaining time of day and controlling any options that must be set for proper operation.

Time of day outputs allow the firmware to figure out which exact second is associated with each PPS pulse. This is only necessary at start-up and infrequently thereafter, as a check to ensure that the firmware is still keeping time correctly.

Some reference clocks will require that one or more options be configured at start-up. For example, the Trimble ICM SMT 360 receiver defaults to broadcasting several pieces of data every second. This is incompatible with the firmware design and must be turned off.

The hardware interface to reference clock serial ports varies considerably. Details of the two reference clocks used for this project are included below. For those using a different clock, be sure to look over the serial port specifications carefully before building the hardware interface.

Figure 2.1: Hewlett-Packard GPS Reference Clock

2.2 HP55300A

This unit (pictured in figure 2.1) is a primary reference clock designed explicitly for the purpose of keeping time in telecommunications applications. It is similar to the HP Z3801A/58503A reference clock. At the time of this writing, there are a couple of these units available on e-Bay for around $450.

The reference clock contains a GPS receiver and oven-controlled crystal oscillator. Once warmed up and stabilized, accurate time is provided for as long as 24 hours or more if the satellite signals are lost. More information on this unit is available through internet searches.

This project makes two or three connections to the HP55300A, the PPS (pulse-per-second) signal and the RS232 Time of Day port. The rising edge of each PPS pulse is aligned precisely (±50ns or better) with UTC time on the second, and the Time of Day port tells exactly which second is coming up next. The 10MHz disciplined output is also used for the advanced variant.

2.2.1 Serial Port Specifics

The HP55300A has a true RS232 serial port which requires a logic inversion between the port and the Mega2560 processor's serial port pins. As luck would have it (or perhaps by design), this serial port works fine with a 0/5V signaling levels for input and the interface does not require any negative voltage supplies.

HP55300A Serial Port	
Signaling levels	±12V, Idle = -12V
Start bits	1
Stop bits	1
Parity	none
Initial baud rate	9600

2.3 GPS Receiver Modules

There are many GPS modules on the market which provide PPS outputs and time-of-day serial ports. Some have built-in antennas while others offer an external antenna connection. Many of these are quite inexpensive and can be substituted for the HP55300A reference clock. The Arduino sketch can be modified to work with these modules instead of the HP reference clock.

Given that the HP55300A is obsolete and fairly expensive even if you can find a used one for sale, what is the downside to using a commercial GPS module instead? The answer is holdover. If the HP55300A loses the satellite signal it has an extremely stable temperature-stabilized reference oscillator which can provide time accurate to 10μs or so for 24 hours – this is called holdover.

With an inexpensive GPS module, if the satellite signal is lost, the PPS output may just quit and the NTP server is left with no reference clock. In this situation, the Arduino sketch is able to provide some limited holdover – but just for a few minutes instead of one or more days. How well holdover works with Arduino depends entirely on how stable its temperature is. In a temperature stabilized environment it may be good for an hour or more but in most real-life situations time errors will build at the rate of one micro-second per minute or more. See the section on holdover for more information.

2.3.1 Trimble ICM SMT 360

The sketch supports the Trimble ICM SMT 360 GPS receiver module. This module is designed specifically for timing applications as opposed to navigation. It provides a 10MHz disciplined clock output which is required by the advanced variant of this project. It also provides automatic holdover for short periods of time if the satellite signal is lost.

This module has the added benefit that the 10MHz signal is always present and can be used for the Mega's system clock.

2.3.2 Trimble Serial Port Specifics

Compared with a standard RS232 interface, the ICM SMT 360 serial port is easier to interface connect to the Mega2560 processor. The only difference compared to the processor's serial port is due to 3.3 versus 5-volt power supply levels.

- Logic inversion is not required.
- Data line from GPS can be connected directly to Mega2560.
- Data line from Mega2560 to GPS needs only a simple resistive divider.

Trimble ICM SMT 360 Serial Port	
Signaling levels	0, 3.3V, Idle = 3.3V
Start bits	1
Stop bits	1
Parity	odd
Initial baud rate	115,200

2.4 Rhubidium Standards

For the more adventurous, rhubidium atomic frequency/time standards scavenged from cellular telephone equipment are available on eBay for one or two hundred dollars these days. An example from an eBay auction is shown on the right. Many of them will accept a PPS input for disciplining the oscillator and would provide even more accurate holdover than the HP55300A. Be aware that the rhubidium lamp used in these modules does have a limited lifetime and may need to be replaced. This option is beyond the scope of this report and is an exercise left for the reader.

Chapter 3

Ethernet Shield Modifications

This project uses a version 2 Ethernet shield based on the WizNet W5500 NIC chip. The shields pictured in figures 1.1 and 6.1 were purchased from SparkFun but other shields based on the W5500 should also work. The newer version 2 shield is recommended as it contains automatic reset circuitry not present on earlier shields and is easier to use.

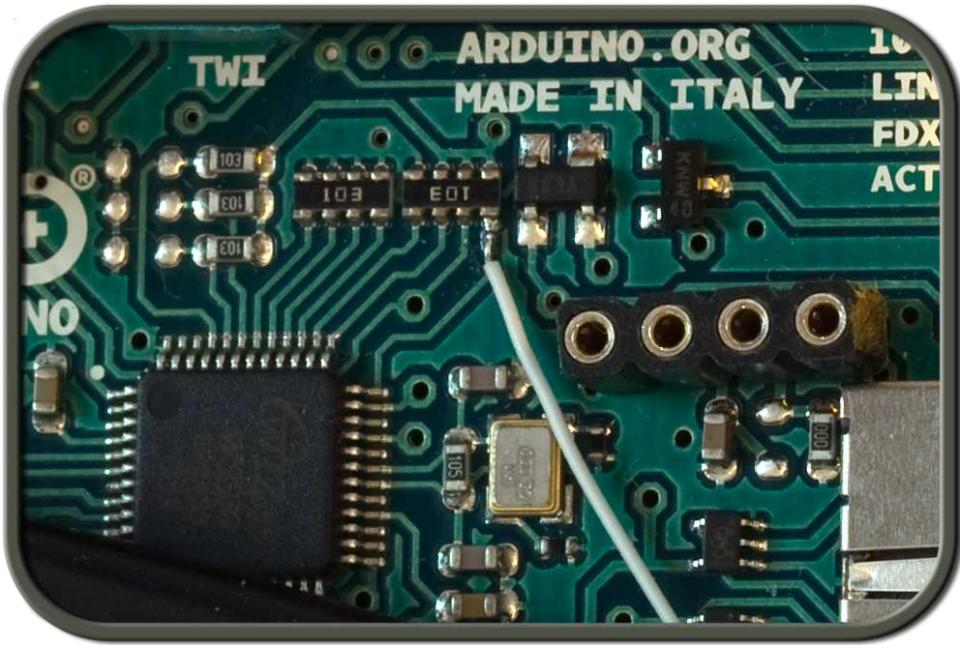


Figure 3.1: Connection to NIC interrupt pin

The following modifications are required:

1. Connect a wire from the W5500's interrupt pin (pin 36) to digital pin 48 on the Mega prototype shield. Instead of trying to solder directly to the interrupt

pin, connection to an associated pull-up resistor is easier. This can be seen in figure 3.1.

2. Older Mega boards may not have all of the socket pins used by the Ethernet shield. The Ethernet shield pins which do not align with any socket pin on an older Mega board are not used and can be cut off, or bent out of the way.

It is also possible that older Ethernet version 1 shields with the W5100 could be made to work if they provide receive interrupts for UDP packets. This is an exercise left for the reader.

Chapter 4

Shield Circuitry

This chapter describes the circuitry required on a hand-built shield. Most of the circuitry is providing an interface between the Mega's serial port and the GPS receiver. It therefore depends on which receiver is being used. The only variant-specific circuitry is a small circuit to condition the 10MHz GPS clock; this is omitted for the basic variant. Both variants of the Mega shield include the following:

- GPS receiver's PPS signal connects to digital pin 49; this is the input capture signal for timer 4 in the processor.
- The interrupt line from the W5500 NIC on the Ethernet shield connects to digital 48 – the input capture signal for timer 5.
- Digital pins 18 and 19 are also the I/O pins for the Mega's USART-1 and are connected to the reference clock's serial port through some interfacing circuitry.
- Digital pins 24-28 are optionally used to drive status LEDs. These LED drive pins can be connected to just about any unused digital I/O pins on the Mega, or not used at all.

The advanced variant's shield also includes circuitry to condition the reference clock's 10MHz output for use as the Mega's system clock.

Resistor values and transistors shown on the schematic are what was used on this project. Many other values and parts may be used successfully. Proper choice of values and parts is beyond the scope of this article and not discussed further.

4.1 Shield Mechanicals

Physically, the prototype shield can be a full-sized Mega prototype shield, or one that has been cut down. One of each has been built for this project.

4.1.1 Full-sized Shield

There is an incompatibility between the standard Mega prototype shield, and the Ethernet shield. The cause of this is shown in figure 4.1. There is a socket along one edge of the Ethernet shield which must plug into a matching header on the Mega. The socket (right) and header (left) are highlighted in the photograph. The Ethernet shield cannot be installed on top of a standard prototype shield because the proto shield does not provide a connection to the center header of the Mega board.

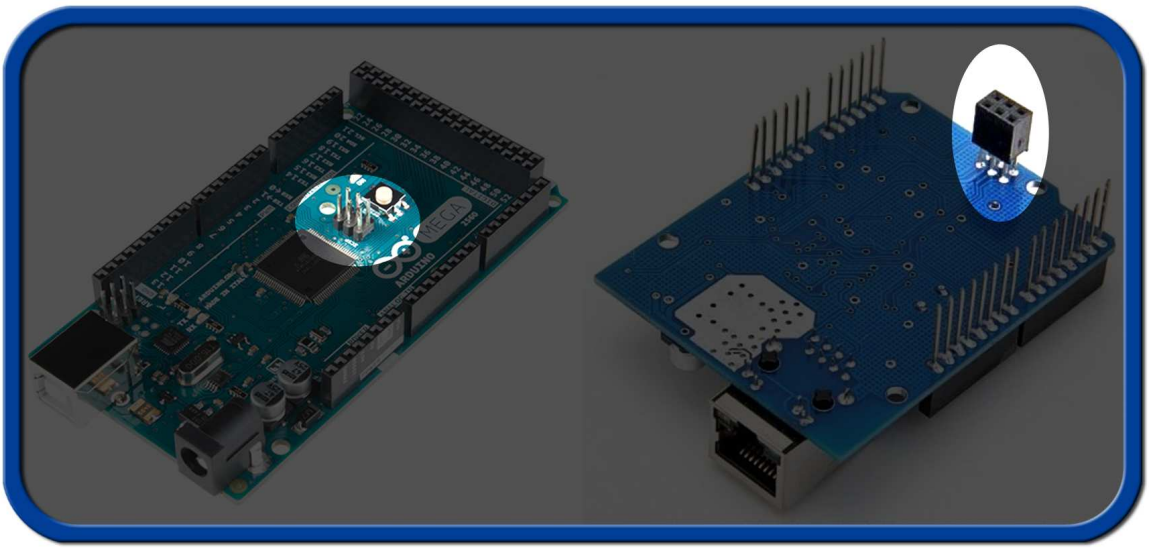


Figure 4.1: Special connector on Etherenet shield

To make this work, a fairly large square hole must be cut in the Mega shield to permit the socket to mate with the matching header on the Mega.

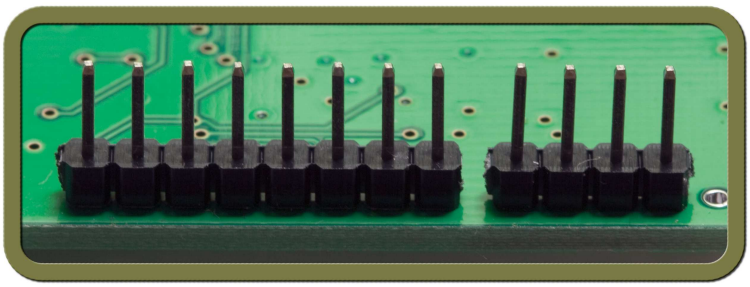


Figure 4.2: Header pins seen from bottom side of shield.

Install pin headers in the lower portions of the prototype shield. A sample of the

headers used is shown in figure 4.2. The black insulator is about 0.10 inch thick and the pins extend about 0.25 inch past the insulator. These should only be installed in analog 9-15, communication 14-21 and the bottom two digital rows. Do not install header pins in any location used by the Ethernet shield.

With the prototype shield installed on the Mega board, the Ethernet shield pins may now pass through empty holes in the prototype shield and plug into the Mega board. The example shown in figure 4.3 shows a full-sized shield sandwiched in between the Mega and Ethernet shield.

Figure 4.3 is a high resolution image of a basic variant shield built for interfacing with the HP55300A, and a lot of detail may be seen by zooming in at high magnification on the photo.

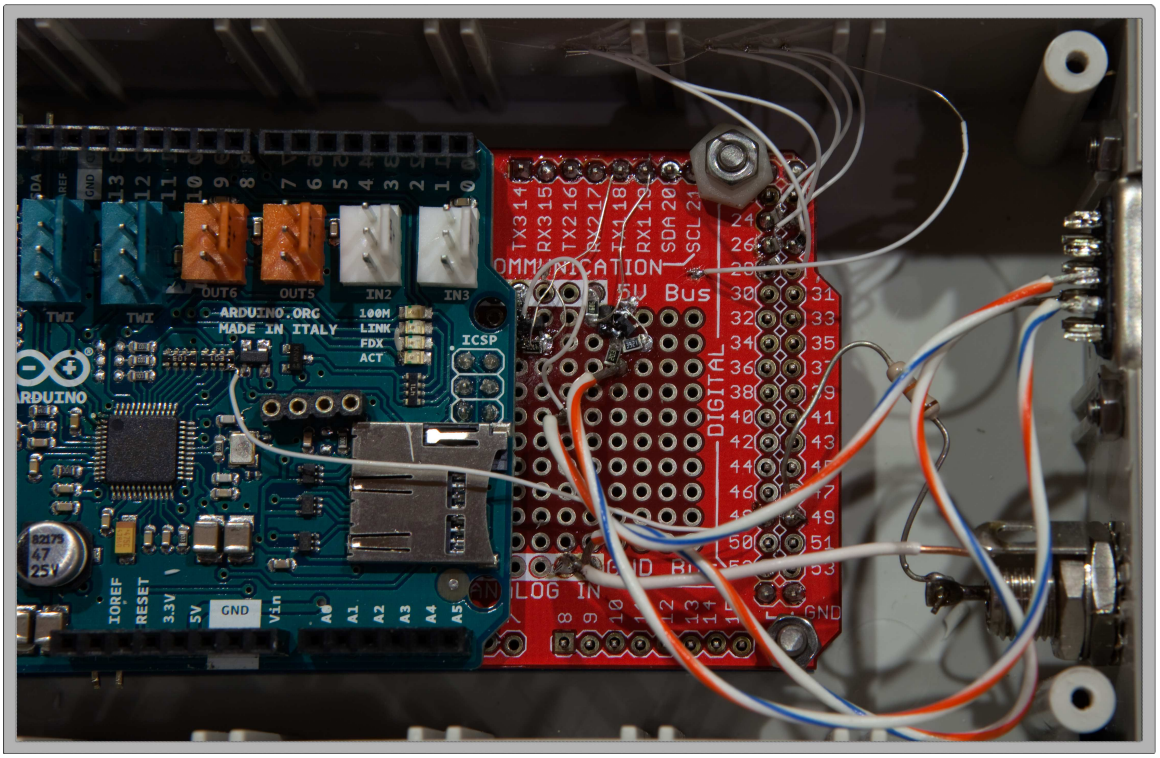
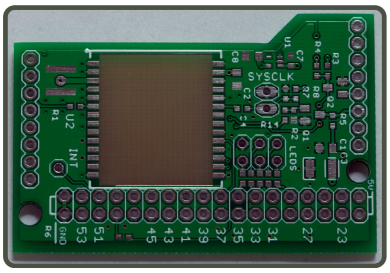


Figure 4.3: Prototype Shield Close-up

The shield in the photo uses SMT components, mostly in SOT-23 and 0805 packages. The current limiting resistors for the LEDs were just soldered directly to the digital pins, tombstone-style.

4.1.2 Cut-down Shield



As an alternative to cutting a large square hole in the prototype shield, the shield can be cut down so that it only overlaps Mega pins not used by the Ethernet shield. A photo of a custom PC board designed with this in mind is shown in the photo to the right. It gives an idea of how a Mega shield would be cut down to match. This custom shield board only plugs into the pins at the lower end of the Mega board (as listed above) and does not interfere with any pins on the Ethernet shield or the Ethernet shield itself. This same board can be also seen in figure 6.1, plugged into an Arduino Mega along with an Ethernet shield. Although this shield was designed for the advanced variant it demonstrates the concept of a cut-down shield which does not overlap the Ethernet shield.

4.2 HP55300A Interface

Figure 4.4 is a schematic of the shield circuitry required to interface with the HP55300A reference clock. The USART pins on the processor have logic levels inverted from what is required on the HP55300A's RS232 interface. This requires some interface circuitry to invert the signals before connecting to the HP55300A.

Additionally, the HP55300A has a standard RS232 interface with ±12V signal levels, but it seems to function properly with the 0-5V output signal produced by this design. Your mileage may vary.

Zener diode `D1` protects the Mega from excessive voltage if a 12-volt signal from an incorrectly wired RS232 cable were to find its way onto pin 3 of the RS232 connector.

The `R2/R3` voltage divider is designed to divide a 15-volt RS232 signal down to roughly 3 volts. For different reference clocks with lower output levels (e.g. 5 volts or less) a different divider ratio would be appropriate.

Many different N-channel enhancement mode MOSFETs could be substituted for the transistors shown in the schematic. That is an exercise left for the reader.

4.3 Trimble ICM SMT 360 Interface

A schematic for interfacing to a Trimble GPS module is shown in figure 4.5. With this GPS module, serial signal levels are the proper polarity and do not need to be inverted; only a simple resistor divider is required on the transmit line from Arduino to GPS module to reduce the 5-volt signal to a save level for the 3.3-volt GPS receiver.

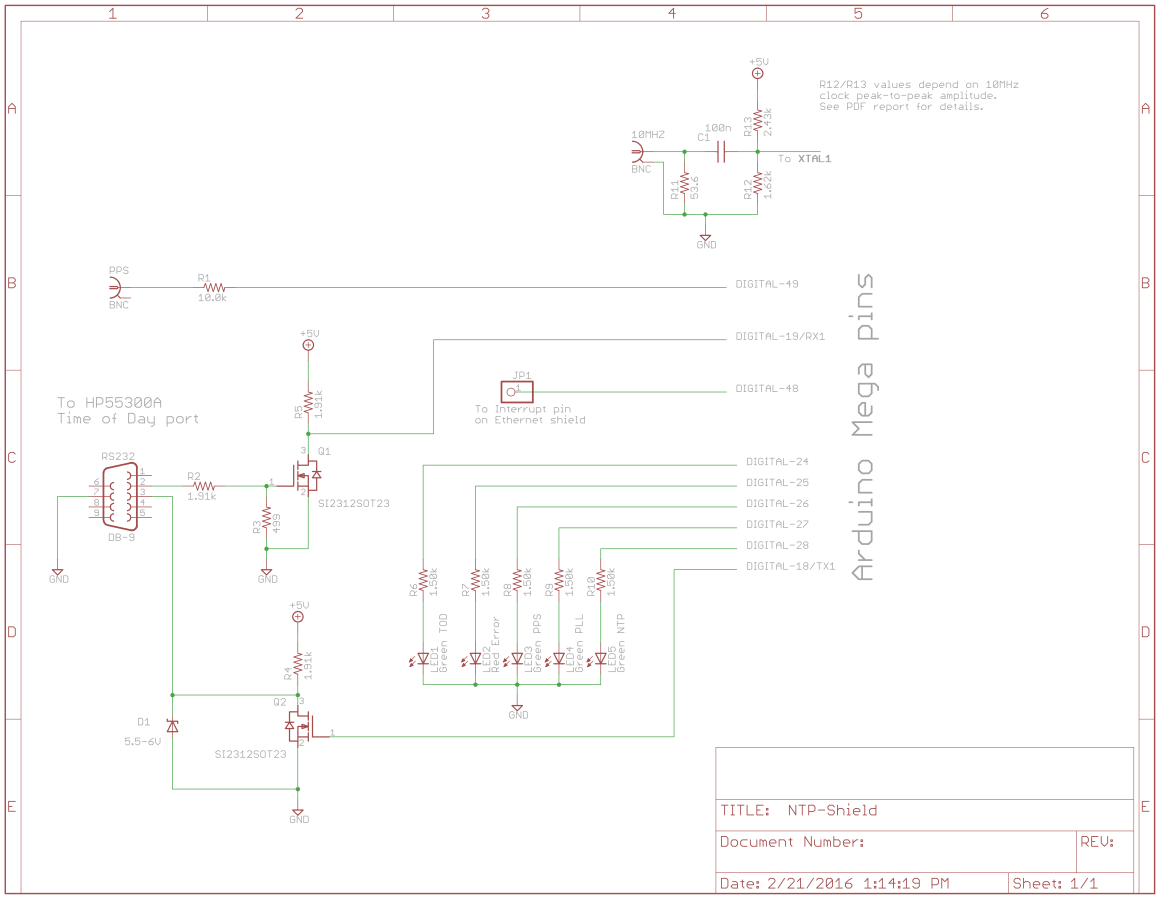


Figure 4.4: Arduino Mega Shield Schematic for HP55300A reference clock

The 3.3-volt serial data output from the receiver is sufficient to drive the Mega's USART input directly with no additional circuitry.

4.3.1 10MHz System Clock

Both shield schematics show the interface between a 10MHz GPS output and the `XTAL1` pin on the ATMega processor. This circuit is only required for the advanced variant, and even then some GPS receivers will not require it. Component values in this circuit depend on the characteristics of the GPS output signal.

Terminating Resistor R11

In most cases this may be omitted. It would be important if the cable connecting 10MHz from the GPS were longer than about ten feet. If the 10MHz source has

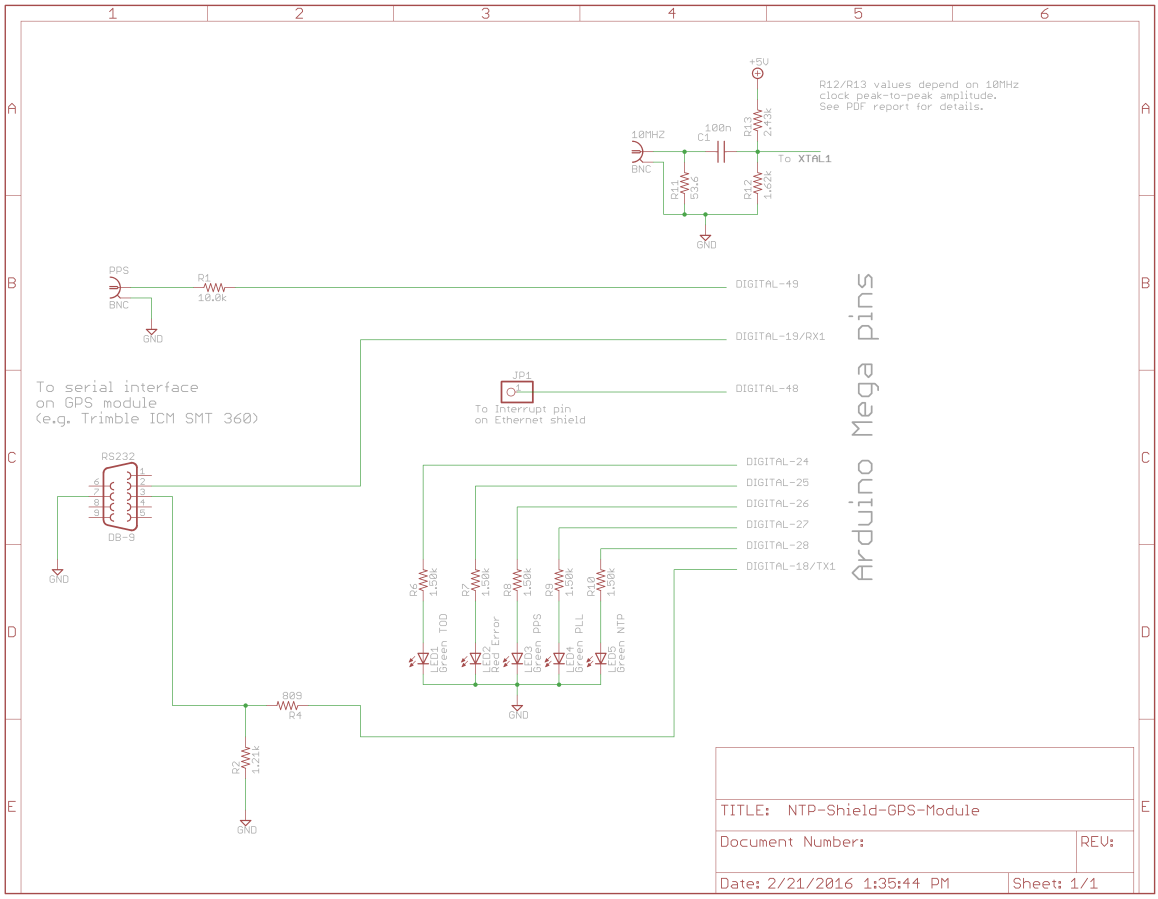


Figure 4.5: Arduino Mega Shield Schematic for Trimble ICM SMT 360 reference clock

a 50Ω source impedance, this resistor will reduce the signal amplitude by 50%. Whatever the case, signal amplitude going out of this circuit must be less than 5 volts peak-to-peak. For both of the clock sources described in this project, R11 may be safely omitted.

Blocking Capacitor C1

In some cases the average DC level of the 10MHz signal may need to be shifted. This is true for the HP55300A but not for the Trimble GPS module. C1 should be replaced by a zero-ohm jumper (or just omitted) if level shifting is not necessary.

Resistive Divider R12/R13

These work in conjunction with C1 to shift the average DC level of the 10MHz signal. For the HP55300A they place the DC level at about 2.4 volts. The resulting shifted signal's excursions should not go below zero volts or above 5 volts. The Trimble module's 10MHz signal does not require level shifting and R12/R13 should be omitted. In fact, for the Trimble module the entire level shift circuit is unnecessary and the 10MHz clock may be connected directly to the Mega processor's `XTAL1` pin.

Connecting the Clock

Remove the crystal or ceramic resonator and any associated capacitors and resistors. Both `XTAL1` and `XTAL2` pins should be floating when this is complete.

The interface circuit is located on the NTP server shield, and it is only necessary to run a short twisted pair of 30-gauge insulated wires from the shield to the `XTAL1` pin and ground on the Mega2560 board.

There is the potential to create radiated electro-magnetic interference in running the 10MHz wires from from the connector or GPS module over to the Mega board. Keep the wires as short as possible and keep them as close to the surface of the Mega PC board as possible. In some cases it might help to use some small diameter coax instead of a twisted pair.

Chapter 5

The Basic Variant

5.1 Hardware

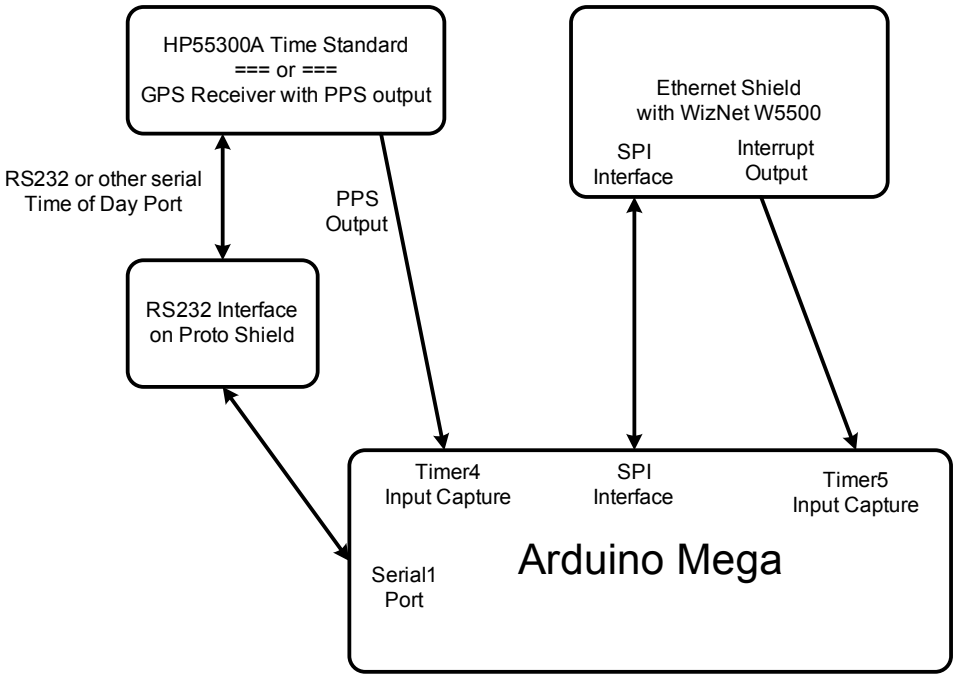


Figure 5.1: Hardware Block Diagram

Figure 5.1 shows the hardware and connections. The design is based on the Arduino Mega platform which uses an Atmel ATMEGA1280 or ATMEGA2560 processor. The processor's hardware includes several flexible timers which are key to

the operation of the NTP server. This article assumes the reader is familiar with these timers and their capabilities.

- Timers 4 and 5 operate with 2MHz clock giving 500ns resolution. They are both operated in compare-match mode and carefully synchronized so their values are always identical.
- Time of PPS is recorded by timer 4's input capture hardware.
- Time of UDP packet receipt and transmission (signalled by W5500 interrupt pin) recorded by timer 5's input capture hardware.
- Arduino's SPI interface controls the W5500 Ethernet NIC.
- Arduino's Serial1 port is used to obtain time of day from the GPS receiver.

5.1.1 Arduino Mega

The orignal project used an old Mega-1280 board although it should work with the new Mega-2560 boards as long as the system clock uses a crystal. Boards using a ceramic resonator will *not* work for this purpose.

The some Mega2560 boards appear to still have mounting pads for a 16MHz through-hole crystal in an HC49/U package so it should be possible to remove the resonator and replace with the crystal. The current version (r3) does NOT have space for a crystal or the parallel capacitors that would be required with it. Changing to a crystal might require re-programming fuses in the ATMega processor, or it might not. If you're going to go this route consider purchasing the most accurate crystal as it will make the hold-over feature last longer. For boards w/o the crystal mounting option, the project version using a 10MHz system clock is recommended.

One example of a replacement crystal is part number 9B-16.000MEEJ-B from TXC Corporation. This is available as of November 2015 from DigiKey (stock number 887-1244-ND) for less than a dollar. It has a specified frequency tolerance of ±10ppm and the same tolerance over temperature.

5.2 The Basic Variant's Weakness

The PPS signal input gives a very precise time reference once every second. However, important events occur (such as receipt of an NTP request packet) which are not aligned with the PPS signal. Some way is needed to measure time accurately in between PPS signals. The obvious tool at hand is one of the Atmel timer/counter modules driven by the processor's clock (a crystal-controlled oscillator). With a 2MHz clock on the timer it would be theoretically possible to measure the time of

an unsynchronized event (e.g. UDP packet arrival) relative to the PPS signal to within $\frac{1}{2}$clock cycle ($\pm$250ns).

This sounds good at first blush, but crystal oscillator does not run at exactly 16MHz. How accurate must the crystal oscillator frequency be to give a maximum error of $\frac{1}{2}$clock tick (250ns) over a period of one second? The answer is 0.25ppm (parts-per-million) which is also 250ppb (parts-per-billion). Realistically, we might want the clock frequency error to be 100ns or less so that the clock alignment uncertainty dominates. The actual goal would then be an accuracy of 100ppb or better.

Any way you look at it, typical crystal frequency tolerances of as much as a hundred PPM are not going to cut it. And that doesn't take temperature variations into account. Some technique is needed to make accurate measurements with this inaccurate clock.

As a first step, it is possible by changing the division ratio of the timer/counter in integer steps to adjust the frequency within a tolerance of about 8ppm and if an accuracy of about 8μs is adequate, the problem is solved. The author wanted to better – if for no other reason than it was a challenge.

The solution implemented in this project is to use the timer/counter as part of a software-controlled fractional-N divider[1] with a worst-case settability error of about 4ppb. A software-based phase-locked loop (PLL) is then used to keep the output of the fractional-N divider aligned with the PPS signal. It does this by making small changes in the divider's ratio until the divided clock is exactly aligned with the PPS signal.

Using this approach, the time of any timer count relative to the PPS signal may be computed with much improved accuracy. Accuracy is limited mainly by the counter/timer resolution of 500ns, but short-term drift of the crystal oscillator adds perhaps a couple hundred ns of additional uncertainty. Appendices A and B have lots of detail on implementation of both fractional-N divider and PLL.

5.3 Interrupt Design

There are two interrupt sources in this design.

- Timer match interrupts occur when timer 4 is equal to the match compare register value and resets to zero. The software adjusts the match value in real time to generate an interrupt period of exactly 1/32 of a second.
- PPS capture interrupts occur when the PPS signal causes the processor to capture timer 4's value into the input capture register, IC4. This is effectively a measurement on the phase of the PPS signal.

[1]Refer to various internet sources or textbooks to learn more about fractional-N frequency synthesis

If these two interrupts were to occur at the same time, the system would not be able to keep good time. To avoid this problem, the phase-locked loop is designed to align the PPS signal half-way between two timer match interrupts. This guarantees there will be a minimum of 1/64 second (roughly 15ms) available for interrupt processing after each interrupt.

The background tasks are also carefully scheduled so as only to run in the clear space between interrupts. This means it is not necessary for these tasks to ever disable interrupts – that might create more variability in timestamps.

The background loop checks for, and responds to NTP request packets between interrupts. Because the receipt of UDP packets in the W5500 are hardware time-stamped, there is no loss of accuracy in polling their arrival.

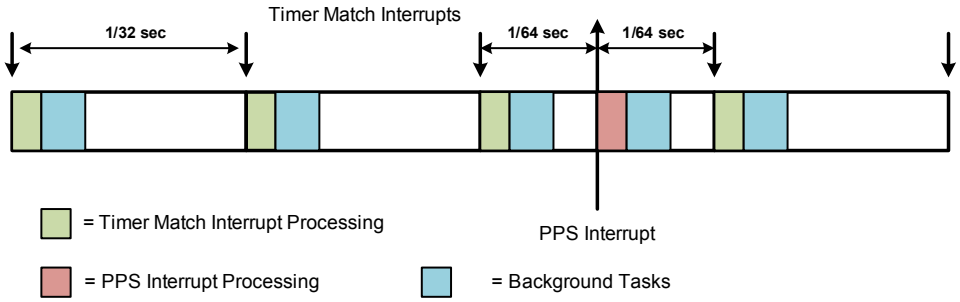


Figure 5.2: Interrupt Design

These concepts are depicted int figure 5.2. The vertical arrows on top represent the occurrence of timer match interrupts 32 times per second. The single vertical arrow on the bottom represents the occurance of a PPS interrupt.

Interrupt service routines set flags to request processing by the background loop. These flags are detected by the background loop soon after they are set which allows those tasks to run in the clear space between interrupts. After associated background tasks are complete, the flags are reset in preparation for the next interrupt.

This design requires that all interrupt and background processing does not consume more than 1/64 of a second (roughly 15ms). Measurements on the current Arduino sketch show that total processing time anywhere does not exceed about 3ms.

Chapter 6

The Advanced Variant

If you've read through the detailed appendices covering fractional-N and phase-locked-loop firmware, it is obvious that a *lot* of effort is expended dealing Arduino's less than perfect 16MHz system clock.

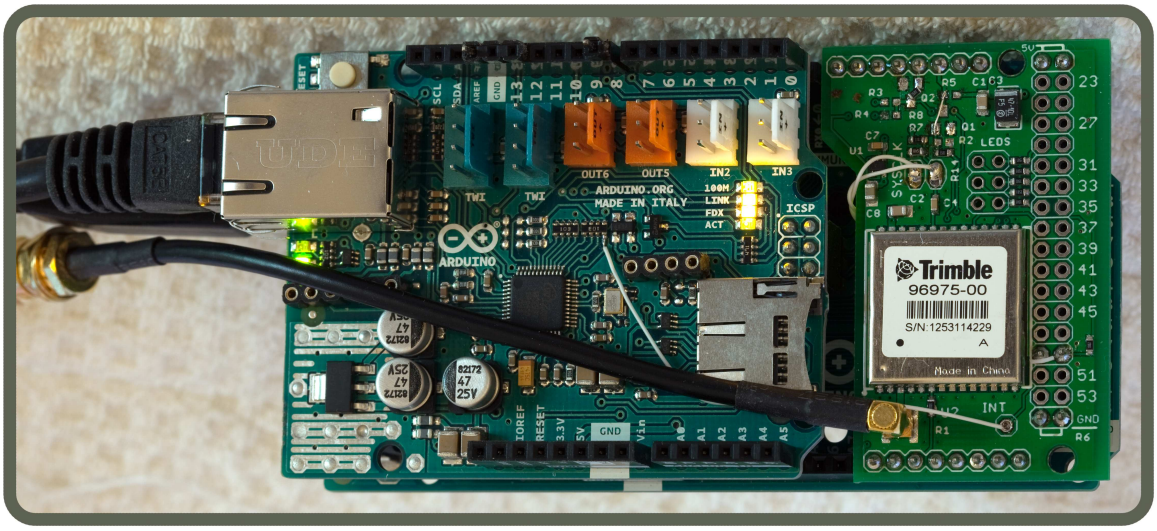


Figure 6.1: Advanced Server with Trimble ICM SMT 360 GPS module

This entire problem can be eliminated by replacing the crystal-controlled system clock with the 10MHz GPS-disciplined clock output from a GPS receiver. The receiver synchronizes the 10MHz clock with GPS time such that there are always exactly ten million clock cycles for every PPS pulse. By using this for the Arduino system clock, internal timers will always count exactly ten million clocks between each PPS and interpolating between PPS pulses becomes trivial. Figure 6.2 shows the block diagram of the advanced variant.

GPS receivers intended for use in timing applications will often provide automatic holdover on the 10MHz clock when satellite signals are lost. This feature works for anywhere between a few minutes and one or more days, depending on the receiver model.

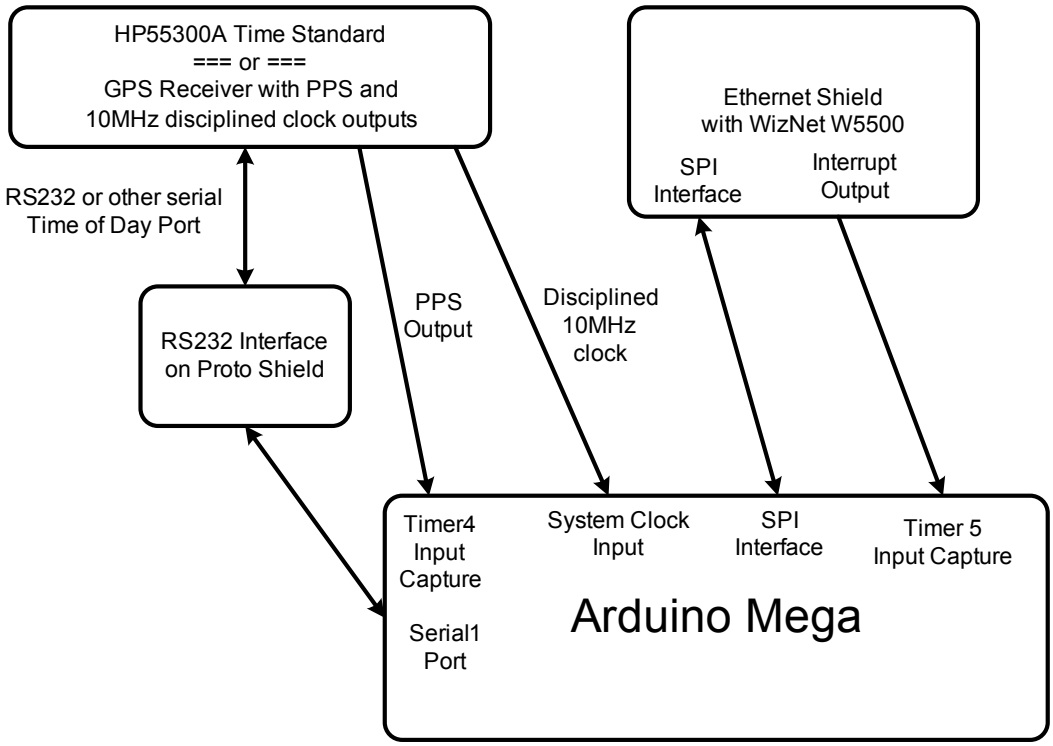


Figure 6.2: Advanced NTP server block diagram

Some GPS receivers will require an option be set to keep the clock cycle count in sync with PPS all the time. When this option is disabled and PPS error exceeds a threshold, the receiver may just move the PPS to the nearest 10MHz clock period with the result that one or two PPS intervals will not contain exactly ten million clock cycles. This behavior is sometimes called *jam sync*.

The Arduino Mega is perfectly happy running on a 10MHz system clock and all of the firmware tasks can still execute fast enough on the reduced frequency. This completely removes any need for a software-based PLL so the firmware becomes trivial in comparison to the variant described in the previous chapter. Changing

the system clock frequency to 10MHz is not without its challenges however.

- The bootloader assumes a 16MHz system clock and uses this information to generate signals at the proper baud rate when communicating with the USB port. Changing to a 10MHz clock requires modification of the boot loader.
- Arduino sketches are built assuming a 16MHz system clock. One of the Arduino installation files must be modified to add another variant of the Mega which has a 10MHz system clock. This is a simple text file and the required modifications are easy.

Those wishing to go down this road will need to re-program the Arduino Mega's bootloader and this requires a special hardware tool that plugs onto one of the interior 6-pin headers on the Mega.

6.1 Running a Mega2560 at 10MHz

This section explains the firmware changes required to operate the Mega2560 with a 10MHz external clock. Section 4.3.1 describes the hardware changes which are also required.

6.1.1 Boot Loader Changes

The bootloader used by Mega2560's is known as the stk500v2. As shown in figure 1.4, the advanced variant requires the Mega2560. While it could work with a Mega1280, that board uses a different bootloader and this project does not include changes that would be required with the Mega1280's bootloader.

The boot loader must be re-built to generate the proper baud rate for data transfers to and from USB. Normally, the Mega2560 performs sketch firmware transfers at 115,200 baud. Running these data transfers at 115,200 or 57,600 baud does not work when using a 10MHz system clock[1]. The next lower rate, 28,800 does work and can be implemented with the following changes to the loader. Modified source code and a pre-built hex file is included in the project package.

A new target was added to `Makefile` named `mega2560_10MHz`. In the new target, the CPU frequency has been changed to 10MHz and upload baud rate modified by adding `-DBAUDRATE=28800` to the end of the `CFLAGS` macro.

WinAvr was used to build the bootloader with the command line:

```
make mega2560_10MHz
```

[1]The reasons for this are not known; the true baud rates should only be in error by 1.3% or less. Perhaps it is related to the increased instruction execution time not keeping up with the boot protocol.

This creates a new hex file named `stk500boot_v2_mega2560.hex`. Program it into the processor using the ISP tool of your choice. Fuse settings do not need to be re-programmed, the settings supplied with the Mega2560 board work fine.

The WinAvr installer is evil. Do not give it permission to modify the `PATH` environment variable. Instead of adding to the path it will replace the entire path with only its required directories.

6.1.2 Arduino IDE Changes

Find the `boards.txt` file in the Arduino installation. Create a new section in the file defining the 10MHz variant. Make a copy of the original section and change CPU frequency to 10MHz and the upload speed to 28,800 baud. Do not copy the `vid` and `pid` portions of the original section into the new section. Figure 6.3 shows an example of what the new section might look like.

If you want to use the Arduino IDE to program the bootloader, then rename the hex file (e.g. to `mega2560_10MHz.hex`) and place a copy of it in the `bootloaders\stk500v2` subdirectory within the Arduino installation directory. You'll need to poke around to find this location, but in some installations it is (relative to the installation folder) here:

`hardware\arduino\avr\bootloaders`

The file name you use must match the `bootloader.file` property specified in `boards.txt`. This is demonstrated in the example in figure 6.3.

6.2 Bootstrapping

When reprogramming the bootloader it is important to have a working 10MHz clock available. The new bootloader can be programmed while running on the 16MHz clock. After that however, a running 10MHz clock must be connected in order to do anything with the Mega. You can connect the 10MHz clock either before or after programming the bootloader.

Try programming the simple blink sketch to verify that all is well at 10MHz.

6.3 The 10MHz Sketch

For the advanced variant there is no fractional-N divider, phase-locked loop or feed-forward frequency correction. The disciplined 10MHz clock eliminates the need for all of that complexity. Timers 4/5 are still used to record PPS and UDP packet events but very little timer management is required. The timers will count a cycle from 0 to 62,499 160 times for every PPS signal.

```
##############################################################

mega10mhz.name=10MHz Arduino Mega or Mega 2560

mega10mhz.upload.tool=avrdude
mega10mhz.upload.maximum_data_size=8192

mega10mhz.bootloader.tool=avrdude
mega10mhz.bootloader.low_fuses=0xFF
mega10mhz.bootloader.unlock_bits=0x3F
mega10mhz.bootloader.lock_bits=0x0F

mega10mhz.build.f_cpu=10000000L
mega10mhz.build.core=arduino
mega10mhz.build.variant=mega
# default board  may be overridden by the cpu menu
mega10mhz.build.board=AVR_MEGA2560

## Arduino Mega w/ ATmega2560
## -------------------------
mega10mhz.menu.cpu.atmega2560=ATmega2560 (Mega 2560)

mega10mhz.menu.cpu.atmega2560.upload.protocol=wiring
mega10mhz.menu.cpu.atmega2560.upload.maximum_size=253952
mega10mhz.menu.cpu.atmega2560.upload.speed=28800

mega10mhz.menu.cpu.atmega2560.bootloader.high_fuses=0xD8
mega10mhz.menu.cpu.atmega2560.bootloader.extended_fuses=0xFD
mega10mhz.menu.cpu.atmega2560.bootloader.file=stk500v2/mega2560_10MHz.hex

mega10mhz.menu.cpu.atmega2560.build.mcu=atmega2560
mega10mhz.menu.cpu.atmega2560.build.board=AVR_MEGA2560

##############################################################
```

Figure 6.3: Example of modified section in `boards.txt`

- When the first PPS pulse arrives, timers 4/5 are both preset half-way through the 62,500 count sequence. This results in PPS interrupts being displaced in time from timer overflow interrupts by one-half timer cycle (3.125ms).
- After initialization, there are only two timer management tasks:
  - Keep a cycle counter modulo 160 so captured times can be computed relative to PPS.
  - Verify that PPS arrives on the same timer count each time. If that does not happen, it indicates the 10MHz clock is not being properly disciplined by the GPS reference clock.

Both timers are configured to run at 10MHz so there is 100ns per timer tick. They are both initialized to the same value so will count in unison forever more.

6.3.1 Interrupt Design

This part of the firmware is mostly identical to what is done in the basic variant, see figure 5.2 for more information.

6.3.2 Arduino Timing Accuracy

The timing functions `millis()` and `micros()` are not mathematically exact unless `F_CPU` happens to be a friendly value, and 10MHz is not a friendly value. As a result, millis() and micros() cannot be used for critical timing with a 10MHz system clock – even though the clock itself is extremely accurate.

Chapter 7

Client/Server Operation

Only rudimentary explanations of NTP operation are provided here. See the web sites below for all of the details.

`https://www.eecis.udel.edu/~mills/ntp.html`

`http://www.ntp.org`

The most common mode of NTP operation is called client/server. Here, the client wishing to synchronize its clock sends a request to the server, and the server replies with clock readings needed to synchronize time. When these transactions are complete, the client will have obtained the following time values:

- Approximate time the UDP request packet was transmitted.
- Accurate time the UDP packet was received by server.
- Approximate time reply packet was transmitted by server.
- Accurate time reply packet was received by client.

Accurate times are those captured either in hardware or OS device drivers. They are typically accurate to a few microseconds or better. It is the approximate times which limit the accuracy of NTP in client/server mode. Approximate times are largely the result of a Catch-22 in the design of NTP's client/server protocol.

Each transmitted UDP packet includes a declaration of the exact time it was transmitted (the transmit timestamp). Problem is, the sender does not really know when the packet will be transmitted...until it's transmitted...and then it's too late to change the transmit time encoded in the packet. Competing Ethernet traffic on a busy network can cause unpredictable delays in the packet's transmission. As a result, transmit timestamps in a UDP packet can by in error by a hundred microseconds or more on a busy Ethernet network.

The best possible performance in client/server mode occurs on a network without un-predictable delays. This can be achieved if there is no network traffic other than NTP packets. For example, a dedicated Ethernet card on a PC could be connected directly to the NTP server with a cross-over cable. The NTP server sketch uses the following procedure to obtain the most accurate transmit timestamps possible on a quiet network connection.

1. Acquire the current accurate time, t_1.
2. Add a fixed delay to this to get $t_2 = t_1 + \delta$.
3. Add another fixed delay to this to get $t_3 = t_2 + \epsilon$.
4. Place t_3 into the outgoing UDP packet as the transmit time.
5. Load the packet into the NIC's output buffer; at this point a single command to the NIC will begin the transmission process.
6. Wait until the current time is t_2 and kick off the NIC transmission.

Above, δ is chosen to be a little bit longer than it takes to build the reply packet and load it into the NIC's output buffer. This guarantees the sketch will be ready to kick off transmission before time t_2.

The value of ϵ is equal to the time it takes the NIC to transmit the packet after being kicked off (on a quiet network w/no other traffic). Both of these delays can be measured by an oscilloscope if some unused digital pins on the Mega are used as flags. The ending of the ϵ delay is signaled by the NIC's `SEND_OK` interrupt. In the current version of the sketch, δ has been set to 800μs and ϵ is around 25μs (depending on the variant of server).

This scheme works pretty well as long as network traffic is quiet. On a busy network all this work is for naught, and variable delays become the dominant factor determining accuracy of the NTP server.

Performance can be improved further through use of NTP's interleaved symmetric mode of operation. This mode replaces approximate transmit timestamps with exact transmit times which are sent after-the-fact in a subsequent UDP packet.

Chapter 8

Interleaved Symmetric Operation

The sketch supports interleaved symmetric active peer mode of operation within the NTP protocol. This mode drastically reduces the effect of variable network delays that occur when the NIC tries to get onto the wire. Figure 9.5 shows how much improvement is possible by using the interleaved symmetric mode.

8.1 Normal Symmetric Mode Behavior

Tutorial descriptions of this mode (found on various web sites) are quite detailed but unfortunately for this author, difficult to understand. Interleaved symmetric operation is described here as understood by the author after reading various tutorials. There is no guarantee this author got it right, and some of the details in this description may be incorrect.

In reading tutorial descriptions of packet flow in symmetric modes, it is easy to miss some of the important aspects of operation. Figure 8.1 is the usual diagram used to show how basic symmetric mode works. From looking at the diagram one might conclude that the arrival of the packet at peer B at time T_2 causes B to send a packet back to peer A at time T_3. Wrong! In reality, the two peers are sending out packets on completely different and independent schedules. That is described this quote from one of the tutorial web pages:

> *"In symmetric modes each peer independently polls the other peer, but not necessarily at identical intervals. Thus, one or the other peer might receive none, one or more than one packets between polls."*

In some sense, the word *poll* is a misnomer, as it implies the process of asking a question and receiving a reply. While a reply of sorts may eventually be received, there is no guarantee the each *poll* is going to receive a reply, nor is there any responsibility on a peer's part to respond to any given poll.

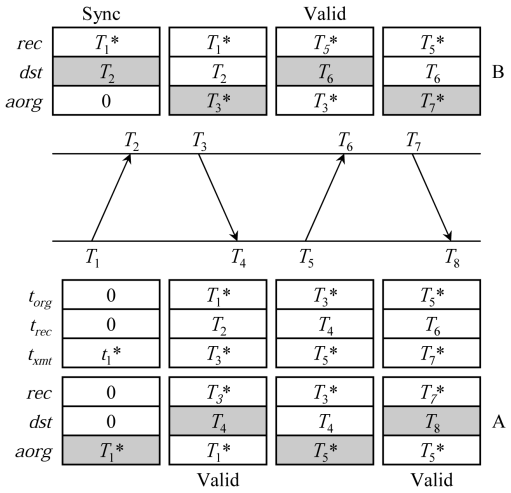


Figure 8.1: The usual simple diagram of Basic Symmetric mode.

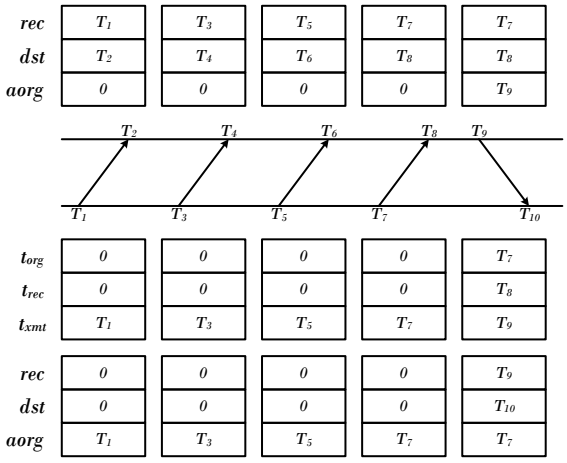


Figure 8.2: A more complex example of Basic Symmetric mode.

For example, A might be sending packets every 16 seconds while be is on a 64-second schedule. As a result, B might receive as many as four packets from A before deciding to send a packet to A. In reality then, this packet is not really a *response* per se to any of A's packets (other than containing timing data regarding the latest packet received from A). Figure 8.1 does not depict this behavior. Figure 8.2 shows the first volley would look like in this case. Each of the first four packets from A to

B cause B's state to be updated but there is no immediate response from B. Only after the fourth packet from A (when B decides that it is time to poll A) does B send the most recent timing data from A.

These details of timing apply equally to the interleaved symmetric mode. The general concept is that A and B independently checking their clocks against each other and do it on independent time-tables. With an understanding of these details, the description of modified behavior practiced by the NTP sketch will make more sense.

8.2 Modified Symmetric Mode Behavior

The NTP implementation in this sketch is a little different. It ignores one of the basic tenets of symmetric mode – that two peers are independently measuring their clocks against each other. In this sketch, the assumption is that all peers are only interested in synchronizing their clocks to the sketch's clock; the sketch does not measure any of the peers' clocks w.r.t its own clock.

The reason for this perversion of symmetric mode behavior is that interleaved symmetric mode is the only way to get transmit hardstamps to a client in NTP. The sketch is really treating symmetric peers like clients some sense of the word.

8.3 Symmetric Mode Basics

Some of the basic aspects of symmetric mode behavior were skipped over until now. The aspects of this behavior which are relevant to the modified manner in which this sketch uses symmetric modes can now be discussed.

Figure 8.3 shows how input NTP packets are transformed to output packets in the basic and interleaved symmetric modes. This makes is crystal clear that the two modes are delineated by which timestamp field from the incoming packet is used to identify a *round*. The basic mode uses transmit stamp fields and the interleaved mode uses receive stamp fields.

The other difference in the two modes is that basic mode places a softstamp in the t_{xmt} field of the outgoing packet, while interleaved mode inserts a hardstamp from the *previous* packet volley.

A very important detail is how each peer uses the data being passed around to compare its clock with a peer's clock. Since this sketch does not measure any peers' clocks, we have ignored it in this description. As long as the sketch inserts the proper data into each transmitted packet the remainder is dealt with by the peer running `ntpd`, and is of no further concern here.

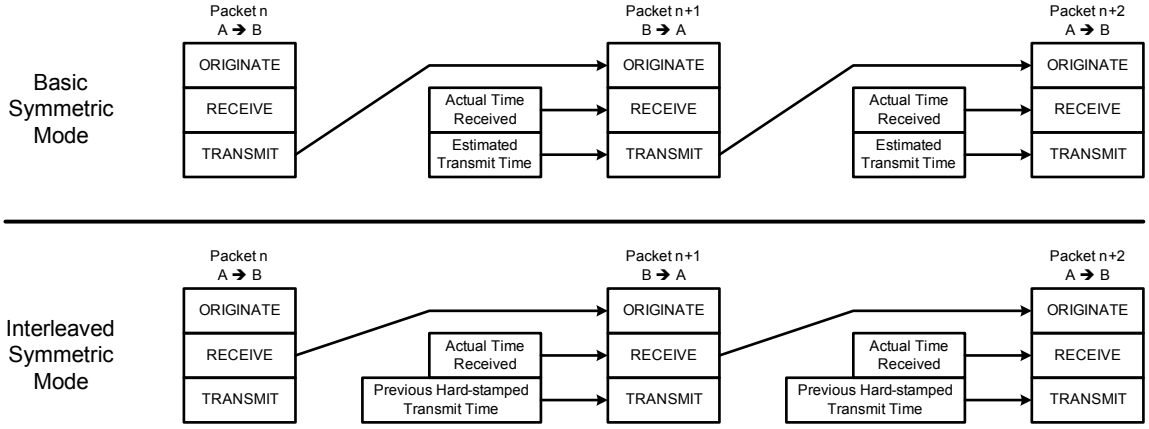


Figure 8.3: NTP Symmetric Peer Modes

8.4 Implementation

The NTP server implements the symmetric modes in a slightly different manner w.r.t. packet transmit timing. The sketch treats symmetric modes partly like a client/server association in the sense that it only transmits symmetric mode packets as a response to an incoming symmetric mode packet. The transmitted packet is otherwise valid in all respects. While the basic interleaved mode is supported in this implementation, there is not much point in using it because it suffers the same problems as client/server mode, caused by variable network delays.

One nice thing about interleaved symmetric mode is that there is no need to create an estimated transmit timestamp and then wait until just exactly the right moment to send it; packet processing can send out the response packet without delay, as soon as it is ready.

Reference implementations documented extensively on various NTP-related web sites use seven or eight state variables for each peer in the interleaved symmetric mode. These variables have what this author finds to be confusing names, different names are used in this implementation. For each peer, the sketch keeps the following state variables (this is not a complete list):

- Interleaved mode flag.
- Basic round ID; transmit timestamp from the most recent outgoing packet.
- Interleaved round ID; receive timestamp from the most recent outgoing packet.
- Transmit timestamp from previous incoming packet; used to detect duplicates.
- Transmit hardstamp for previous outgoing packet.

8.4.1 Active and Passive Peers

NTP distinguishes between active and passive symmetric peer associations. In the passive case, the NTP node has not been configured with the other node as a symmetric peer. This NTP node implementation treats any symmetric mode packet as if it came from a configured peer. As such, all symmetric mode packets are considered to be active, not passive.

8.4.2 Peer State Allocation and Storage

Since state must be kept for each symmetric peer, this server places a limit on how many symmetric peers are supported. This is configurable in the file `NtpConfig.h`. The state data for each peer requires 40-bytes of storage.

8.4.3 What is Bogus?

Cryptic references to a packet being *bogus* will be found in the sketch. In NTP parlance, this word refers to an incoming packet whose round ID (the originate timestamp field) does not match the recipient's idea of the current round ID. For example, in interleaved mode the round ID is equal to the receive timestamp field from the most recent outgoing packet. If the next incoming packet's originate timestamp field does not match, it is considered *bogus*.

8.4.4 Peer State Management

Managing peer states works as follows.

- For each packet received, an array of these structures must be searched for a match on the peer's IP address.
- If the peer is found in the table, a pointer is returned and used by packet processing as the peer's current state and updated as necessary.
- If there is no such peer in the table, a new entry is initialized and the pointer returned for packet processing.
- If the table is full, then it is searched for any entries that may have timed out and can be replaced. If so, a new entry is initialized at the timed out location and returned. Otherwise a table-full indication is returned.

8.4.5 Non-interleaved Mode

There is an inconsistency between on-line NTP documentation and source code in the 4.2.8 release of NTP. On-line documents indicate that the interface to an interleaved peer starts out in the interleaved mode but can fall back to non-interleaved operation if the peer does not support interleaved mode. The source code however implements just the opposite behavior – the interface starts out in non-interleaved mode and will switch to interleaved mode if it receives valid interleaved-mode packets. This behavior is implemented in lines 1221 through 1237 in the file `ntp_proto.c`.

While the sketch does implement the ability to switch from basic to interleaved mode, it is effectively disabled because the interface is always initialized to interleaved mode. Once in interleaved mode, the receipt of basic mode packets will not cause a switch back to basic mode. This is intentional: there is no performance advantage to basic peer mode compared to client/server mode.

Chapter 9

Testing and Calibration

Notes about testing the server's performance are presented here. These apply generally to both variants and is presented here as a separate chapter. There are two types of testing described here:

- Internal testing generally involves measuring signals and comparing with firmware generated values. Analysis of PLL performance by analyzing firmware debug outputs is also part of this process.
- External testing requires connecting the NTP boxes to a test network and analyzing performance measured at an independent node on the network (such as a Linux PC).

9.1 PLL Performance

In this section, measured PLL performance for the basic variant are presented. Refer to appendix B for explanation an explanation of the artifacts discussed below.

In figure 9.1 is a plot showing the phase (a.k.a time) of PPS signals measured by the frac-N divider over about a 30-minute period. This represents typical tracking behavior of the PLL. The raw phase samples are plotted as blue x marks while the result filtered by the 16-tap FIR loop filter is in red. It is the filtered signal which generates feedback corrections to the frac-N divisor.

Readily apparent are the artifacts generated by over-sampling the frac-N subsystem as discussed earlier in this article. The FIR filter does a fair job of filtering out much of the noisy artifact behavior.

When the frac-N term k approaches an integer multiple of 64, as shown in figure 9.2, the FIR filter becomes irrelevant. What is apparent in this figure is the response of the PLL to a large phase step. Although there is significant overshoot, this performance is still better overall than with loop gains lowered to eliminate most of

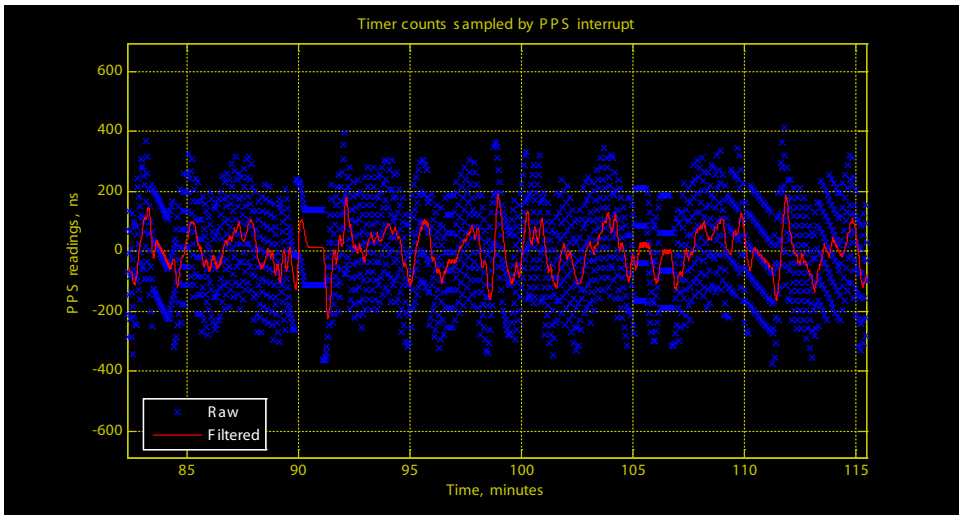


Figure 9.1: Typical Short-Term PLL Performance

the overshoot. The phase variations plotted in this figure are mostly mathematical artifacts caused by a very slow frequency drift in the system clock.

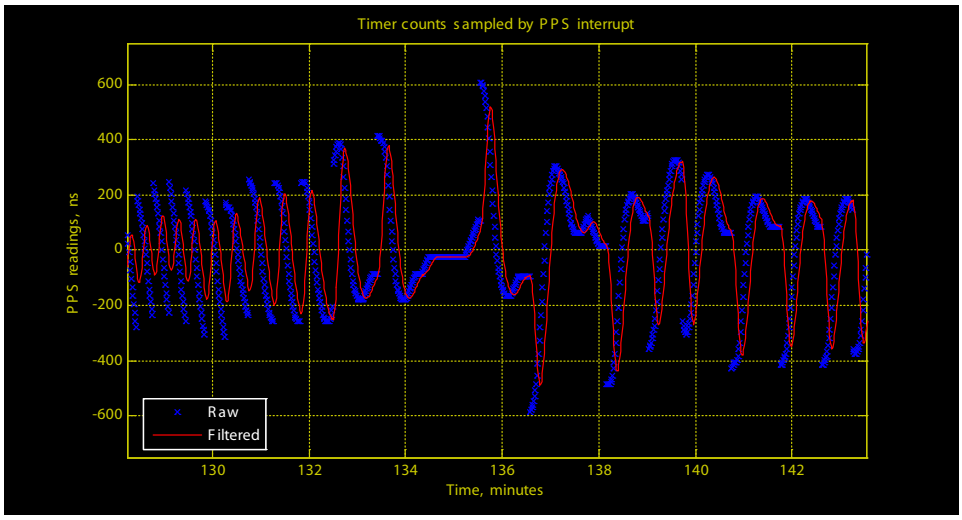


Figure 9.2: Performance with k passing integer multiple of 64

9.2 Network Test Configurations

Early tests compared one local NTP server to several external servers on a PC running Windows 7. It was discovered that this version of Windows does not have adequate clock performance to make these tests meaningful.

A better confirmation of performance was obtained by comparing the times reported by two NTP servers built for this project. One is a basic variant with HP55300A reference clock; the other an advanced variant with the Trimble GPS receiver. The NTP daemon on Linux was configured to use the advanced variant as a symmetric peer with the `prefer` option – this forces the daemon to use that server as the only input to the clock discipline algorithm. This allowed a comparison of the two NTP servers by examination of the `peerstats` log file.

Three different network configurations were used which are depicted in figure 9.3. These three configurations and the names used to refer to them are explained below.

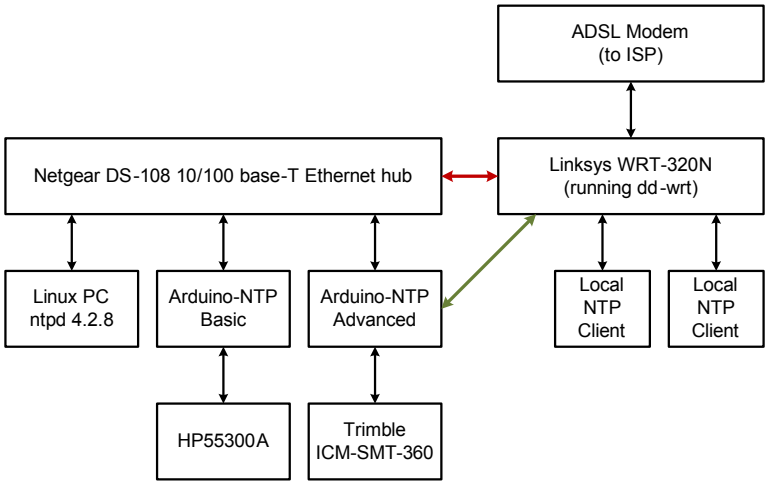


Figure 9.3: Network configuration used for testing

9.2.1 Quiet

In this configuration, the Linksys router is disconnected and both NTP boxes are connected to the Netgear router. Conceptually, the red and green arrows are deleted in this setup. During these tests the only network traffic was NTP packets to both NTP boxes initiated by the Linux PC.

9.2.2 Busy

Here, the Linksys router is connected to the Netgear router (red arrow), and both NTP boxes are connected to the Netgear router (green arrow is deleted). There are two Windows PCs on the Linksys router running `ntpd` and querying both NTP boxes. There is no NTP traffic to the NTP boxes from the ADSL modem.

9.2.3 Two Hops

In this scenario, the Busy configuration is modified by disconnecting one of the NTP boxes from the Netgear router and re-connecting to the Linksys router (green arrow).

9.3 Measurements

The NTP daemon in all tests was configured to output both `loopstats` and `peerstats` log files.

9.3.1 Delay

Delay statistics are important in analyzing overall performance of the NTP servers. With all NTP endpoints connected to the same network hub, delays between endpoints should be constant, and relatively insensitive to network traffic levels. While there could be variable delays for an NIC to get access to send a packet, the actual time required to send the packet should be fairly constant. Delay measurements should also exhibit low values of jitter or standard deviation.

9.3.2 Offset

With the NTP daemon (Windows or Linux) configured to use only one of the NTP servers as input to clock discipline, the offset statistics should be stable if the NTP server itself is stable.

9.4 Windows 7 Test Results

Typical clock loop and offset behavior for Windows 7 is shown in figure 9.4. In these tests, both servers were configured as interleaved symmetric peers in the Quiet network configuration.

While the two servers are in relatively close agreement, the millisecond-level variations in offset are too large to make a meaningful comparison. This sawtooth type of behavior is typical clock discipline performance for NTP running on Windows.

The only thing it really demonstrates is that Windows 7 is not a suitable platform for testing the project's NTP servers.

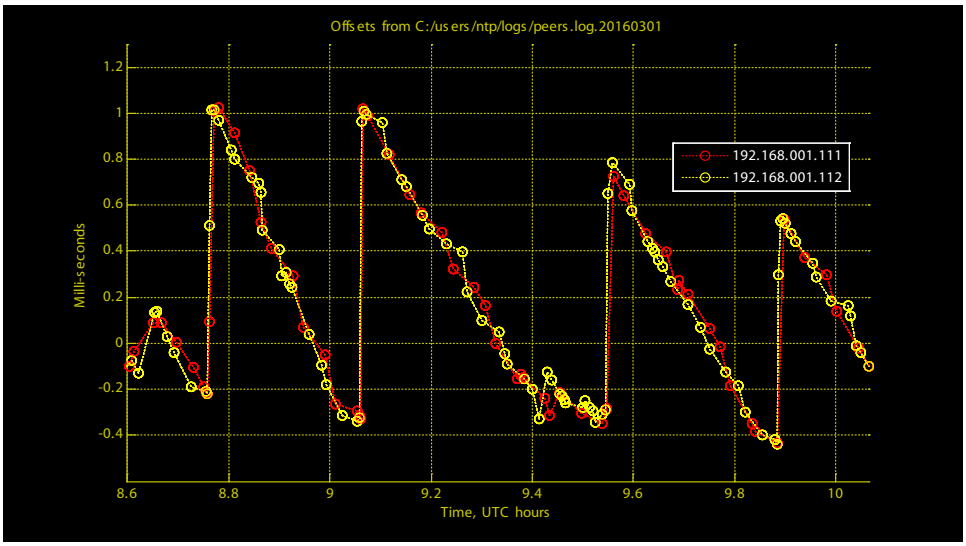


Figure 9.4: Clock discipline behavior on Windows 7

9.4.1 Windows 7 Test Notes

It was necessary to set the system environment variable below on Windows 7. Without this, `ntpd` would sometimes disable interpolation depending on what state the system's multi-media timers happened to be in when the service started. With interpolation off, you only get a resolution of 1ms on NTP statistics like delay and jitter. While enabling interpolation may cause issues when other applications modify multi-media timers, it is necessary to get the higher resolution for some of the tests that were being run.

```
NTPD_USE_INTERP_DANGEROUS=1
```

9.5 Linux Test Results

Refer to figure 9.3 for the network configurations shown in these graphs. The upper plot shows delay to one of the project's NTP boxes. The lower plot is the same measurement using interleaved symmetric mode. Delay averages in the three network configurations are shown with yellow, green and cyan lines on top of the actual data points (red x's) from the `peerstats` file.

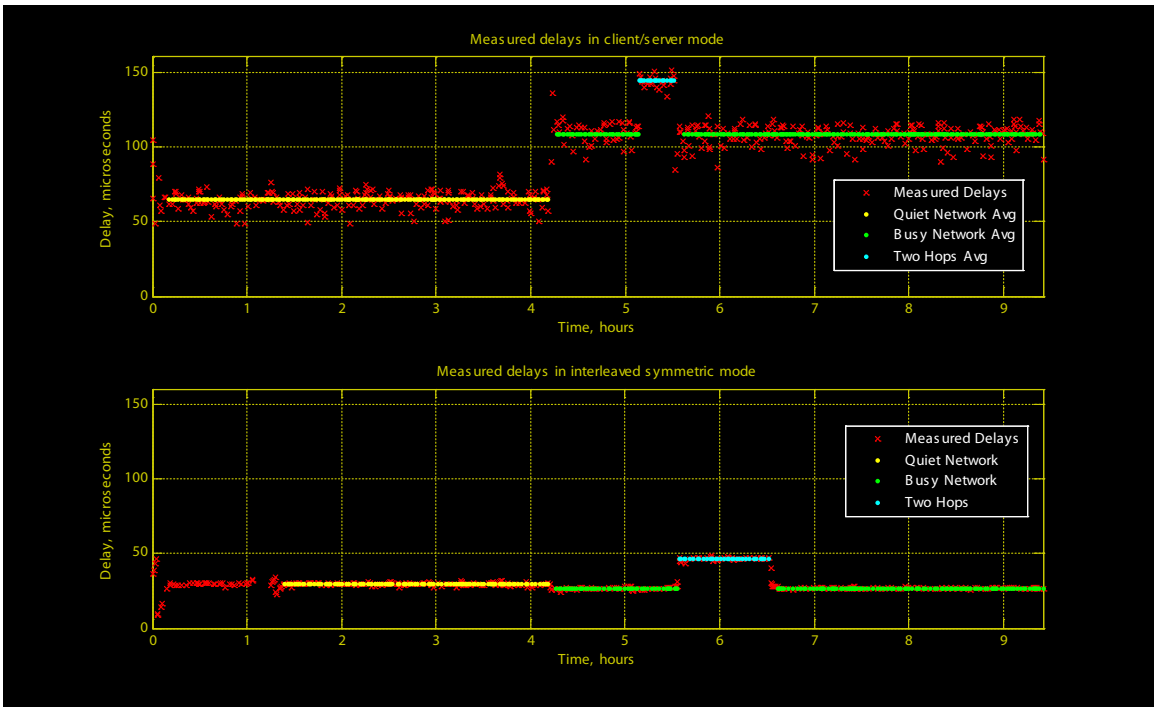


Figure 9.5: Delay measurements in different NTP modes.

The first obvious difference is that measured delay values are much less noisy in interleaved symmetric mode (hereafter, ISM). In fact, the standard deviation of measured delays is about six times smaller in ISM.

Ideally, measured delay should not be sensitive to network traffic levels with everything connected to a single Ethernet hub. In client server mode, the measured delay increases by about 50μs when the second network hub is connected. Interleaved symmetric mode on the other hand shows only about a 3μs decrease with increased network traffic.

With the third network configuration (two hops), client/server mode indicates a larger increase in delay than with interleaved symmetric mode.

9.5.1 Impact on Accuracy

In figure 9.3 measured delays are noisier and change more with changes in network configurations in the client/server mode. This translates directly to more noise and larger errors in measured clock offsets. Figure 9.6 shows measured offsets to the two NTP boxes during the same test. Raw test data is shown with x's and running averages with solid lines.

In figure 9.5 network traffic increases when the Linksys router is connected, just after four hours on the X-axis. In figure 9.6, there is virtually no change in measured

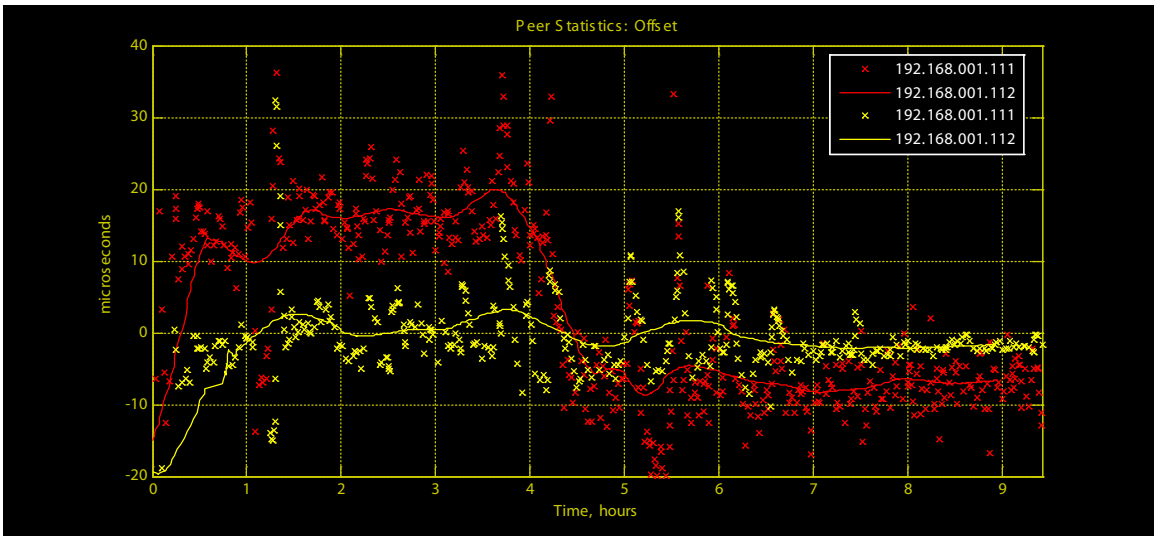


Figure 9.6: Offset measurements in different NTP modes.

offsets at four hours for interleaved symmetric mode, but client/server mode shows a 25μs change. This is the amount of offset change one would expect from a 50μs change in measured delay (assuming the change in delay were due to a timestamping error).

Appendix A

The Phase Locked Loop

There is a lot of very technical material to follow in both this and the next appendix. It all has to do with fixing the inaccuracy in the 16MHz system clock so that we can accurately measure time in between PPS signals.

All of this detail is included here because the author had to work this all out to make the firmware work. This is complex to the point that it had to be written down in some detail to get it right.

A.1 Block Diagram

In practice, the software sketch keeps Arduino's clock in sync with the PPS pulse generally within ±1 timer count (500ns). The processor provides a nominal 2MHz clock to the 16-bit timer 4 (T4) which is then nominally divided by 62,500 to yield a timer match interrupt 32 times a second. These numbers are all *nominal*. In reality the processor clock is *not* 2MHz and requires compensation if accurate timestamps are to be generated.

Figure A.1 shows a block diagram of how Arduino's crystal oscillator is compensated to achive the goal. The input on the left, labeled f_R is the reference frequency provided by the reference clock. In this model it is a constant value representing for example 2MHz exactly, or 1Hz exactly (the PPS signal). This block diagram is conceptual and this constant value does not appear anywhere directly in the software – the closest equivalent would be the PPS signal coming from the reference clock.

In this model, phase is considered to be the integral of instantaneous frequency and conversely, instantaneous frequency is the derivative of phase.

$$\phi(t) = \int_0^t f(t)dt$$

$$f(t) = \frac{d\phi(t)}{dt}$$

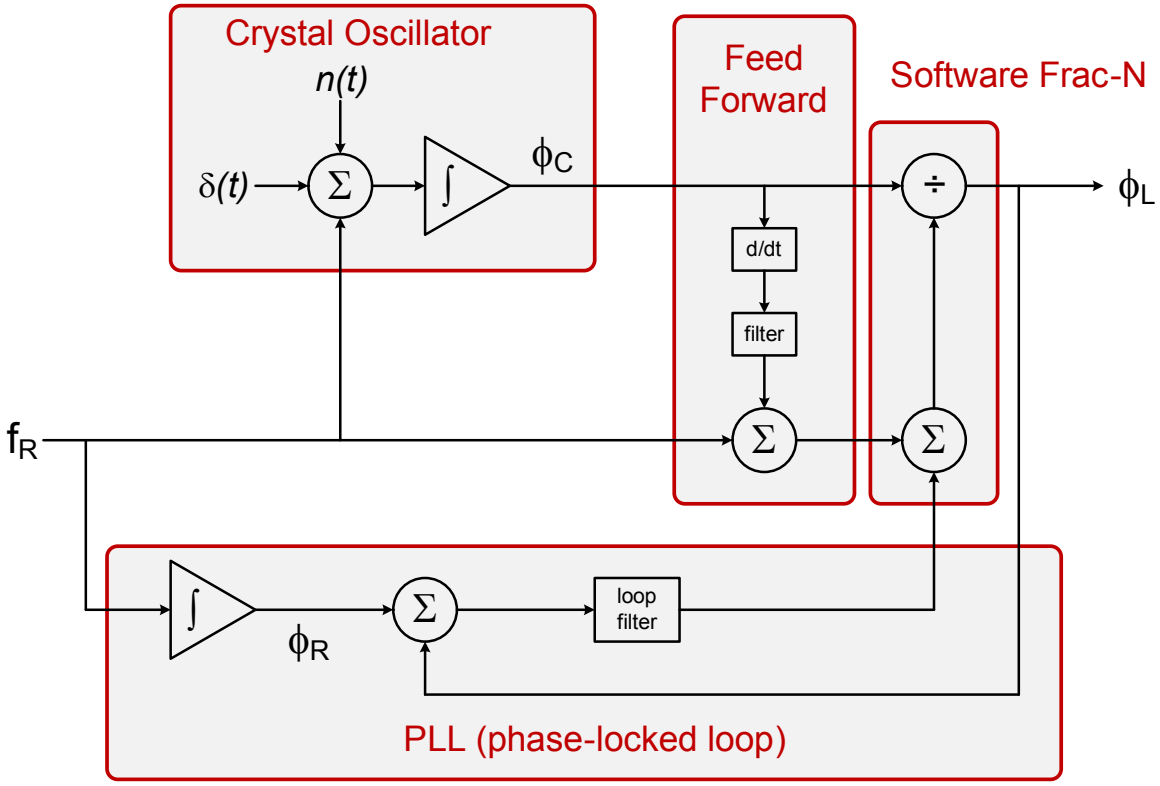


Figure A.1: Phase-Locked Loop (conceptual model)

In these equations phase is in units of cycles – not radians. It is also possible to talk of phase in units of time, nano-seconds or micro-seconds for example. When using time units for phase one must also specify some reference frequency such as 1 Hz for example. In this example, a phase of 1μs is equal to 10^{-6} cycles. In what follows phase will often be discussed in unit of time relative to a frequency of 1Hz – the PPS pulse frequency. In this sense, the terms *phase* and *time* will be used interchangeably below.

A.1.1 Crystal Oscillator

The crystal oscillator integrates the reference frequency plus unwanted short ($n(t)$) and long-term ($\delta(t)$) errors to produce the output phase, ϕ_C.

A.1.2 Feed-forward

The feed-forward block differentiates this phase to estimate frequency, low-pass filters the result and compares it to the reference frequency to generate a correction. It then applies the correction to the fractional-N divider. This gets the resulting frequency very close to the desired value, leaving only small errors for the PLL to deal with. The main reason for using feed-forward in this design is to alleviate the need for 64-bit integer math in the PLL code.

A.1.3 Phase-locked Loop

The phase-locked loop samples the output phase from the divider once per second using the PPS signal. It also has a reference phase ϕ_R generated by integrating the reference frequency. A correction is generated by comparing the actual phase ϕ_L to the reference phase and running the result through the loop filter.

The loop filter contains an integrator (it is a type-I control system) to allow frequency ramps to be tracked with zero following error.

It is useful to remember here that the block diagram is conceptual. The software does not actually generate an infinitely increasing ϕ_R as implied; instead it counts phase (time) modulo 32 timer cycles (one second) so there is no real need for an infinite phase ramp.

A.1.4 Software Frac-N

Conceptually, the fractional-N block can be viewed as simply dividing the phase output of the crystal oscillator by a variable number (determined by the output of the PLL's loop filter).

A.2 Feed-forward Correction

In this and following sections, additional detail is provided for individual blocks shown in figure A.1.

To allow the PLL's digital loop filter to run with high fractional precision in 32-bit integer math, the numerical values of the phase error must be kept relatively small. To achieve this goal, the frequency difference between the PPS signal and crystal oscillator is continuously measured and used to generate a feed-forward correction. This correction does not depend on the output of the fractional-N divider – it only looks at raw timer counts accumulated between PPS signals. This removes a majority of frequency error leaving only small offsets for the PLL to manage.

At every PPS interrupt, the number of timer clock cycles since the last PPS is recorded. Ideally there should be two million cycles (with a 2 MHz timer clock).

The difference is a measurement of the clock's frequency error. These errors are passed through a digital filter to smooth what is mostly quantization noise due to sampling the PPS signal with a 2MHz clock. Because the 2MHz clock is quite stable over a period of a minute or so (not accurate – but stable), the digital filter is also able to provide more resolution on the clock frequency error by effectively averaging the individual frequency error measurements.

The output of the filter is then used to adjust the fractional-N ratio. In the field of control system theory, this type of correction is called *feed-forward* – because the result of the correction does not feed back into the next measurement.

The digital filtering process introduces some delay into the correction process. A sudden change in clock frequency will not appear immediately at the filter's output. When temperature is changing over time, it produces a changing clock frequency over time also and the digital filtering will lag behind the resulting frequency ramp. This is another source of residual frequency error which is not removed by feed forward. As a result, feed-forward correction does not remove all of the frequency error in the clock. In addition to phase control, the feedback portion of the phase-locked loop also removes residual errors from this process.

When the PLL first starts up, a 64-tap FIR filter is used to produce the feed forward correction. A third order IIR filter with lower cutoff frequency is also started in parallel with the FIR filter. Once the IIR filter has settled it is then used instead of the FIR filter. It requires three to five minutes for this switch-over to occur.

The feed-forward design keeps the resulting integrated phase error over a one-second interval to be typically within a few timer counts or a couple of micro-seconds. Thus, inputs to the digital loop filter (which uses units of ns) are kept small enough to allow 32-bit integer math to be used there.

A.3 Phased-Locked Loop

This is (in software) a typical proportional-plus-integral (PI) control loop. Phase-locked loops are inherently non-linear and difficult to analyze; that will not be attempted here.

Suffice it to say that feedback gains have been hand tuned to give good overall performance. For large phase steps there is what many would consider excessive overshoot with some ringing. However, once the loop is settled, these gains give better tracking performance than lower values which result in less overshoot.

A.3.1 Zero Reference Phase

For reasons explained below (see the *Interrupt Design* section below), the PPS signal is phase aligned to occur exactly half-way through a nominal timer cycle. This is

achieved by adding a numeric offset of 31,250 (one half nominal timer cycle) to measured phase (time) when computing phase error in the PLL.

A.3.2 Loop Filter

Due to the existence of frac-N artifacts[1], it is desirable to pass measured phase errors through a low-pass filter. This does not remove all artifacts and when the fractional divisor is close to an integer multiple of 64 it will not be of much use. On the other hand, overall loop performance is improved with the filter in place.

The low pass filter adds some delay to the loop and delay is the enemy of control loop stability. As a result, the amount of delay has been kept to a minimum, meaning the cutoff frequency is higher than would otherwise be desirable.

The filter implemented in software is a 16-tap symmetric FIR filter with linear phase and has a delay of 7.5 samples (seconds).

The output of the low-pass filter is applied to a simple integrator. The integrator and low-pass filter outputs are then summed with hand-tuned gain constants to produce a PI (proportional-plus-integral) correction term for the frac-N divisor.

A.3.3 Loop Initialization

After initial power-up, the PPS signal will be arbitrarily aligned somewhere within the the frac-N cycle consisting of 2048 timer cycles. It will also be arbitrarily aligned within an individual timer cycle – not necessarily half-way through where we want it to be.

While the loop would eventually find phase lock under these conditions it could take a very long time. To achive a reasonable initial lock-up time, two actions are taken when the first PPS pulse arrives.

- Hardware timers T4 and T5 are reset to a point about half-way through their cycles. The exact value has been hand-tuned for a fast initial lock-up.
- Cycle count variables in the frac-N sub-system are initialized so the next PPS signal will occur at the start of a frac-N cycle.

A.3.4 Holdover

With the HP55300A there is no need for a holdover function in the Arduino sketch. The reference clock can keep accurate time without a satellite signal for a full day or more.

When using a GPS module with PPS output in place of the HP55300A, it may not be unusual to lose the satellite signal (and PPS) for short periods of time. In

[1] See the section *Artifacts* below.

this scenario a holdover function starts running in the sketch. The basic holdover operation just keeps the current frac-N divisor constant while the PPS signal is lost. This works okay as long as temperature is not changing. With typical changes in room temperature however, it can easily lose 40μs or more over just a 5 minute outage.

A.3.5 Frequency Prediction

To improve on the holdover performance when temperature is changing, an experimental holdover prediction block has been implemented. It uses a recent measurement of frequency changes (over the last 10 minutes) to predict frequency over the next 10 minutes and adjusts the frac-N divisor accordingly. In several tests it was able to maintain time within about 10μs or better over a period of 10 minutes.

Once a minute, the output of the frequency feed-forward filter is saved to a 11-element FIFO. The time between first and last elements is therefore 10 minutes and an estimate of linear frequency change can be made in this way. For the first 10 minutes of holdover, the frac-N divisor is adjusted according to the estimated frequency ramp to improve tracking. After 10 minutes no further corrections are provided as predictions too far in the future tend to become quite inaccurate.

If the FIFO history is not available, then the current frac-N divisor is kept constant during holdover. The FIFO length could perhaps be enlarged to cover 20 minutes but that would probably be as far as things could be stretched.

Being experimental the feature is rather strict about applying corrections. After any holdover event ends, a minimum of 15 minutes of valid PPS signals must occur before the frequency ramp correction will operate again.

This capability is experimental and may be disabled in the `NtpConfig.h` file.

Appendix B

Fractional-N Divider

Given a fractional-N divider running in software, the challenge becomes one of figuring out what the exact phase is at any given timer count within the frac-N cycle. This is a non-trivial problem, and the algorithm used in the sketch is explained in this section.

B.1 Fractional-N Overview

There are many resources on the internet where one can find a tutorial on fractional-N (or frac-N) frequency dividers. A short introduction is provided here to help explain the process of phase computation. See other internet sources for a more thorough explanation of frac-N.

Given a clock signal with frequency f_c, we can divide it by counting cycles up to some integer value, m to define the period of the divided clock:

$$f_d = \frac{f_c}{m}$$

Suppose that we wish to synthesize a clock at some arbitrary frequency, f_r by dividing the clock at f_c. In most cases the ratio between the two clock frequencies will not be an integer:

$$r = \frac{f_c}{f_r} \; ; \; \lfloor r \rfloor \neq r$$

Thus, simple integer division by m will result in some frequency error. The error can be reduced by periodically dividing f_c by $m + 1$ instead of m. If this is done once for every n cycles of the divided clock, then the number of cycles of f_c elapsed for n cycles of the divided clock will be $nm + 1$. The resulting average frequency is

$$f_d = f_c \, \frac{n}{nm + 1} = \frac{f_c}{m + 1/n}$$

Although we've succeeded in creating a clock with an average frequency divided by a non-integer value, the resulting divided clock will be jittery because every n'th timer cycle will be a different length. Dealing with the jitter turns out to be the real challenge, and fixing that will be discussed below. But for now let's ignore the jitter problem.

So now we know how to divide by $m + 1/n$ but that doesn't give us access to all possible integer ratios. For example we can divide by $3\frac{1}{2}$and $3\frac{1}{4}$, but not $3\frac{3}{4}$. This technique can easily be modified to do this.

Instead of adding a single clock every n timer cycles, if we add k clocks then the total number of clocks in every n cycles will be $mn + k$ instead of $mn + 1$ as we first considered. The resulting average divided frequency now is:

$$f_d = f_c \frac{n}{nm + k} = \frac{f_c}{m + k/n}$$

If we're going to add k extra cycles every n timer cycles, then just exactly when do they get added? One could just add them all at the beginning or end. This approach has two drawbacks:

- The cycle where these counts are added can become quite a bit longer as a normal cycle as k approaches n.
- The hardware timer being used to divide by m will need to count to $m + k$ during the extended cycle. There may not be enough bits in the counter to do this.

To get around this, the k extra clocks can be evenly dispersed, one at a time throughout the n cycles.

Now, n is not always going to be an exact multiple of k so the number of timer cycles between extra counts will not be perfectly even. For example with $n = 8, k = 3$ we could add extra counts on timer cycles 0, 3, 6 and 8.

To implement this, a running sum, s is defined that increments by k for every timer cycle. At the end of a full divided clock period consisting of n timer cycles, the sum will be equal to kn – obviously an exact multiple of n. Instead of allowing s to keep incrementing in this fashion it is examined at the end of each timer cycle after being incremented by k. If the result is equal to or larger than n then an extra count is added during the next timer cycle. After the next cycle finishes (i.e. *after* the extra count has been clocked), s is decremented by n. In this fashion s counts modulo-n although $s \geq n$ during the cycle in which a count is being added.

It would also work to decrement s by n immediately when the need for an extra count is detected. However, doing it as described above makes tracking phase a bit easier as described below.

The fractional-N division process described above allows the synthesis of frequency division ratios of arbitrary precision. That is, if we are willing to live with arbitrarily large values of n. In summary, this is how hardware-based fractional-N (a.k.a. "Frac-N") frequency dividers work.

B.2 The Challenge

Taking a step back to look at the big picture, the inaccuracy of the Mega's 16MHz system clock is the only reason that fractional-N and PLL sub-systems are necessary. In the project version which uses the GPS 10MHz clock as the Arduino's system clock, none of this is required because the 10MHz clock is perfectly synchronized with the PPS signal.

In review of the problem, consider that a UDP packet will arrive at the W5500 NIC chip on port 123 containing an NTP request. Arrival of that packet will cause the W5500 to assert its interrupt output, which will trigger a timer 5 input capture at some arbitrary point in between two PPS signals and at some arbitrary point within the frac-N cycle of 2048 timer cycles. From the captured timer count, the exact time elapsed since the last PPS signal must be determined, ideally ±250ns (i.e. within $\frac{1}{2}$timer cycle). This was the most difficult task in the development of the software.

B.3 Nitty Gritty Details

This section contains a lot of intricate math; it is documented here because the author found it necessary to get the firmware working correctly. Methods for accurately dividing the inaccurate 2MHz clock down to 1Hz±4ppb have been discussed above. Given a timer count value anywhere within the frac-N cycle, we need to compute the exact phase (aka time) of that count relative to the PPS signal.

The frac-N sub-system produces a 32Hz output clock which has an *average* frequency settable in 8ppb increments. The time period between individual 32Hz output clocks does not have this high resolution – only the average period over a span of 64 seconds exhibits this behavior. Keeping this in mind, the *average* clock period of the divided clock is

$$T_d = \frac{1}{f_d} = \frac{m + k/n}{f_c} = T_c(m + k/n)$$

where,

T_d is the average period of the 32Hz frac-N output clock.

f_d is the average frequency of output clock.

T_c is the true period of the (inaccurate) 2MHz timer clock.

f_c is the true frequency of the 2MHz timer clock.

m, n, k are the frac-N divider parameters previously defined.

A full frac-N cycle contains n cycles of length T_d so is of length (nT_d). Up till now, f_c has been considered a known quantity. In this application however, the values of m, k, n have been adjusted (by the PLL) to get a known average value for f_d or T_d and f_c is unknown. Specifically, the average of 32 frac-N output clock periods is adjusted to equal one second as defined by the PPS signal.

B.3.1 Brute Force Solution

The obvious solution is to just multiply the known length of a frac-N cycle (nT_d) by the timer count relative to the start of the entire frac-N cycle (c), then divide by the total length of a timer cycle $(mn + k)$:

$$t(c) = nT_d \, \frac{c}{mn + k}$$

To make this calculation with sufficient accuracy for this project requires 64-bit math – including a 64-bit integer division.

B.3.2 32-bit Solution

A different way to approach this problem (which can be implemented in 32-bit integer math) is by tracking timer cycles and added timer counts. Below, the word *nominal* will be used to refer to times based on the reference time period, T_d. For example, since the integer portion of the frac-N divisor is m, a *nominal* undivided clock cycle is T_d/m.

The true period of the inaccurate 2MHz clock, T_c can be expressed in terms of the average period of the 32Hz divided clock T_d:

$$T_c = \frac{T_d}{m + k/n}$$

Consider the true time elapsed for a timer cycle of m counts (i.e. there is no added count in the cycle). At any count (c) from 0 to m within this cycle the time expired is:

$$t(c) = c\, T_c = \frac{c\, T_d}{m + k/n}$$

Below, it will be convenient to have this expressed in terms of one nominal undivided clock cycle, T_d/m:

$$t(c) = c\,\frac{T_d}{m}\,\frac{m}{m+k/n} = c\,\frac{T_d}{m}\left(1 - \frac{k/n}{m+k/n}\right)$$

$$t(c) = c\,\frac{T_d}{m} - \frac{T_d}{m}\,\frac{ck/n}{m+k/n} = \frac{T_d}{m} + \delta(c) \tag{B.1}$$

The second term above (δ) can be viewed as a correction to the nominal undivided clock period.

$$\delta(c) = -\frac{T_d}{m+k/n}\,\frac{ck}{nm} = -T_c\,\frac{ck/m}{n}$$

At the end of the first timer cycle ($c = m$), the correction is:

$$\delta(m) = -\frac{T_d}{m+k/n}\,\frac{k}{n} = -T_c\,\frac{k}{n}$$

At the end of several (e.g. α) timer cycles, assuming no extra ticks have been added, the difference is simply α times larger:

$$\delta(\alpha m) = -T_c\,\frac{\alpha k}{n}$$

Recall that in running the fractional-N process a running sum s is kept which increments by k for every timer cycle. Until the first added clock cycle this running sum is equal to αk in the above formula, so:

$$\delta(\alpha m) = -T_c\,\frac{s}{n}$$

Next consider the addition of an extra timer count at the end of the α'th cycle. The correction δ is simply increased by T_c.

$$\delta(\alpha m + 1) = -T_c\,\frac{s}{n} + T_c = -T_c\,\frac{s-n}{n}$$

Thus, adding an extra count can be handled by subtracting n from s. This is the sought after result. To summarize the process of computing phase, return to the full formula, not just the correction we've been working with.

Any timer count within the 2048-cycle frac-N period can be defined by breaking it up into full timer cycles (α) plus a count within the current cycle (c). Referring to (B.1) above, the true time at any such point can be shown to be:

$$t(\alpha, c) = \alpha T_d - T_c\,\frac{s + kc/m}{n} \tag{B.2}$$

Here, α is the number of full timer cycles elapsed since zero phase was passed (we assume that zero phase is coincident with the start of a timer cycle). c is the

number of counts elapsed within the current incomplete timer cycle. During the first cycle after zero phase, $\alpha = 0$ and c runs from zero to m. In a cycle with an added count, c will actually go to $m + 1$ at the end. s is the running sum, incremented by k for every full timer cycle and decremented by n *after* any timer cycle which included an extra count.

The numerator of the fraction in the above equation:

$$s + k\,\frac{c}{m}$$

includes what is essentially a pro-rated addition of k to the running sum s – since at the end of the cycle k would be added to s, a pro-rated amount needs to be added if we are only part-way through the cycle.

B.3.3 Artifacts

Analysis of PLL performance shows certain very obvious artifacts which are a result of the fractional-N divider's design. They are mathematical artifacts in the sense that they do not necessarily represent true phase tracking behaviors but rather phase measurement errors.

The PLL sub-system is based around an interval of 32 timer cycles (1 second). The frac-N sub-system is based around an interval of 2048 timer cycles (64 seconds). This difference creates some artifacts in the PLL's behavior.

During one frac-N interval there are $mn+k$ cycles of the inaccurate 2MHz timer clock. First consider a case where $mn + k$ is by coincidence a prime number. If the frac-N divisor is adjusted perfectly, then there will be exactly $mn + k$ timer clocks during a 64-second interval as defined by the PPS signal. This will be exact. However, since $mn + k$ is prime and covers an interval of 64 seconds, there will no intermediate 2MHz clock cycles which align exactly with any of the 63 PPS signals in between. Every 64th PPS signal will be aligned exactly with a 2MHz clock edge but the other 63 will not.

Now consider a second situation where $mn + k$ just happens to be an integer multiple of 64; in this case all 64 PPS signals within a frac-N cycle will be aligned exactly with a 2MHz timer clock edge.

When analyzing the recorded phase (e.g. time) of PPS signals this behavior becomes strikingly apparent. Depending on the exact value of $mn + k$ a plot of measured phases is seen to jump around over a range of $\pm$250ns ($\frac{1}{2}$clock tick) in various patterns.

In one sense, this could be considered a form of under-sampling of the frac-N system and it results in some funny looking analysis data which might be considered aliasing in some sense of the word. On the other hand, it does in fact allow the PLL to control the frac-N phase to a tighter tolerance. In the end, we put up with the artifacts of oversampling because the overall result is a more accurate system.

These odd behaviors are a purely mathematical artifact of the system design and do not represent any real jitter or wander of the crystal oscillator. As such, it is desirable to filter them out so the PLL does not attempt to follow them. This is the reason for a 16-tap FIR low-pass filter as part of the PLL's loop filter block.

When $mn + k$ is close to a multiple of 64, these artifacts can take on a very low-frequency behavior and in this situation it is not possible to filter them out in the loop filter, and the PLL will attempt to track them. This will result in additional phase wander, typically of as much as ±250ns. This seems to be unavoidable and it would require a change in the overall system design to improve it. The project version which uses the GPS 10MHz clock for Arduino's system clock does not have this behavior.

Experiments with smaller values for n in the frac-N divider have been tried. From one perspective it would seem that $n = 32$ would be a good choice. This would make the frac-N cycle be exactly one second and would completely eliminate the artifacts. In practice however, although the artifacts do vanish, the loop is unable to maintain phase ±500ns in this case. It seems better to live with the artifacts.

Appendix C

Firmware

The sketch file itself (`StratumTen.ino`) is relatively small and most of the functionality is implemented in several C++ classes. The standard library for the version 2 Ethernet shield had to be modified slightly to enable interrupts on the W5500 NIC chip.

The name, *StratumTen* is a reference to the use of the 10MHz GPS-disciplined clock in the advanced variant of the project.

C.1 `NtpConfig.h`

This is the one mandatory file that must be checked for correct options. There are many comments in this file explaining how to set options, and a comment line indicating the end of normal user options; don't make changes below this line unless you know what you are doing.

- Be sure to set the MAC address to match the value that was delivered with the Ethernet shield.
- Set a hard IP address for the server.
- Choose the correct option for project variant (basic/advanced) and specify which GPS receiver is being used.

C.2 C++ Classes

Short descriptions of some (but not all) of the classes are given below.

C.2.1 `Timekeeping`

This virtual class provides for the management of the Mega2560's timer 4/5 hardware and provides client with exact NTP time associated with events such as UDP packet reception.

C.2.2 Fractional-N Classes

The `FractionalN` class implements a standard fractional-N frequency divider in software, using the Mega's timer 4 (T4) in compare-match mode.

The `NtpFractionalN` class derives from the above two classes. It provides the interpolation to accurate NTP time from timer values based on the equations developed in appendix B.

C.2.3 `NtpPhaseLockedLoop`

This class processes PPS time capture events on timer 4 (T4) and adjusts the frac-N divisor for every PPS capture to keep the frac-N output locked to the PPS signal. An in-depth description of this is presented in appendix A.

The class requires a reference to a frac-N instance for which it manages the divisor. It also uses the frac-N instance to obtain phase information at PPS captures as part of the operation of the feedback loop. Feed-forward correction is also implemented in this class.

C.2.4 `DisciplinedTimers`

Also derived from the `Timekeeping` class, the advanced variant uses this class instead of fractional-N and phase-locked loop classes to manage timers.

C.2.5 `NtpServer`

This class provides functions to process and reply to NTP client requests. Its constructor requires two class references, to a `Timekeeping` instance from which to obtain timestamps and to a UDP instance so it can send the NTP reply. This class can reply to client/server requests as well as interleaved symmetric mode packets.

While it also has the ability to respond to non-interleaved symmetric mode packets, that ability is disabled at this time. There is no performance advantage over client/server mode so it was not enabled.

C.2.6 Filter Classes

Several classes are defined which implement discrete time digital filters, both FIR and IIR variants. These are used in by the PLL class in filtering feed-forward corrections and in the main loop filter block.

C.2.7 `HP55300A`

This class provides communications with the HP55300A through an RS232 interface to its Time of Day port. The single purpose of this class is to obtain current calendar time and associate it with a specific PPS pulse. This allows the frac-N class instance to know the exact NTP time.

C.2.8 `IcmSmt360`

Provides communications with and control of the Trimble GPS module. It can obtain time-of-day, check receiver health and status and initializes desired options in the receiver.

C.2.9 `NtpTime`

This utility class provides functions to determine NTP time from UTC calendar dates and times.

C.2.10 Arduino library changes

The Ethernet2 library has been slightly modified to enable interrupts on the WizNet W5500 NIC chip. The standard library does not use interrupts from the W5500 so this change causes no conflicts. Whenever a UDP socket is opened, the RECV event is enabled to cause an interrupt from the W5500. This will then occur when a NTP client request is received. The interrupt is wired to the input capture pin for timer 5 and this provides a hardware timestamp on incoming NTP requests.

The firmware package includes a library named EthernetNtp which contains these changes.

This design might also work with the original Ethernet shield using the WizNet W5100 NIC chip but that has not been tested. It is an exercise left for the reader.

C.3 Fudge

There is what this author finds to be a somewhat annoying term in the world of NTP, which goes by the name *fudge*. This word is often used to name variables in

software with no additional explanation. It is annoying because to the uninitiated it conveys only a vague and ambiguous meaning. Other than to complain about it here, the term is not used in this document, nor in the software.

What the term refers to (as far as this author can discern) are small time offsets used to fix errors in generated timestamps. Sometimes it compensates for controlled software delays and other times it is a hand-tuned offset covering a multitude of sins.

C.4 Hidden Interrupts

Under the hood, Arduino is generating interrupts all the time. If left un-managed, this would interfere with accurate timing and cause semi-random jitter in the NTP server's transmit timestamps.

- Timer0 is used internally by the millis() and micros() functions. It runs at one-64$^{\text{th}}$ of the system clock frequency regardless of the value of `F_CPU`. and generates several hundred interrupts per second. This can easily interfere with the generation of accurate transmit timestamps. A quick count of instruction cycles in the overflow ISR for Timer0 gives roughly 90 cycles or 9μs spent in the ISR on average (with a 10MHz clock).

- Serial ports use interrupts for both sending and receiving data. These interrupts can also interfere so it is important to ensure that no serial I/O is active while processing an NTP request. This sketch uses Serial0 for USB communications, and Serial1 for GPS communications. With the exception of USB data input from a host computer it is possible to ensure that no serial I/O is active during NTP request processing.

C.4.1 Solution

There are two parts to managing interrupts to avoid errors in transmit timestamps.

1. Keep track of when serial I/O is active and do not respond to NTP requests during those times. This presumes that there is no incoming I/O from the USB port. Other than that, the sketch decides when to initiate serial output to the USB port and that is easily managed.

   Once everything is up and running, there will infrequent I/O to the GPS receiver so this does not cause very many NTP requests to be skipped.

2. Stop Timer0 during the time-critical section while responding to an NTP request. This will work as long as the Ethernet library does not use Timer0, and that does seem to be the case. This will result in the loss of roughly 1ms

of time for every NTP request received. The sketch is not using these timers for anything critical so the lost time is of little concern.

(a) If less impact on millis() and micros() is desired, then after timestamp generation is complete and the transmission kicked off, manually add the approximate amount of missed time to the timer value and associated overflow variables, then re-start the timer. See `wiring.c` for the relevant code and variables.

Appendix D

NTP Timestamp Calculations

For the purpose of debugging, these calculations provide useful insight. The question is, what happens if an error is made striking the transmit timestamp?

$$\begin{aligned} t_2 &= t_1 + \delta_1 + \epsilon \\ t_3 &= t_2 + \rho = t_1 + \delta_1 + \epsilon + \rho \\ t_4 &= t_3 - \epsilon + \delta_2 = t_1 + \delta_1 + \rho + \delta_2 \end{aligned}$$

The NTP offset formula is:

$$o = \frac{t_2 + t_3 - t_1 - t_4}{2} \tag{D.1}$$

$$= \frac{2t_1 + 2\delta_1 + 2\epsilon + \rho - -2t_1 - \delta_1 - \rho - \delta_2}{2} \tag{D.2}$$

$$= \frac{\delta_1 - \delta_2 + 2\epsilon}{2} \tag{D.3}$$

$$o = \epsilon + \frac{\delta_1 - \delta_2}{2} \tag{D.4}$$

The NTP delay formula is:

$$d = t_2 + t_4 - t_1 - t_3 \tag{D.5}$$

$$= 2t_1 + 2\delta_1 + \epsilon + \rho + \delta_2 - 2t_1 - \delta_1 - \epsilon - \rho \tag{D.6}$$

$$= \delta_1 + \delta_2 \tag{D.7}$$

This can be extended to consider the effect of an error in striking the transmit timestamp. Simply let

$$t_3 = t_1 + \delta_1 + \epsilon + \rho + \epsilon_t \tag{D.8}$$

This leads to:

$$o = \epsilon + \frac{\delta_1 - \delta_2}{2} + \frac{\epsilon_t}{2} \tag{D.9}$$

$$\tag{D.10}$$

$$d = \delta_1 + \delta_2 - \epsilon_t \tag{D.11}$$

The net result is that an positive error in striking the transmit timestamp shows up as an equal decrease in measured delay and half as much of an increase in mesured offset.

Appendix E

NTP Full Packet Structure

Here is an accounting of the entire on-wire packet structure of an NTP client/server or peer mode packet. This includes all of the enclosing bytes that will be added at lower levels in the protocol stack. It is valid for 100 base-TX Ethernet.

- Preamble, 7 bytes
- SOF, 1 byte
- Source/destination MACs, 12 bytes
- Length, 2 bytes
- Payload
  - IPv4 header, 20 bytes minimum
  - UDP header, 8 bytes
  - UDP payload, 48 bytes (the NTP packet)
- FCS (checksum), 4 bytes

Ethernet frame overhead is 26 bytes, plus another 28 bytes for IPv4 and UDP headers for a total overhead of 54 bytes. Thus a 48 byte UDP payload requires a total of at least 102 bytes (816 bits) to be transmitted at 100Mb/s over the wire. Total transmit time is 8.16μs.

Packets with authentication are closer to 1000 bits but this project does not use or support them.

Appendix F

Handy-Dandy Acronym Decoder Ring

GPS Really? You are supposed to know this one already. Try the Internet.

IDE Integrated Development Environment. A friendly computer program and environment which makes it easier to create and test computer programs.

NIC Network Interface Controller. This is the integrated circuit which is directly connected to the Ethernet cable and provides the hardware level electrical interface.

NTP Network Time Protocol.

PCB Printed Circuit Board.

PPS Pulse-Per-Second. This is an electrical output from a GPS receiver which provides an electrical pulse exactly once every second, on the second.

TCXO Temperature Compensated Xtal Oscillator. *Xtal* is a funny way to spell the word crystal, used mostly by engineers.

UDP User Datagram Protocol. A low-level, low overhead data transfer technique used to send data over Ethernet.