Let's build…
An NTP Server

By Bruce Hall, W8BH

PART 2: NTP

This device is a GPS-disciplined clock with microsecond resolution. It provides stunningly accurate time information to any device on the local wireless network.

Part 1 introduced hardware and software for GPS timekeeping. Part 2 explains how time is distributed on a computer network.

STEP 6: UDP LISTENER

Time information on the Internet is formatted according to the NTP protocol and sent/received over UDP "User Datagram Protocol". Let's write a small bit of code that listens for UDP. Fortunately, there is an ESP8266 library for UDP that shields us from all the messy details. So, the first thing our sketch contains is this library. We need to throw in the WiFi library too, to interface with the radio hardware:

```
#include <ESP8266WiFi.h>                          // WiFI support
#include <WiFiUdp.h>                              // UDP packet support

#define WIFI_SSID "networkID"                     // replace with your ID
#define WIFI_PASS "password"                      // replace with your password
#define UDP_PORT   1234
```

The three defines specify the name and password for our local WiFi connection, and the port on which we will listen for UDP traffic. Think of the UDP port as an address. For this example, we arbitrarily chose 1234, but certain services use specific ports. NTP, for example, uses port 123.

There is excellent online documentation for the UDP library here: UDP — ESP8266 Arduino Core

The setup() code calls UDP.begin() to start listening for UDP messages:

```
void setup() {
  Serial.begin(115200);
  connectToWiFi();
  UDP.begin(UDP_PORT);
  Serial.print("Listening on UDP port ");
  Serial.println(UDP_PORT);
}
```

And the loop call UDP.parsePacket() to detect incoming messages:

```
void loop() {
  int packetSize = UDP.parsePacket();         // look for packet data
  if (packetSize) {                           // data was received, so...
    printPacket();                            // print it out
    acknowledgePacket();                      // and send a reply
  }
}
```

Printing a packet calls UDP.read(), specifying a character buffer to store the data. Since it is just a string of characters, it can be printed with Serial.print():

```
void printPacket() {
  int len = UDP.read(packet, 255);          // read the data
  packet[len] = '\0';                       // terminate the incoming string
  Serial.print("Packet contents: ");
  Serial.println(packet);                   // print the packet contents
}
```

Constructing a reply requires 3 calls: beginPacket(), write(), and endPacket(). When endPacket() is called, the reply message is sent back to the remote device:

```
void acknowledgePacket() {
  UDP.beginPacket(UDP.remoteIP(),UDP.remotePort());
  UDP.write("Roger that!");
  UDP.endPacket();
}
```

That's the entire sketch. To test it,

1. Download Step6.ino from GitHub.
2. Replace the SSID and password entries with your own network credentials.
3. Open the Serial Monitor at baud rate 115200.
4. Compile and upload the code to your board.

The serial monitor will report when the MCU connects to your network. After connecting to your network, it will listen for, print, and reply to all UDP messages sent to port 1234. But how do we create UDP messages to test it? You can either program a second ESP8266 to send messages, or… you can use a handy utility called "Packet Sender". Type your message, specify the IP address and port, and click 'Send'.

STEP 7: SENDING TIME MESSAGES

Did you try Step 6 with Packet Sender, creating UPD messages and getting replies? If not, go back and try it. After you do, imagine that we type the UDP message "Time". And imagine that the answer returned by our sketch IS the time. We don't have to imagine it, because we have everything we need: Step 6 gave us the ability to send/receive messages and Step 5 gave us the time message to send. All we need to do is merge the two sketches.

Download and review Step7.ino. Here is the new acknowledgement routine:

```
void acknowledgePacket() {
  int h = gps.time.hour();                  // get the hour
```

```
    int m = gps.time.minute();                    // get the minutes
    int s = gps.time.second();                    // get the seconds
    sprintf(reply,"%02d:%02d:%02d UTC",h,m,s);    // copy time to reply buffer
    Serial.print("Packet sent:     ");
    Serial.println(reply);                        // print the packet sent

    UDP.beginPacket(UDP.remoteIP(),UDP.remotePort());
    UDP.write(reply);
    UDP.endPacket();
  }
```

The only tricky bit is stuffing a character buffer with formatted time, highlighted above. Otherwise, this sketch is exactly the same as Step 5 and 6 combined. Run it and you'll see the same clock display as before. But now, if you send it a UDP message to port 1234, it will reply with the current time. Step 7 is a complete time server – *in less than 100 lines of code!*

NETWORK TIME PROTOCOL (NTP) INTRODUCTION

We are nearly there. The last hurdle is formatting our time message according to the NTP specification. If our message is NTP formatted, it can be read and understood by nearly all network time-requesters. Conversely, if our message is not NTP formatted, it will be ignored by all time-requesters. We must use NTP. The plan forward is to take the Step7.ino sketch, put the time data into this NTP data structure, and call it Step8. But a lot of explanation is needed first.

Warning: a lot of technical details follow. Feel free to skip to "Step 8" to preserve your sanity.

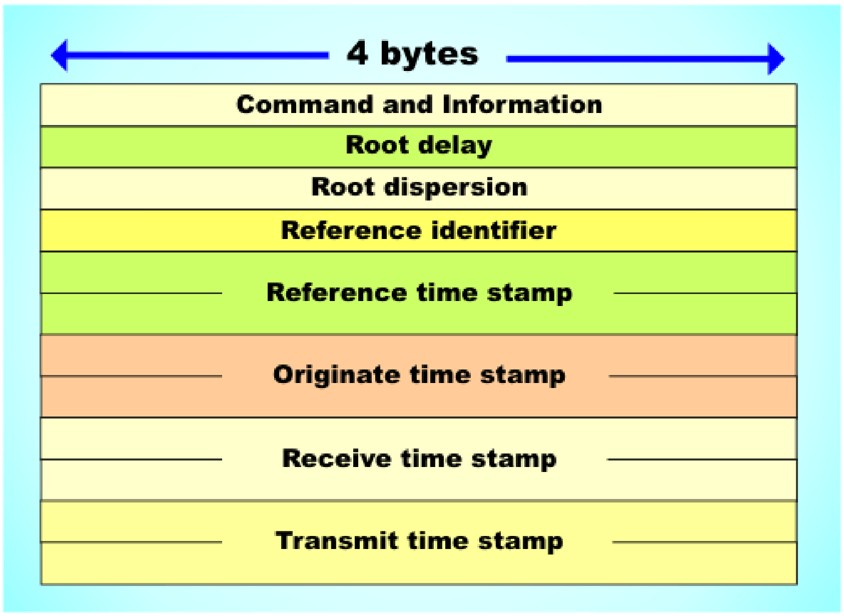
GORY DETAILS I – NTP DATA STRUCTURE

The basic structure of an NTP message is a 48-byte UDP packet.

The packet is often represented by 12 "chucks" of data, each 4 bytes in width.

An NTP packet contains 4 different timestamps, each 8 bytes in length. NTP clients use these timestamps to help determine delays introduced by the network.

Other NTP fields, such as precision, root delay, and root dispersion, provide information on the quality of the data being provided. In other words, they indicate the amount of error the timestamps may contain.



NTP has been around a long, long time. Some consider it the oldest, active internet protocol in use today. It is currently in version 4. Here are links to the NTP reference documents:

- Version 1: RFC 1059
- Version 2: RFC 1119
- Version 3: RFC 1305
- Version 4: RFC 5905

NTP timestamps are 8-bytes in length and have their own structure. The first four bytes represent the number of seconds since 1/1/1900. The last four bytes represent the fraction of a second, in units of 2^-32 second (roughly 0.2 nanoseconds):

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Integer Part                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Fraction Part                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The structure of an NTP packet can therefore be expressed in code like this:

```
struct ntpTime_t {                          // NTP timestamp is 64-bits:
  uint32_t seconds;                         // # seconds since 1/1/1900
  uint32_t fraction;                        // 0x00000001 = 2^-32 second
};

struct ntpPacket {                          // full packet is 48 bytes in length:
  uint8_t    control;                       // leap, version, and mode
  uint8_t    stratum;                       // 0=undef, 1=primary, 2-15 secondary
  uint8_t    polling;                       // polling frequency, in 2^x seconds
  int8_t     precision;                     // precision, in fraction of second
  int32_t    rootDelay;                     // round-trip delay to/from reference
  uint32_t   rootDispersion;                // maximum accumulated error
  char       refID[4];                      // reference ID
  ntpTime_t  referenceTime;                 // time of last ref clock update
  ntpTime_t  originateTime;                 // time of client transmission
  ntpTime_t  receiveTime;                   // time server received NTP request
  ntpTime_t  transmitTime;                  // time server/client sent message
}
```

Each component of the NTP data structure is discussed below.

CONTROL: The first byte in the NTP packet is the "Control" byte. The format of its 8 bits is LLVVVMMM, where LL is leap second indicator, VVV is the NTP version number, and MMM is the mode. Here are a few examples:

- 0x1B = 0001.1011 = 00 011 011 = leap 0, ver 3, mode 3.
- 0x13 = 0001.0011 = 00 010 011 = leap 0, ver 2, mode 3
- 0x0B = 0000.1011 = 00 001 011 = leap 0, ver 1, mode 3.

For the NTP request to be valid, the version number must be 1 through 4. NTP has several different modes, but for our purposes all requests must be from clients (mode 3) and all replies are from a server (mode 4). A valid request will also have a leap second value of 0 (none) or 3 (unknown). When replying to an NTP request, the version information should remain the same, but the mode must change from client to server. One method is to mask the control byte with the 0x38, which will keep the version number intact but zero out the leap code and mode. Adding 4 to the masked value will set the mode to mode 4 (server).

STRATUM: the level of each server in the hierarchy is defined by a stratum number. Primary servers are assigned stratum one; secondary servers at each lower level are assigned stratum numbers one greater than the preceding level. As the stratum number increases, its accuracy degrades depending on the network path and system clock stability. Stratum numbers 1 through 16 are defined, where stratum 1 represents a server attached to a reference clock like GPS. Stratum 0 is undefined. However, a server may send "Kiss-o'-Death" (KoD) messages to clients using a Stratum 0 packet.

I find the next four fields (polling, precision, root delay, root dispersion) very confusing. I have seen them interpreted in different ways.

POLLING: an exponent (2^x) that describes the maximum interval between successive messages. Valid values range from 4 (2^4 = 16 seconds) to 17 (2^17 seconds = 36 hours). Suggested minimum and maximum defaults are 6 and 10, respectively. As far as I can tell, the POLLING number is not applicable to messages sent by a server.

PRECISION: 8-bit signed integer representing the precision of the system clock, in log2 seconds. For instance, a value of -20 corresponds to a precision of about one microsecond. RFC 5909 states that precision can be determined when the service first starts up as the minimum time of several iterations to read the system clock. I confess that I find that definition confusing. A uBlox NEO-6M GPS generates its 1ppm signal within 30 nS of the actual start of the second, on average, and within 60 nS 99% of the time. However, these results are reported by our server with a resolution of 1 microsecond. Since GPS timing accuracy is so much finer than our ability to report it, precision of our "system" is constrained by its 1 microsecond resolution. I am therefore using a precision to -20. Let me know if you think it should be something else!

ROOT DELAY: is the total round-trip delay to the reference clock, in units of 2^-16 seconds. The reference clock in our case is hardwired via external interrupt, so there is no "round-trip". The time from electrical impulse to recording time is on the order of 10 uS. However, the time is not available until the interrupt routine has completed, making the effective interval from GPS hardware to server time about 60 uS.

ROOT DISPERSION: is the maximum error relative to the reference clock, in units of 2^-16 seconds. For most systems, dispersion is highly dependent on the system clock. As a ballpark figure, the ESP8266 oscillator has a frequency drift of 25ppm. Which is to say that one measured second by this clock could have an error +/- 25 microseconds. If 8 seconds have passed since our last GPS synchronization, the time error could be as large as 8*25 = 200 uS.

REFERENCE ID: for secondary servers, this usually corresponds to their IP address. But for primary (Stratum 1) servers, this field indicates the type of reference clock. The value for our NTP server is "GPS", since the GPS service is our time reference.

The last four fields in the packet are all 64-bit timestamps, and defined as follows:

REFERENCE TIMESTAMP: the time when the system clock was last set or corrected.
ORIGIN TIMESTAMP (t1): the time at the client when the request departed for the server.
RECEIVE TIMESTAMP (t2): the time at the server when the request arrived from the client.
TRANSMIT TIMESTAMP (t3): the time at the server when the response left for the client.

In addition, the client also records a DESTINATION TIMESTAMP (t4), not contained in the message, which is when the server response is received by the client.

Why so many timestamps? These timestamps are used by the client to determine and adjust for delays introduced by the network. It is useful to know how a client uses the timestamp data. Consider the following client-server example:

Client NTP request	
…	
ORIGINATE TS	12:00:00
RECEIVE TS	(blank)
TRANSMIT TS	12:00:00

1. Client sends a request packet at time t1, recording the time of its request as t1 (ORIGINATE). In this example, the client's clock is running slow.

Server NTP response	
…	
ORIGINATE TS	12:00:00
RECEIVE TS	12:05:03
TRANSMIT TS	12:05:04

2. At time t2, when the packet arrives at the server, the server records t2 (RECEIVE), which is the time that the request was received.

3. The server processes the request. In this example, processing takes one second. The server then sends a response back to the client at time t3 (TRANSMIT).

4. The client receives the response at time t4 (DESTINATION), which for this example is 12:05:07. Destination time is not recorded in the NTP packet. To summarize, t1 = 12:00:00, t2 = 12:05:03, t3 = 12:05:04, and t4 = 12:00:07.

5. The client computes:
   a. Offset (theta) = Time at Server – Time at Client = ½ * [(t2-t1) + (t3-t4)]
   b. Round trip delay (delta) = Time from Client to server to Client = (t4-t1)-(t3-t2).

Using the above data and formulas,

a) Time offset between server and client = ½ * (05:03 + 04:57) = ½ * (10:00) = +5 minutes.
b) Round-trip network delay = (00:07) – (00:01) = 6 seconds.

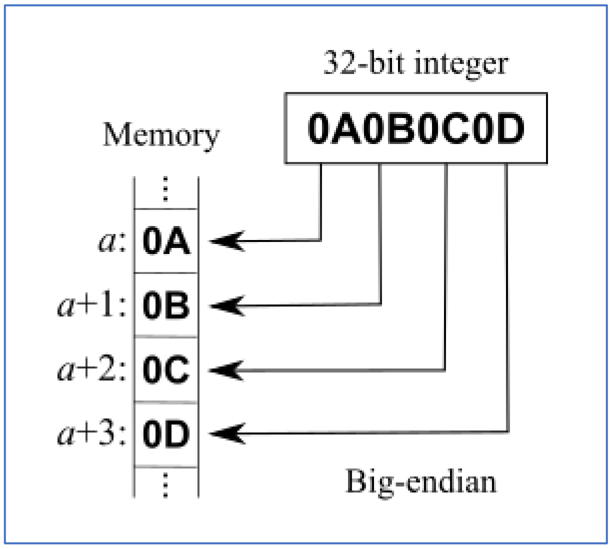
Finally, the client adjusts its own time (adds 5 minutes), using an offset that is *independent of network speed.*

GORY DETAILS II – NTP TIMESTAMPS

Time on Arduinos is in the Unix format: the current time is the number of seconds since 1/1/1970. However, NTP time represents the number of seconds since 1/1/1900. The difference between the two timestamps is exactly 70 years, which equates to over 2.2 billion seconds:

NTP time = Unix Time + 2208988800 seconds

Another annoying complication is "endianness". On the ESP8266, integers are stored in memory with the least-significant byte first. On your local area network (and the Internet), however, those same integers travel with the most-significant byte first (see graphic). For data transmitted over the internet, we must reverse the byte order before it can be used by the microcontroller. Conversely, any data to be transmitted over the network must be converted from little-endian to big-endian format.



Fortunately, there are dedicated functions for this purpose. The function **ntohl** (read: "net to host long-integer") converts 4-byte integers from network (big-endian) format to host (little-endian) format. The function **htonl**() does the opposite conversion.

The following lines of code take into account endianness, as well as the 70-year time difference:

netTime = **htonl** (hostTime + 2208988800L);
hostTime = **ntohl**(netTime) - 2208988800L;

Once the timestamp data is correctly formatted, it can be put into a NTP timestamp structure. Recall that an NTP timestamp has two components, a 4-byte value for seconds, and a 4-type value for the fraction of a second:

```
struct ntpTime_t {                          // NTP timestamp is 64-bits:
  uint32_t seconds;                         // # seconds since 1/1/1900
  uint32_t fraction;                        // 0x00000001 = 2^-32 second
};
```

Let's assume that our server produces time to the nearest second. Then the fractional part is zero. Our code to fill a NTP timestamp now looks like this:

```
time.seconds = htonl(t+2208988800L);        // convert host time to net time
time.fraction = 0;                          // no fractional seconds
```

Putting it all together, we can create a function that takes an Arduino/unix timestamp and creates the corresponding NTP timestamp:

```
ntpTime_t unixToNTPTime(time_t t) {
  ntpTime_t response;
  response.seconds = htonl(t+2208988800L);
  response.fraction = 0);
  return response;
}
```

STEP 8: SERVING UP NTP TIME

The preceding paragraphs stumped me for a long time. Kudos to you if you got this far. The NTP server creates NTP messages that contain time information. Here is how we prepare a response message:

```
void prepareResponse() {
  ntp.control        = (ntp.control&0x38)+4;
  ntp.stratum        = 0x01;
  ntp.precision      = 0x00;
  strncpy(ntp.refID,"GPS",4);
  ntp.originateTime  = ntp.transmitTime;
  ntp.referenceTime  = unixToNTPTime(now());
  ntp.receiveTime    = ntp.referenceTime;
  ntp.transmitTime   = ntp.referenceTime;
}
```

The first few fields are filled out as above (see NTP data structure). The highlighted line shows the current time, returned by now(), formatted as an NTP timestamp. This value is used to fill referenceTime, receiveTime, and transmitTIme.

Setting originateTime is a bit tricky. originateTime is the time when the requesting device sent its request – which is the value of ntp.transmitTime in the incoming request packet. The first line of the following code takes data from the request packet and copies into the data structure:

```
void answerQuery() {
   memcpy(&ntp,&packet,sizeof(ntp));
   prepareResponse();
   memcpy(&packet,&ntp,sizeof(ntp));
   UDP.beginPacket(UDP.remoteIP(), UDP.remotePort());
   UDP.write(packet,sizeof(ntp));
   UDP.endPacket();
}
```

After the response is prepared, it then copies the data structure back into a packet for transmitting. The final three lines send the UDP data packet out over the network.

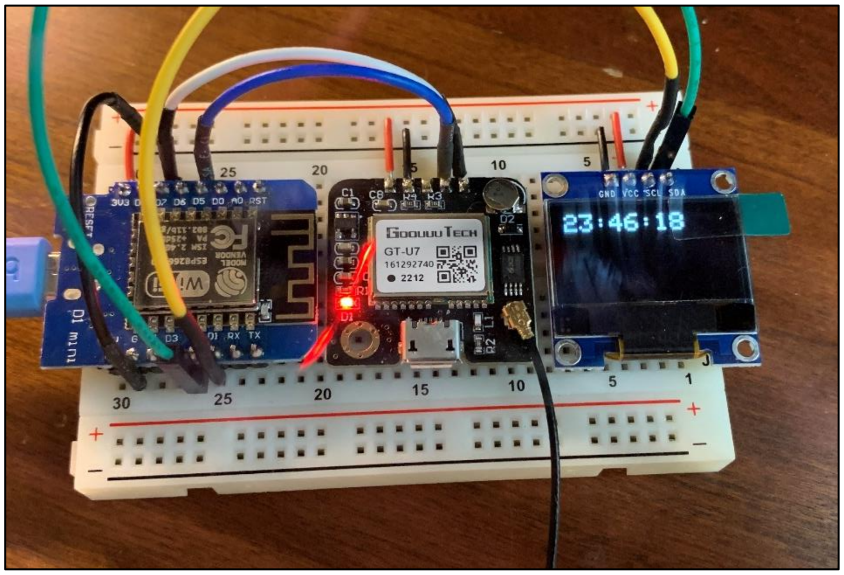
That covers Step8.ino, an NTP time server with one-second resolution. Your breadboarded project is now capable of receiving NTP requests and serving up time on your local area network. Try it! Run the sketch and wait until the time and IP information appears on the OLED display. Then, on a Windows11 PC, open the control panel, then select “Clock and Region” > “Set the time and date” > “Internet Time” > “Change settings”. For the server, enter the IP listed on your OLED display, such as 192.168.86.21, and click on the “Update Now” button. If successful, Windows will report the time of synchronization.

BUT WAIT, THERES MORE

Arduino time functions make it easy to manipulate time that is measured in seconds. For most timekeeping, seconds are good enough. However, GPS provides time accuracy to the microsecond. Additional coding makes it possible to add microsecond data, making this server a very accurate and precise time source.

The final sketch, ntp_server.ino, adds the following features:

- Microsecond time resolution.
- RTC fallback when GPS is unavailable.
- Customizable ASCII time output via serial port.
- Display of GPS lock status and GPS satellite count.
- Display of Wi-Fi signal strength and packets sent/received.
- Wi-Fi credential setting via mobile phone.



Best of all, it still runs on the breadboarded server from Part 1.

Breadboarding is great for experimentation, but a more permanent solution is needed to make this a usable device. Part 3 of this series describes how to build your own NTP time server.

73, Bruce.

Last updated: July 2, 2023