Embedding it better...

µTasker Document

µTasker – SNTP (Simple Network Time Protocol)

Table of Contents

1. Introduction 3
2. Activating SNTP and Demo in the µTasker Project 4
3. Calculating the Synchronised Time from Timestamps 6
4. Kiss-o'-Death Packet 7
5. Poll Interval 7
6. Example of Synchronised Time Calculation 7
7. Local Time 10
8. Conclusion 11

1. Introduction

This document describes the Simple Network Time Protocol together with its implementation in the µTasker project and an example of its practical use.

SNTPv4 is described in RFC 4330 = Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI. It is a sub-set of the more accurate Network Time protocol (NTP RFC 1305) which is used to synchronize computer clocks in the Internet. Whereas NTP achieves 1..50ms accuracy in the Internet, SNTPV4 can achieve several hundred ms accuracy in the Internet. In a LAN the accuracy is much higher due to the absence of the larger transmission delays in the Internet.

SNTP is based on connectionless UDP frames and SNTP servers listen on port 123.

List of Stratum 1 NTP servers on the Internet:
http://support.ntp.org/bin/view/Servers/StratumOneTimeServers

List of Stratum 2 NTP servers on the Internet (generally used by clients for requesting time form):

http://support.ntp.org/bin/view/Servers/StratumTwoTimeServers

All times are in UTC (Universal Time Coordinated), which is represented by two long words; the first being the number of seconds since the reference date 1.1.1900 and the second a fraction of a second.

2. Activating SNTP and Demo in the µTasker Project

By setting the define `USE_SNTP` in `config.h` the SNTP demo is activated. Since it is UDP based it requires that Ethernet and UDP are also enabled; `USE_TIME_SERVER` should however be disabled so that it doesn't interfere with the test.

The demo requests a new time synchronisation every 15s from a sequential list of SNTP servers (`SNTP_ServerList[]`). This allows testing frequent synchronisations without involving a single server to be requested too often (*a minimum of 15s request interval is stipulated but lower polling rates are advisable*).

When the test is started, a UDP socket is first bound to port 123 ready for the transmission of a request. The local time is set to an arbitrary value by calling

```
fnSynchroniseLocalTime((SNTP_TIME *)&default_time);
```

where

```
const SNTP_TIME default_time = {REFERENCE_TIME, 0};
```

REFERENCE_TIME is a known number of seconds at a chosen point in time in the past – presently 6th August 2009 at 0:0:0. This value is also used to calculate the number of seconds elapsed since this date, which is more practical than calculating from 1.1.1900.

The transmission is evoked by `fnRequestSNTP_time(0)`, which doesn't advance the time server, whereby subsequent synchronisations (every 15s) use `fnRequestSNTP_time(1)`, which advances through the list of time servers on each test.

Note that the first transmission may not be possible since the gateway address must first be resolved by the ARP layer. In this case the transmission is immediately repeated on reception of the ARP event `ARP_RESOLUTION_SUCCESS`.

Note also that a local time-stamp timer is synchronised to the default time when `fnSynchroniseLocalTime()` is called. This timer is described in chapter 7.

When a response is received from the time server it is analysed and a new local time synchronised to. If there is no answer, no synchronisation is performed and the next timer server will be requested 15s later.

Each time a synchronisation request is attempted the local time stamp is sent to the debug output, in its UTC format and the days, hours, minutes and seconds since the reference date.

If a response is received from the requested time server, the enclosed time stamps as well as the new synchronised time is printed to the debug output to allow monitoring the process.

A typical sequence of tests will look as follows – note that not all servers answer; some send an ICMP message back to inform that they have no listening port.

```
Time requested at 0xce60fabe:0x7e9e16f0 45:19:02:22:7e9e16f0

Server rx: 0xce60fabf:0x20057edc 45:19:02:23:20057edc
Server tx: 0xce60fabf:0x200e54a2 45:19:02:23:200e54a2
Received at: 0xce60fabf:0x050c7f10 45:19:02:23:050c7f10

New time: 0xce60fabf:0x63411dcf 45:19:02:23:63411dcf  (new synchronisation)

Time requested at 0xce60face:0x593e0950 45:19:02:38:593e0950

Server rx: 0xce60face:0x19faa2b4 45:19:02:38:19faa2b4
Server tx: 0xce60face:0x1a00b85d 45:19:02:38:1a00b85d
```

```
Received at: 0xce60face:0x5d983b90 45:19:02:38:5d983b90

New time: 0xce60face:0x1c2ac6a8 45:19:02:38:1c2ac6a8  (new synchronisation)

Time requested at 0xce60fadd:0x177fa850 45:19:02:53:177fa850 (no response)

Time requested at 0xce60faec:0x17884010 45:19:03:08:17884010 (no response)

Time requested at 0xce60fafb:0x17902080 45:19:03:23:17902080

Server rx: 0xce60fafb:0x1f6c0505 45:19:03:23:1f6c0505
Server tx: 0xce60fafb:0x1f73019d 45:19:03:23:1f73019d
Received at: 0xce60fafb:0x2076a4e0 45:19:03:23:2076a4e0

New time: 0xce60fafb:0x23e2c581 45:19:03:23:23e2c581  (new synchronisation)
```

Note that the fraction part of the time is displayed as UTC long word value and not converted to a second fraction.

Typically there is a correction of a fraction of a second between each request to different servers.

3. Calculating the Synchronised Time from Timestamps

Calculating the present time is based on timestamps in the transmission and reception UDP frames:

- Originate Timestamp T1 time request sent by client
- Receive Timestamp T2 time request received by server
- Transmit Timestamp T3 time reply sent by server
- Destination Timestamp T4 time reply received by client

The roundtrip delay (d) and system clock offset (t) are defined as:

- d = (T4 - T1) - (T3 - T2)
- t = ((T2 - T1) + (T3 - T4)) / 2.

Sanity checks to avoid inaccuracy in case of server errors or malicious server attacks are performed before the calculation is made, as recommended by RFC 4330:

1. When the IP source and destination addresses are available for the client request, they should match the interchanged addresses in the server reply.

2. When the UDP source and destination ports are available for the client request, they should match the interchanged ports in the server reply.

3. The Originate Timestamp in the server reply should match the Transmit Timestamp used in the client request.

4. The server reply should be discarded if any of the LI, Stratum, or Transmit Timestamp fields is 0 or the Mode field is not 4 (uni-cast) or 5 (broadcast).

4. Kiss-o'-Death Packet

A sever can tell a client to stop making any more requests and go to another server. If the server returns Stratum field of 0 it is to be interpreted as this case and an alternative server used if available.

The demo project's list of SNTP servers includes a field called ucAvoidServer. If a server responds with the Kiss-o'Death packet this field is set so that the server is no longer requested during the test sequence.

5. Poll Interval

The demo project requests a new synchronisation once every 15s. The real polling rate will depend on the local crystal accuracy and will generally be much less frequent than this. If a server doesn't respond it is recommended to exponentially increase the polling interval (exponential back-off) until the maximum tolerated interval is reached.

6. Example of Synchronised Time Calculation

When the SNTP client wants to start a new synchronisation it first takes a time stamp of the local time, which is put into the transmission message. The UDP frame is sent out to the SNTP server, which generates an answer with two additional time stamp fields:

- Reception time of the request
- Transmission of the answer

This response is received some time later by the client, which again makes a time stamp of the reception, resulting in the following 4 time stamps being available:

- Transmission time of original request referenced to the *local time*
- Reception time of the answer referenced to the *local time*
- Reception time of the request referenced to the *server time*
- Transmission time of the response referenced to the *server time*

As an example, the following times are used:

- The client sends at 0xce25e411:0x50027654 and receives the response at 0xce25e412:0x18248019 [local times]
- The server receives at 0xce25e413:0x44b01506 and responds at 0xce25e413:0x44b01506 [server times] (note that the reaction time at the server is 0 in this case but is often a small fraction of a second value)

This gives a round trip time of the delay between sending and receiving the message, minus the server delay (zero in this case).

0xce25e412:0x18248019 - 0xce25e411:0x50027654 = 0x00000000:0xc82209c5 (0.781769s)

This value is not actually relevant for the clock synchronisation adjustment.

The system clock adjustment is given by the server delay, plus the delay from server transmission to client reception (with differing time references) divided by 2.

(0 + (0xce25e412:0x18248019 - 0xce25e413:0x44b01506))/2 = 0x00000001:0x909c99cf

This example shows that the clock offset is a positive value because the client is behind the server time. The client thus needs to adjust its time by +1.564889. If the clock offset is a negative value the client is ahead of the server and so the correction is also negative.

The previous calculation can be verified by using the UTC data/time values of the time stamps, which are:

- Client sent time stamp: 6th August 2009 – 23:21:53.31254
- Server received time stamp: 6th August 2009 – 23:21:55.2683
- Server sent time stamp: 6th August 2009 – 23:21:55.2683
- Client received time stamp: 6th August 2009 – 23:21:53.0943

This can be represented graphically to aid in understanding:

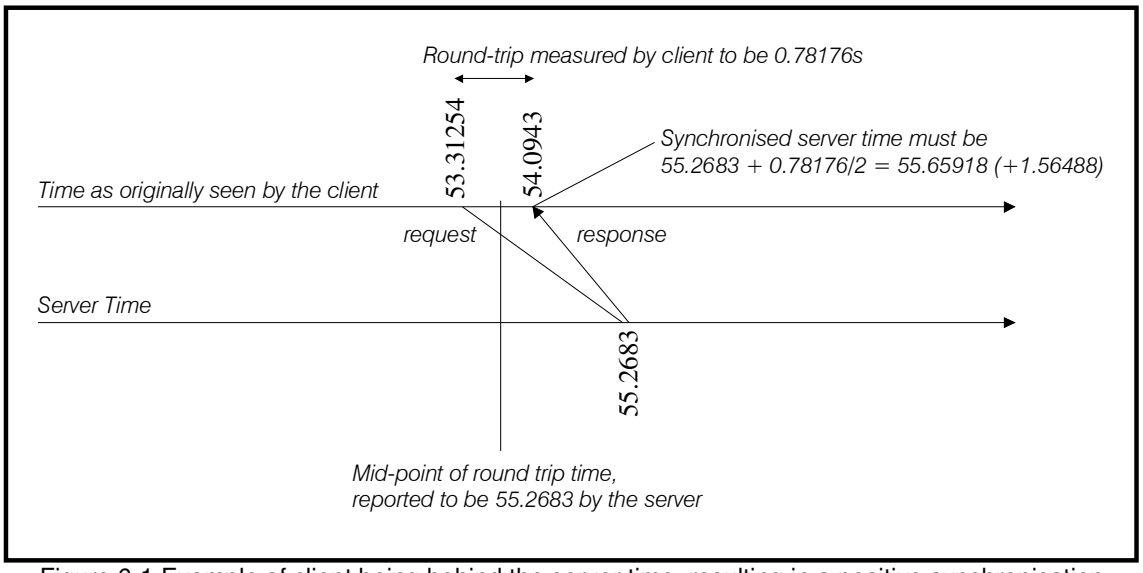

Figure 6-1 Example of client being behind the server time, resulting in a positive synchronisation correction

Consider the case when the client's clock is running ahead of the server's clock. In this case a negative correction is required. The time format calculation increases in complexity when this has to be taken in to account and, rather than generally require more complication in the arithmetic routines, the client time stamps are manipulated (decreased by the same number of full seconds until all arithmetic becomes positive), resulting in the same synchronization correction but more stream-lined calculation.

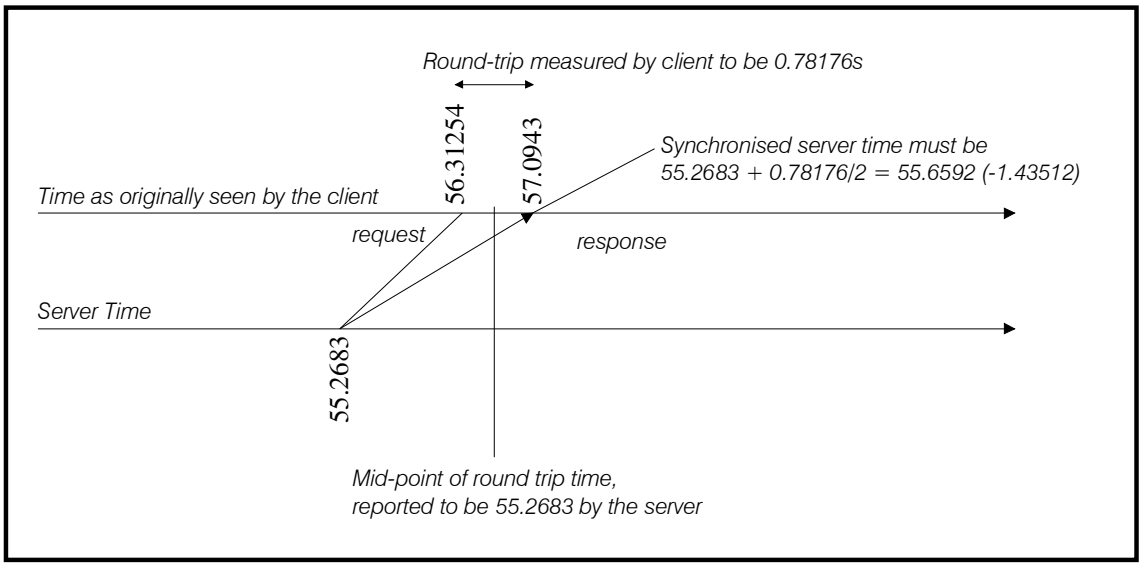

Figure 6-2 Example of client being ahead of the server time, resulting in a negative synchronisation correction

From the above diagram 57.0943 – 1.43512 = 55.6592. That is, the present time needs to be corrected by -1.43512.

As reference, assume that the client time stamps are both reduced by 2s and the same calculation performed. This time the result is 55.0943 + 0.56488 = 55.6592, which is the same value but without having to handle negative time stamps differences.

The demo project includes the following routines for arithmetic operations on two time stamps in UTC format:

```
static void fnSubtractTime(SNTP_TIME *result, SNTP_TIME *input, SNTP_TIME *minus);
static void fnAdditionTime(SNTP_TIME *result, SNTP_TIME *input, SNTP_TIME *add);
static void fnDiv2Time(SNTP_TIME *result, SNTP_TIME *input);
```

As discussed in the negative synchronisation case, negative arithmetic can be avoided. The routine `fnSubtractTime()` therefore always expects to subtract an older time stamp from a more recent time stamp.

7. Local Time

Since the client needs to time-stamp its SNTP transmission and measure the round trip delay from transmission until receiving the response it is necessary to have a local hardware timer. In some cases it may be possible to share this task with the system tick but more generally a dedicated hardware timer may be more appropriate.

The SAM7X has three 16 bit timers and it is assumed that one of these is free for dedicated use as a local time stamp timer, enabling the counting of seconds over a long period of time and also accurate time-stamp information in the fractional second region.

By configuring one timer to generate a 1Hz interrupt, the interrupt routine can count the seconds. The hardware timer also counts local clock pulses and the momentary count value can thus be used as a fraction of a second time-stamp.

When the processor starts it doesn't have any local reference time and so a default time is first configured as discussed in chapter 2. Timer 2 is used as a free running 1s interrupt counter. Although it is possible to read the hardware count value at any time in the SAM7X it is not possible to write its value. For this reason the hardware timer is reset to zero count value on each synchronisation and an offset value held locally to enable the hardware count value to be converted to a synchronised fractional value when required. Initially the offset value is zero.

It is imperative to be able to obtain an accurate local time-stamp value whenever required. This is necessary during the SNTP synchronisation sequence and also for application time-stamp use. The following routine was designed to achieve this requirement from the free-running 1s timer in the SAM7X. Note also that the timer is never stopped, so there is also no accrued inaccuracy resulting from a large number of time-stamp requests.

```
// Get the present time stamp
//
static void fnGetPresentTime(SNTP_TIME *PresentTime)
{
    uDisable_Interrupt();              // don't allow second interrupt to disturb,
                                          but don't stop timer either
    PresentTime->ulSeconds  = ulUTC_second; // present seconds value
    PresentTime->ulFraction = GET_TIMER_PRESENT_VALUE();
    if (PresentTime->ulFraction < 100) {// is it possible that the timer overflowed
                                           while reading present value?
        uEnable_Interrupt();           // if a second overflow occurred during the
                                        protected region it will be take place now
        PresentTime->ulSeconds  = ulUTC_second;    // present seconds value,
                                                  possibly after interrupt increment
        uDisable_Interrupt();          // block again
    }
    uEnable_Interrupt();               // allow timer to interrupt again since the
                                    values have been captured and are synchronised
    PresentTime->ulFraction += usFractionOffset; // respect offset to the local
                                                    timer (can only be positive)
    if (PresentTime->ulFraction >= TIMER_PERIOD_FULL_SCALE) {
        PresentTime->ulSeconds++;
        PresentTime->ulFraction -= TIMER_PERIOD_FULL_SCALE;
    }
    PresentTime->ulFraction *= TIMER_PERIOD_FULL_SCALE; // convert to fraction of a
                                                    second, assuming 16 bit full scale
}
```

The important characteristic of the time-stamp retrieval is that the seconds counter (`ulUTC_second`) remains synchronised to the free-running hardware timer even when the

time-stamp request is made at a critical instance where it rolls over from its maximum to minimum value. Since the value `ulUTC_seconds` and the hardware timer cannot be requested at the same point in time there is a risk that the hardware timer triggers an interrupt while the second value is being read. This means that the hardware timer value will be very low and the seconds value be one second behind since the second interrupt is still waiting to be processed; an error of up to 1s can result if this is not protected against.

The time stamp retrieval solution protects second count increments by disabling the interrupt routine when retrieving the second value and the hardware counter value. The routine takes only a few clock cycles to complete and the possibly critical case can be recognised by the fact that a low hardware count value has been read – if the value is high there is no possibility of an overflow second interrupt waiting.

In the possibly critical case the interrupts are freed again and the `ulUTC_second` value read for a second time. If there has been no overflow missed (no waiting timer interrupt) the second count value will still be the same. If there were a waiting timer overflow interrupt the interrupt would immediately be taken so that the subsequent read of the `ulUTC_second` will be an incremented value.

This technique thus avoids the potential race state. The routine blocks interrupts for only a very short period of time and the process or reading time stamps doesn't cause any inaccuracy in the hardware timer count value since it is never stopped in the process.

8. Conclusion

SNTP has been introduced as a simple method of synchronising local time.

The implementation in the µTasker project has been described, based on a demonstration which requests the time every 15s sequentially from a list of pre-defined SNTP servers.

The availability of an accurate local time stamp time is imperative to the operation of the SNTP client and application time-stamp use. The realisation of this time stamp timer from a hardware timer in the SAM7X has been discussed in details due to its critical roll.

Modifications:

V1.00 20.9.2009: First version