Peculiarities of NTP protocol realization for microprocessor systems with limited computing resources

D A Plotnikov, W K M Aljeazna, V I Lachinand, K Yu Solomentsev

Platov South-Russian State Polytechnic University (NPI), 132, Prosveshcheniya st., Novocherkassk, Rostov region, 346428, Russia

E-mail: dpl68@mail.ru

Abstract. The article gives brief information on the structure of the NTP protocol messages, describes the problems that arise during the formation and analysis of such messages in microprocessor systems with limited computing resources. Various ways of converting the time values into the timestamp format of the NTP protocol are analyzed, and recommendations for choosing the most appropriate method are given.

1. Introduction
An important task, solved in the development of distributed information and measurement systems, is the synchronization of the elements of these systems. It is necessary that the data coming from each element should be tied to a common time scale with an error not exceeding the permissible value. One solution to this problem is the NTP protocol [1], which provides the ability to synchronize local clocks of client devices with clock servers of the exact time. This protocol is widely used in computer networks [2, 3], as well as in embedded systems [4].

Without going into the details of the NTP protocol, described in [1, 5], we will consider the procedure for obtaining information about the need to adjust the local clock of system element, called a “client device”. As shown in Figure 1, a client device that wishes to synchronize its clock with a time server sends a request to the NTP server indicating the time of sending this request on its clock $T_1$. The server, receiving the request, fixes the time of its receipt by server clock $T_2$ and then prepares a response containing the values of $T_1$ and $T_2$. Just before the response is sent to the client, the sending time by the server clock $T_3$ is recorded to the response. The client, having received a server response containing the values of $T_1$, $T_2$ and $T_3$, remembers the time of response receipt $T_4$ by its clock.

Based on the assumption that transport delays in the delivery of the messages in the directions "client-server" and "server-client" are the same, the offset of the client clock in relation to the server clock can be calculated by the formula:

$$\text{Offs} = \frac{[(T_2 - T_1) + (T_3 - T_4)]}{2},$$  \hspace{1cm} (1)

and the total transport delay "client-server-client" – according to the formula:

$$D = (T_4 - T_1) - (T_3 - T_2).$$  \hspace{1cm} (2)

Knowing the Offs value, the client device can adjust its clock, instantly or gradually increasing their readings by this value. The more the transport delays in the directions "client-server" and "server-
client” differ from each other, the more the error of time synchronization will be. The value of $D$

together with other parameters calculated and received from the server is used to estimate this error.

![Figure 1. NTP protocol diagram: $T_1$ – request sending time; $T_2$ – request reception time; $T_3$
– response sending time; $T_4$ – response reception time.](image)

![Figure 2. NTP timestamp format.](image)

2. Detailed problem description

Timestamps $T_1$ ... $T_4$ are transmitted in the so-called “NTP Timestamp Format” [1] and are
represented by two 32-bit words (Figure 2). The first word contains the number of full seconds passed
from 0 hours on January 1, 1900; the second word is the fraction of the current second. These data can
be considered as single 64-bit word representing a fixed-point unsigned number [6] in the “32.32”
format (that is, 32 bits for the integer part and 32 bits for the fractional one).

Calculations using equations (1, 2) can be performed in a fixed-point format without any additional
transformations. The operations of addition and subtraction of numbers in this format are exactly the
same to operations with integers and are effectively implemented even by the simplest
microprocessors. In [1] it is noted that if the discrepancy between the clocks of the client and the
server does not exceed 34 years, the calculations will give the correct result.

However, the NTP timestamp format does not usually correspond to the time representation format
in microprocessor systems (MPS) operating with fractions of seconds. In NTP the timestamp fraction
of a second is a binary fraction (i.e. fraction with base “2”). On the other hand, when implementing the
MPS, it is more convenient to count the time in decimal fractions of a second: tenths, hundredths,
thousandths, etc. This is due to ensuring the necessary accuracy of timestamps: as a rule, the time
value is represented to the end user in the form of hours, minutes, seconds and decimal fractions of
seconds. If we count the time in binary fractions of a second, then when they convert to decimal, a
rounding error will occur (for example, the value 0.1 s cannot be represented as a finite binary
fraction).

The time counter in the MPS is implemented in various ways. It can be a single integer containing
the number of decimal fractions of a second elapsed from a certain date (for example, the number of
milliseconds since 0 hours on January 1, 2000). Sometimes two integers are used to simplify internal
conversions of date and time: one contains the number of complete days elapsed from a certain date,
the other - the number of decimal fractions of a second that have passed since the beginning of the
current day. There are other options, but in any case, there is some counter that contains the number of
decimal fractions of a second. Therefore, to get timestamps $T_1$ and $T_4$, the integer value of this counter
must be converted to a binary fraction. To perform the clock correction, the binary fraction $Offs$
obtained by equation (1) as a result of interaction with the NTP server must be converted into an
integer value equal to the number of decimal fractions of a second, which must be added to the MPS time counter.

Using double precision floating-point calculations to perform the described transformations is suggested in [1]. For example, to convert an integer value of a milliseconds counter to a binary fraction of the "32.32" format, it is divided by a real constant 1000.0 and then multiplied by \(2^{32}\) and rounded to the integer (of course, we can immediately multiply the counter by the constant \(2^{32} / 1000.0\)). To convert the Offs value from the "32.32" format to the number of milliseconds, we have to divide it as a single 64-bit integer number by the real constant \(2^{32}\), multiply the resulting value by 1000 and round it to an integer. Then this integer have to be added to the millisecond counter of the MPS in order to correct it.

The described transformations are easily performed by powerful MPS, especially with hardware floating-point calculations support. However, in the era of "Internet of Things" microprocessor controllers are integrated into a wide variety of items: coffee machines, kettles, alarm clocks, home security devices, etc. Many of them have to synchronize their internal clock with the time service. The use of relatively expensive high-performance controllers in these applications can significantly increase their cost. Simple, inexpensive controllers is more appropriate, but they have limited computational resources, so floating-point calculations can take an unacceptably long time.

3. Materials and methods
We can try to reduce the format conversions time by avoiding floating-point calculations and using fixed-point arithmetic. Real numbers with a fixed point are represented as \(M \times 2^P\), but unlike floating-point numbers, the mantissa \(M\) is an integer, and the exponent \(P\) is a constant implied implicitly. When using such numbers, arithmetic operations are performed only with integer mantissas, which significantly reduces the requirements for the processor computing resources. For example, there are two numbers \(X1 = M1 \times 2^{P1}\) and \(X2 = M2 \times 2^{P2}\). Then the product \(X1 \times X2\) can be written as \((M1 \times M2) \times 2^{(P1 + P2)}\). Since the exponents are implied implicitly, it is only necessary to perform integer multiplication \(M1 \times M2\), keeping in mind that the exponent of the result will be equal to \(P1 + P2\), and the bit depth of the product is the sum of the numbers of bits in the factors \(M1\) and \(M2\). When performing addition and subtraction, you first need to align the exponents of numbers if necessary, which is easily accomplished by shifting the mantissa. The disadvantages of fixed-point numbers include a much smaller range of possible values than floating-point numbers, as well as the need to monitor exponent overflow and changes.

Let us consider several examples of the fixed-point arithmetic use in relation to the time format conversion problem. Let the MPS time counter contain the number of milliseconds \(N_{MS}\) that have passed since 0:00:00 January 1, 2000. If we limit the value of this counter to the December 31, 2009, which is quite acceptable for the vast majority of currently used MPS, then it is not difficult to determine that 42 bits is sufficient for it. It is necessary to convert the value of this counter into NTP timestamp format "32.32" (Figure 2).

3.1. Example 1
In the simplest case, we can get two values from \(N_{MS}\) – the total number of seconds \(N_S\) and the number of milliseconds of the current second \(n_{MS}\), – using integer division (div) and remainder (mod)operations:

\[
N_S = N_{MS} \text{ div} 1000; \\
n_{MS} = N_{MS} \text{ mod} 1000.
\]

The bit depth of the \(N_S\) value does not exceed 32, so it can be copied to the highest NTP timestamp word. It should be kept in mind that the NTP time base is 0 hours on January 1, 1900, so \(N_S\) must be increased by a number of seconds between the dates 1.01.1900 and 1.01.2000. In this case, the overflow of the 32-bit result may occur, but this situation for the standard implementation of the NTP protocol is permissible [1].
The fractional part of the timestamp should be represented in the format "0.32". To calculate its value, the number of milliseconds \( n_{MS} \) should be divided by 1000.0, and then multiply by \( 2^{32} \) to obtain the mantissa of a fixed-point number in the "0.32" format. In other words, the desired fractional part can be obtained by an integer multiplication of \( n_{MS} \) by a constant \( K1 = 2^{32} / 1000.0 \), rounded to the integer. It should be noted that overflowing this operation is impossible, since the number of milliseconds will always be less than 1000.

In the process of the described transformations, the constant \( K1 \) rounding error takes place. In absolute terms, it does not exceed half of the least significant digit, that is, 0.5, which corresponds to a relative error of 0.5 / \( K1 \) or \( 1.16 \times 10^{-5} \% \). The second factor, \( n_{MS} \), has an exact value; its relative error is zero. Since the relative errors add up during multiplication, the result will be obtained with an error of \( 1.16 \times 10^{-5} \% \). Then the maximum absolute error of the result is 0.12 microseconds, which is acceptable for a time counter with millisecond resolution.

3.2. Example 2

The drawback of example 1 is the need to use integer division and remainder operations. If these operations are used for numbers with a bit depth higher than the processor’s, then the execution time is very long. In our case, the divider has 42 bits, and the popular ARM processors have 32 bits. To get rid of this drawback, we can immediately try to convert a 42-bit millisecond counter \( N_{MS} \) to the "32.32" format. As in example 1, the \( N_{MS} \) must be multiplied by the constant "2^{32}/1000" for this. However, if we use the constant \( K1 \) described above, the absolute error of the result will be unacceptably large. In fact, the maximum value of a 42-bit counter is approximately \( 4.4 \times 10^{12} \) ms. If a relative rounding error of the constant \( K1 \) is \( \delta_{K1} = 1.16 \times 10^{-5} \% \), then the maximum absolute error of the result will be about 510 s, while an acceptable value for a counter with millisecond resolution is 0.1 - 0.05 ms. Therefore, in this case, to convert the format, we must use the constant \( K2 \) calculated with greater accuracy and having an increased bit depth. Let us determine the number of bits \( N_{K2} \) of this constant. Based on the acceptable absolute error of 0.1 ms, the relative calculation error \( \delta_2 \) should be less or equal to 0.1 / \( (4.4 \times 10^{12}) \), that is \( 2.27 \times 10^{-12} \% \). Since one of the factors, \( N_{MS} \), has an exact value, the relative rounding error of the constant \( K2 \) \( \delta_{K2} \) should not exceed the specified value. From the formula:

\[
delta_{K2} / 100 \geq 0.5 / (2^{N_{K2}} / 1000)
\]

we get:

\[
N_{K2} \geq \log_2 (50000 / \delta_{K2}).
\]

Thus, to achieve the required value of \( \delta_{K2} \), the constant \( K2 \) bits depth must be at least 55.

Using the standard integer type "uint64_t"of the C language, we can define the \( N_{MS} \) millisecond counter and the \( K2 \) constant as 64-bit numbers. To represent the \( K2 \) constant in the "0.64" format we calculate it as \( 2^{64} / 1000.0 \) and then round to the integer. When multiplying \( N_{MS} \) and \( K2 \), we get a 128-bit result in the "64.64" format. An analysis of the possible result values shows that the highest 32 bits will always be zero, so they can be discarded. We also can discard 32 lower-order bits that represents fractions of nanoseconds and are out of interest. As a result, we get a fixed-point number in the "32.32" format, corresponding to the NTP timestamp format. Similarly, the reverse conversion from the "32.32" format to an integer equal to the number of decimal fractions of a second can be performed.

4. Results

The authors carried out an experimental study of methods for converting the timestamp format to determine their speed. The fixed-point methods considered in Examples 1 and 2 were analyzed (methods 1 and 2 respectively), as well as the standard floating-point method described in [1]. During the experiment, the duration of the millisecond counter conversion to NTP timestamp format was
determined. The study was performed on a microcontroller LPC12C24 [3] with a Cortex-M0 core [8] running at a clock frequency of 10 MHz. The experiment results are given in table 1.

### Table 1. Comparison of time format conversion methods.

|                  | Floating-point method | Fixed point method 1 | Fixed point method 2 |
|------------------|-----------------------|----------------------|----------------------|
| Duration, μs     | 85.6                  | 49.2                 | 12.0                 |
| Speed factor     | 1                     | 1.7                  | 7.1                  |

**Figure 3.** Performance of time format conversion methods: “1” – floating-point method; “2” – fixed-point method 1; “3” – fixed-point method 2.

Let us consider the advantages and disadvantages of the methods studied.

The floating-point method shown in [1] is as simple to understand as possible, it ensures the portability of programs by using only the standard features of the C language, does not require analysis of calculation errors due to the high accuracy of the data types applied. However, the speed of this method was the lowest, which limits its use in the MPS with low performance.

The fixed-point method considered in example 1 showed a higher speed, also using only the standard capabilities of the C language. The disadvantages of this method include the need to evaluate the calculation error, and the use of resource-intensive operations to find the remainder and integer division of large numbers. It should be noted that the second disadvantage is absent on 64-bit processors with hardware support for division operations.

The fixed-point method considered in Example 2 has the maximum speed: the conversion is performed seven times faster than it is in the first case. However, this method requires multiplying 64-bit numbers to produce a 128-bit result. 128-bit numbers are not currently supported by many widely used C compilers. Trying to implement the multiplication function programmatically using standard C types leads to a significant complication of the program and a decrease in its performance. Therefore, multiplication was implemented in the form of a simple subroutine in the assembly language, but this worsened the portability of the program. Nevertheless, the method can be recommended in cases where portability to processors of a different architecture is not an essential requirement.

### 5. Conclusions

Summarizing, we can conclude that the use of fixed-point arithmetic significantly accelerates the execution of time-format conversion functions when implementing the NTP protocol. This makes it advisable to use the described technique in the development of MPS with limited computing resources. In addition, reducing the processor load by optimizing the time-conversion functions can reduce its energy consumption, which is especially important for battery-powered devices.
References
[1] RFC 5905 Network Time Protocol Version 4: Protocol and Algorithms Specification. Retrieved from: http://www.ietf.org/rfc/rfc5905.txt
[2] Jie X, Liang X, Lian X and Delin Z 2017 Research on network timing system based on NTP. IEEE International Conference on Electronic Measurement & Instruments (ICEMI) DOI: 10.1109/ICEMI.2017.8265967
[3] Ruiqing P, Peng H, Wenxue Y, Min G and Bin Z 2010 The optimization techniques for time synchronization based on NTP. 2nd International Conference on Future Computer and Communication DOI: 10.1109/ICFCC.2010.5497413
[4] Chao C, Huang S and Hung H 2009 Embedded System on NTP. Fourth International Conference on Computer Sciences and Convergence Information Technology DOI: 10.1109/ICCIT.2009.180
[5] Rytilahti T, Tatang D, Köpper J and Holz T 2018 Masters of Time: An Overview of the NTP Ecosystem. IEEE European Symposium on Security and Privacy (EuroS&P) DOI: 10.1109/EuroSP.2018.00017
[6] Fixed Point Arithmetic on the ARM. © Advanced RISC Machines Ltd (ARM) 1996. Retrieved from: http://infocenter.arm.com/help/topic/com.arm.doc.dai0033a/DAI0033A_fixedpoint_appsnote.pdf
[7] LPC11Cx2 / Cx4 32-bit ARM Cortex-M0 microcontroller. Product data sheet. Rev. 3.2 - 4 January 2016. © NXP B.V. 2016. Retrieved from: http://www.nxp.com/docs/en/data-sheet/LPC11CX2_CX4.pdf
[8] Cortex-M0 Technical Reference Manual. Copyright © 2009 ARM Limited. Retrieved from: https://static.docs.arm.com/ddi0432/c/DDI0432C_cortex_m0_r0p0_trm.pdf