the high-performance real-time implementation
of TCP/IP standards

Simple Network Time Protocol (SNTP)

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Part Number: 000-1052
Revision 5.0

Contents

Chapter 1

Introduction to SNTP

The Simple Network Time Protocol (SNTP) is a protocol designed for synchronizing clocks over the Internet. SNTP Version 4 is a subset of the Network Time Protocol (NTP) and the two protocols are completely compatible and interchangeable. Both protocols utilize User Datagram Protocol (UDP) services to perform time updates in a simple, stateless protocol. Though not as complex as NTP, SNTP is highly reliable and accurate. In most places of the Internet of today, SNTP provides accuracies of 1-50 ms, depending on the characteristics of the synchronization source and network paths. SNTP has many options to provide reliability of receiving time updates. Ability to switch to alternative servers, applying back off polling algorithms and automatic time server discovery are just a few of the means for an SNTP client to handle a variable Internet time service environment. What it lacks in precision it makes up for in simplicity and ease of implementation. SNTP is intended primarily for providing comprehensive mechanisms to access national time and frequency dissemination (e.g. NTP server) services.

NetX SNTP Client Requirements

In order to function properly, the NetX SNTP Client package requires that a NetX IP instance has already been created. In addition, UDP must be enabled on that same IP instance and should have access to the *well known port 123* for sending time data to an NTP Server, although alternative ports will work as well. Broadcast clients should bind whatever UDP port their broadcast server is sending on, usually 123. The NetX SNTP Client host application must have IP addresses for one or more NTP or SNTP time servers with synchronized clocks.

Further, the host application must be able to obtain local time from an external time device or source independent of the SNTP Client thread itself **before** running the SNTP Client task. This baseline time is used by the NetX SNTP Client to apply adjustments to correlate with internal system time. The host application uses NTP server time updates to adjust its local device time. More information is available in the **Local Clock Operation** section.

NetX SNTP Client Limitations

There is no NetX SNTP or NTP Server package.

The SNTP Client API does not accommodate IPv6.

Precision in local time representation in NTP time updates handled by the SNTP Client API is limited to microsecond resolution.

The SNTP Client API does not have services for implementing DHCP and DNS discovery of time servers, although it can accommodate the host application that wishes to do so.

RFC 4330 recommends the Client re-resolve its current time server IP address at periodic intervals, ideally using DNS. However that service is not provided directly by this API. The host application can, however, update its active server list while the NetX SNTP Client task is running. However, to change servers it must suspend sending and receiving updates while resetting its NTP server.

NetX SNTP Client does not support authentication mechanisms for verifying received packet data, which RFC 4330 highly recommends for applications using multicast and manycast modes for SNTP updates. However it can accommodate the host application that wishes to do so. See the NTP Time Stamp for more information about authentication fields in the NTP time format.

NetX SNTP Client Operation

RFC 4330 recommends that SNTP clients should operate only at the leaves (highest stratum) of their local network and preferably in configurations where no NTP or SNTP client is dependent them for synchronization. Stratum level reflects the host position in the NTP time hierarchy where stratum 1 is the highest level (a root time server) and 15 is the lowest allowed level (e.g. Client). Stratum is discussed in further detail in the **NTP Time Stamp Format** section.

The NetX SNTP Client can operate in one of two basic modes, unicast or broadcast to obtain time over the Internet. In unicast mode, the Client polls its NTP/SNTP server on regular intervals and waits to receive a reply from that server. The Client verifies that the reply contains a valid time update from the intended server applying a set of 'sanity checks' recommended by RFC 4330. The Client then applies the time difference, if any, with the Server clock to its local clock. In broadcast mode, the Client listens on its own subnet for time update broadcasts and maintains its local clock after applying a similar set of sanity checks to verify the update time data. Sanity checks are described in detail in the **SNTP Client Operation** section below.

Before the Client can run in either mode, it must establish its operating parameters. This includes setting up time outs for receiving time update data, a server polling interval, and a maximum lapse of time the Client (e.g. host application) can run safely without a time update. See **SNTP Timeout Management** for more details on timeout handling.
Secondly, it must establish its time update server. To do so, it creates list of active NTP servers in order of preference. The first server on the list is the primary time server, and the remaining servers are alternative servers in the event that the primary (or alternative server) becomes unavailable. Broadcast Clients can optionally create a similer list of broadcast servers, but more importantly must supply a network domain on which the Client listens for time updates. Servers on the active broadcast list must belong to this domain. If a Client accepts a broadcast from an NTP server on its domain but not on its active server list, it will still accept server NTP packets but log a warning that this is an unknown domain server.

Once a valid time server is established, the Client essentially waits to send and receive its regularly scheduled time updates, and updates its local time accordingly. If the Client encounters a serious error during unicast or broadcast operation it suspends operation and temporarily returns control to the host application. The host application can decide

to switch to an alternative server or even switch mode of operation and attempt to resume the time update service. A more complex example than the demo_netx_sntp_client.c program is in the *Examples* directory of the NetX SNTP Client API package, which features a host application who switches servers from its active server list.

Local Clock Operation

The SNTP Client API requires the Client to be configured with a *get_local_device_time ()* and *set_local_device_time ()* callback function to be able to read and adjust local time. The default callbacks in the *nx_sntp_demo.c* file use microprocessor ticks in lieu of a local clock to get and set local time.

Before the SNTP Client runs, the host application must initialize the SNTP Client with a local time for the Client to use as a baseline time. See the *demo_netx_sntp_client.c* for an example of this.

NetX SNTP Client Timeout Management

The host application in creating and initializing the NetX SNTP Client instance must set a maximum lapse time for which it is acceptable to run without receiving a valid time update. This ‘max lapse’ is based on how long the host application can run correctly/safely and is must be computed from the drift rate of the local clock. The host application must also set a server timeout and poll interval. These are discussed in more detail in the **SNTP Unicast Operation** and **SNTP Broadcast Operation** sections below.

The SNTP Client includes an update timer, *nx_sntp_update_timer,* for which the Client sets an *nx_sntp_update_timer_timeout*. This timer enables the Client to operate within its time constraints. Every time the *nx_sntp_update_timer* expires, it decrements the time remaining from the Client max lapse by the *nx_sntp_update_timer_timeout*. The smaller the update timer timeout, the finer time resolution is available for updating the time remaining. The time remaining is saved to the Client *nx_sntp_update_time_remaining* field. This field is visible to the Client task when it is sending and receiving server time updates. When there is no time remaining, the update timer logs a warning. The SNTP Client *_nx_sntp_client_run_unicast* and *_nx_sntp_client_run_broadcast* services check the *nx_sntp_update_time_remaining* field to ensure it does not continue running once it has expired.

When a valid time update is received, the server timeout and the *nx_sntp_update_time_remaining* field is restored to the full value. The host application can also restore these values as needed.

If the Client must abort association with its time server, before resuming time service from another server it must 'reset' its parameters. This is provided by the *_nx_sntp_client_reset_broadcast* and *_nx_sntp_client_reset_unicast* services. This mainly involves clearing the current and previous server time messages saved with the Client profile, resetting the Client's current server IP address, and deactivating the update timer till the Client is ready to resume receiving time updates.

SNTP Sanity Checks

The Client examines the incoming packet for the following criteria:

- Source IP address must match the current server IP. Since the broadcast Client does not know its server IP ahead of time, it must determine if the first packet it receives is from a valid subnet time server. See the **SNTP Broadcast Operation** section for how the broadcast Client establishes its server.

- Sender source port on incoming NTP packets must match with the NetX SNTP Client task expects for server source port.

- Packet length must be the minimum length to hold an NTP time message (see figure 1).

If the packet appears to be valid, the time data is extracted from the packet buffer and a time stamp is created. The Client then applies a set of specific 'sanity checks' on the time data:

- The Leap Indicator set to 3 indicates the server is not synchronized. The Client must find an alternative server or if none is available, apply the back off algorithm (see below) to slow down its polling interval on the server (double the interval for each missed server reply) till the server responds or the Client server times out.

- A Stratum field set to zero is known as a Kiss of Death (KOD) packet. The SMTP Client KOD handler for this situation is a user defined callback. The small example demo file contains a simple KOD handler for this situation. The Reference ID field optionally contains a code indicating the reason for the KOD reply. At any rate, the KOD handler must indicate how to handle receiving a

kiss of death from the SNTP server. It may want to remove the server from the Client list, switch to an alternate server temporarily, or possibly continue service with the server (not recommended but there may be exceptional circumstances).

- The Server NTP version, stratum and mode of operation must be matched to Client service. The Client should discontinue time updates from servers failing to meet these criteria and should remove the server from their active server list.

- If the Client is configured with a server clock dispersion maximum, the Client checks the server clock dispersion on the first update received only, and if it exceeds the Client maximum, the Client rejects the server.

- The Server time stamp fields must also pass specific checks. For the unicast server, all fields must be filled in (non NULL). The Origination time stamp must be equal to the Transmit time stamp in the Client's NTP time message request. This protects the Client from malicious intruders and rogue server behavior. The broadcast server need only fill in the Transmit time stamp, since it does not receive anything from the Client it has no Receive or Origination fields to fill in.

  Certain sanity checks brand a time update as a 'bad' time update. The SNTP Client sanity check service tracks the number of consecutive bad time updates received from the same server. If a user-defined limit is reached on these bad time updates (if the server time out doesn't expire first), the Client terminates its association with this server and returns to the host application to switch servers and optionally remove the previous server from its active list. When a valid time update is received, the bad time update count is cleared.

- The host application can define a custom set of sanity checks to apply in addition to those in *nx_sntp_client_apply_sanity_check*. See the description of *nx_sntp_client_create* in Chapter 3 for specific details.

If *nx_sntp_client_apply_sanity_check* returns a non successful status to the SNTP Client, the SNTP Client checks that return status against a list of error codes that the RFC consider fatal errors. If the status is a fatal error, the SNTP Client aborts time updates from that server. See Appendix B for a list of 'fatal' errors. For error codes falling outside of

this category, the SNTP Client assumes was limited to just that particular time update packet, and handles by discarding the packet.

If the server time update passes the sanity checks, the Client then attempts to process the time data to its local time. When it receives the server update, the Client immediately obtains the local time. This is the Destination time stamp used in the round trip time calculation. Clients use this Destination time stamp and the server reply Transmit time stamp to calculate the round trip time to the server. See **Appendix A** for details on NTP round trip time calculations. The Client can be configured to abort its association with its server if it cannot compute the round trip time. Half the round trip time (e.g. the transit time from server back to Client) is added to the server clock time for the 'corrected' server time.

The Client computes the time difference between its local clock and the corrected server time. If the time difference is greater than the user defined Client minimum time adjustment, and less than the user defined Client maximum time adjustment, the Client applies the server time to its own local clock. If the time difference is greater than the Client maximum time adjustment, but this is the first update received from the current server, the Client can be configured to ignore the max adjustment limit, in order to expedite getting synchronized to server time.

NetX SNTP Client Unicast operation

An SNTP Client running in unicast mode must be initialized before starting unicast operation. This process enables the Client to add unicast servers to its active list. First it checks the NX_SNTP_UDP_UNICAST_SERVER_ADDRESSES list and adds all qualified server IP addresses to its 'active' list. Before and during the unicast Client run, the host application can add or remove servers using its GUI or autonomous discovery such as DHCP. This list is mutex protected and changes to it through the SNTP Client API are therefore thread-safe.

Next, the initialization process sets server time out and its own polling interval. The server time out is the length of time the Client can wait between valid time updates received from the same server. The poll interval is the length of time the Client must wait between sending unicast requests to its time server. The SNTP Client API continually checks the time remaining on the Client's maximum lapse time without a valid update when setting these timeouts.

If the server timeout or poll interval exceeds the ‘max lapse’ time remaining, the Client aborts unicast operation and returns control to the host application to handle the situation. As explained previously, a valid time update causes the Client to restore the time remaining on both the server time out and the *nx_sntp_update_time_remaining* field on the max lapse to the full value.

If the Client receives time updates or other UDP packets from other servers while waiting for a response from its server, it ignores these packets and continues waiting for the update packet from its server, until the poll interval since the request was sent expires.

When that time has expired, the Client resets the poll interval, sends another time request and resumes waiting for the server reply. The Client can also extend the poll interval if the back off algorithm parameter is set. The back off algorithm is increases the current poll interval each time by a constant factor e.g. doubling the poll interval. This is a reasonable approach if it appears the server is simply not synchronized yet, or is temporarily down, and the Client may not have any other valid servers to receive time updates from.

The Client has a random wait feature on startup. The Client thread is put to sleep for a random time before sending its first unicast request to the current server. This is intended to take the load of a server servicing a large number of clients possibly all restarting at the same time e.g. after a network shutdown or power failure. The random number generator is left as a user defined callback.

SNTP Manycast Operation

A variation on unicast is the manycast (or anycast) protocol. This is essentially the same as unicast protocol, except that the Client sends out a request for time update services on a preconfigured manycast IP address, and generally accepts the first server who responds with a valid time update. Thereafter it is identical to unicast operation. The RFC recommends that the Client host implement authentication before using protocols such as manycast where the host is vulnerable to rogue servers and malicious intruders.

SNTP Broadcast Operation

An SNTP Client running in broadcast mode must be initialized before starting broadcast operation. First the Client must set its broadcast

domain. In broadcast operation, the SNTP Client listens for time updates from the first server on its subnet it receives a response from.

Next, the Client adds broadcast servers to its active list. First it checks the NX_SNTP_UDP_BROADCAST_SERVER_ADDRESSES list and adds all qualified server IP addresses to its 'active' list. Before and during the unicast Client run, the host application can add or remove servers using its GUI or autonomous discovery such as DHCP. This list is mutex protected and changes to it through the SNTP Client API are therefore threadsafe.

The Client in broadcast mode does not poll its servers as it does in unicast mode. It checks the incoming packet sender IP against the current server IP to verify they match. If the Client has not established its server yet (so there is no current server IP to match against), it then looks for sender IP on its active server list. If not found, it verifies the sender IP at least originates from another host on the Client broadcast domain and logs a warning that it has received a packet from an unknown time server on its own subnet.

The broadcast Client then sets the broadcast server timeout. Since it does not poll the server there is no poll interval or Client *exponential_backoff_rate* parameter to set. Similar to unicast operation, the time remaining on the server timeout and on the max lapse time out is reset every time the Client receives a valid server update.

However, the broadcast Client can be configured with the option to send an initial unicast request. When the Client receives its first packet from a valid broadcast server on its subnet, it sends a single unicast time request to that server. If it receives no reply, the Client resumes normal broadcast operation with this server unless the Client is configured to require a round trip time calculation, in which case it rejects the server and must continue listening for another server on its subnet.

SNTP Multicast Operation

A variation on broadcast is the multicast protocol. This is identical to the broadcast protocol, except that the Client listens on a network address outside its LAN for broadcast servers. It accepts the first server who responds with a valid time update and thereafter operates in broadcast mode. The RFC recommends that the Client host implement authentication before using protocols such as multicast where the host is vulnerable to rogue servers and malicious intruders. If the Client's multicast server fails, the Client resumes listening on the multicast IP for another broadcast server.

SNTP and Multi Homed Hosts

Starting with NetX 5.3, NetX supports multi homed hosts. For how to use this feature with NetX SNTP Client, please see the **Small Example System** later in this document.

SNTP and NTP Server Time Services

The RFC 4330 prohibits unicast clients from polling servers more frequently than once per minute. An NTP server can and will deny service to clients violating this limitation. In the interests of being a 'good network citizen client polling interval should be a great as the client's host application or device can run safely. Further, if there are a large number of client's on the same subnet as a designated time server, it is recommended that clients be configured to apply a random wait on their first poll, so as to avoid bogging down network traffic and overwhelming the time server.

An SNTP Broadcast server can be set up to handle unicast requests. This is encouraged by the RFC so that their broadcast clients can compute a round trip time delay to the server.

NTP and SNTP servers should operate only at the root (stratum 1) of the subnet, and then only in configurations where no other source of synchronization other than a reliable radio clock or telephone modem is available.

NTP Time Stamp Format

The SNTP protocol uses the identical time stamp format, shown in Figure 2 below, as NTP, as part of the effort to keep NTP and SNTP hosts interoperable. The NTP time stamp contains several descriptor fields mostly for the server to share information about its time service, such as NTP version, clock precision, mode of operation, leap second warning, and NTP stratum. The remaining required fields are time stamp fields for recording when a request was received, when it was transmitted and when the server clock was last synchronized. The time stamp also contains two optional fields for authentication, Key Identifier and Message Digest.

Each time stamp represents time in a 64 bit field. The upper 32 bits contain time since the turn of the previous century (01-01-1900) in seconds, and the lower 32 bits contain the fraction of a second in fixed point notation. The SNTP Client API contains the tables and conversion formulas used in the Network Time Protocol Distribution Version 4 software (http://www.ntp.org/downloads.html) for converting time fractions in to milliseconds and microseconds. Using this format, the NTP time format will run out of range in a 32 bit field in the year 2032. The proposed plan is to roll over the seconds using an as yet unimplemented 'epoch' field which will be incremented by one for each block of time where the seconds must be rolled over. For an extensive discussion on this topic, visit http://www.eecis.udel.edu/~mills/y2k.html.

Note: The NetX SNTP Client API includes a utility for displaying NTP time in human readable format using the *nx_sntp_client_utility_convert_seconds_to_date()* and *_nx_sntp_client_utility_display_date_time()* function calls. This is demonstrated in the demo program in the *Examples* directory of the NetX SNTP Client package.

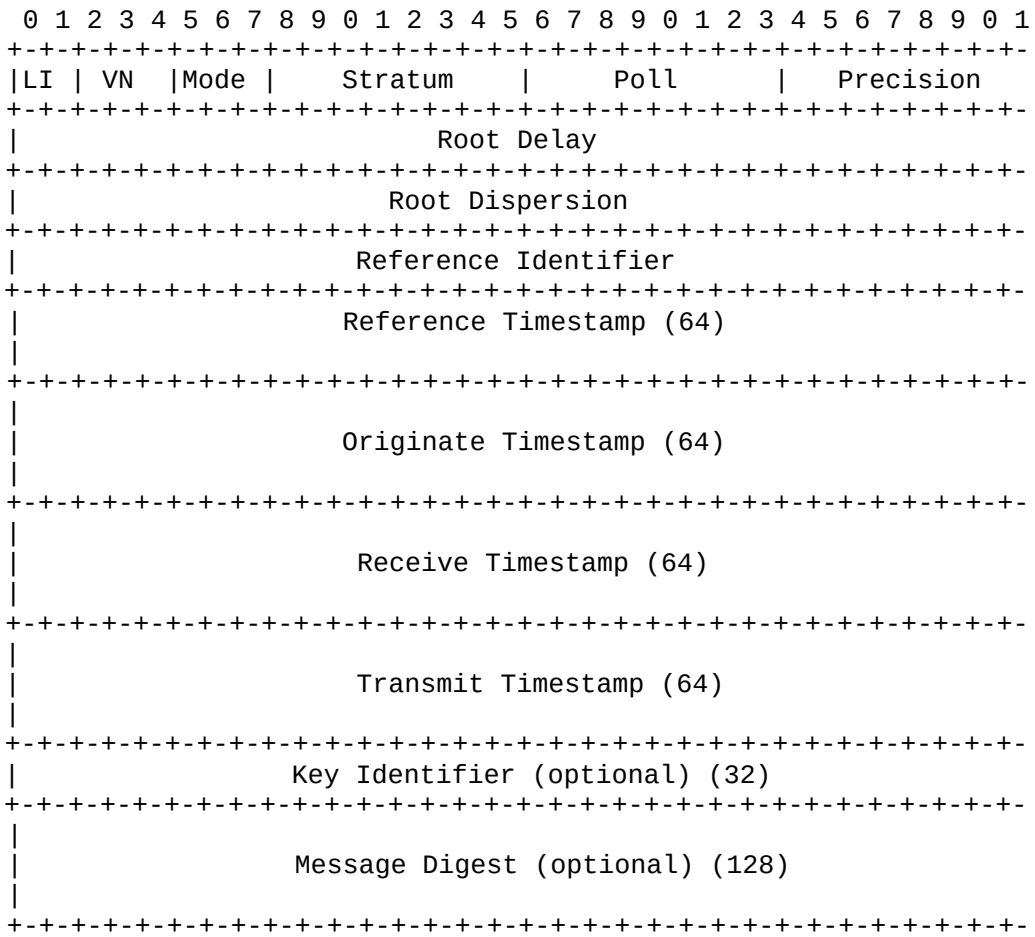

Figure 2 NTP time stamp format

For SNTP client messages, most of these fields are zero.

Leap Indicator (LI): This is a two-bit code warning of an impending leap second to be inserted or deleted in the last minute of the current day. This field is significant only in server messages, where the values are defined as follows:

LI	Meaning
0	No warning
1	Last minute has 61 seconds
2	Last minute has 59 seconds
3	Alarm condition (clock not synchronized)

On startup, servers set this field to 3 until the server clock is synchronized.

Version Number (VN): This is a three-bit integer indicating the NTP or SNTP version number.

Mode: This is a three-bit number indicating the protocol mode. The values are defined as follows:

Mode	Meaning
0	Reserved
1	Symmetric active
2	Symmetric passive
3	Client
4	Server
5	Broadcast (server)
6	Reserved for NTP control message
7	Reserved for private use

In unicast and manycast modes, the client sets this field to 3(client) in the request, and the server sets it to 4 (server) in the reply. In broadcast mode, the server sets this field to 5 (broadcast). The other modes are not used by SNTP servers and clients.

Stratum**:** This is an eight-bit unsigned integer indicating the Stratum or hierarchy among time servers. A ‘1’ indicates the top level of the hierarchy (server) and anything lower, down to 15, is either a server or more likely a client. This field is significant only in SNTP server messages. Its values are defined as follows:

Stratum	Meaning
0	kiss-o'-death message (see below)
1	primary reference (e.g., synchronized by radio clock)
2-15	secondary reference (synchronized by NTP or SNTP)
16-255	reserved

Poll Interval**:** This is an eight-bit unsigned integer used as an exponent of two, where the resulting value is the maximum interval between successive messages in seconds. This field is significant only in SNTP server messages, where the values range from 4 (16 s) to 17 (131,072 s -- about 36 h).

Precision: This is an eight-bit signed integer used as an exponent of two, where the resulting value is the precision of the system clock in seconds. This field is significant only in server messages, where the values range from -6 for mains-frequency clocks to -20 for microsecond clocks found in some workstations.

Root Delay: This is a 32-bit signed fixed-point number indicating the total roundtrip delay to the primary reference source, in seconds with the fraction point between bits 15 and 16. This data is not used in the SNTP Client API.

Code	External Reference Source
LOCL	uncalibrated local clock
CESM	calibrated Cesium clock
RBDM	calibrated Rubidium clock
PPS	calibrated quartz clock or other pulse-per-second source
IRIG	Inter-Range Instrumentation Group
ACTS	NIST telephone modem service
USNO	USNO telephone modem service
PTB	PTB (Germany) telephone modem service
TDF	Allouis (France) Radio 164 kHz
DCF	Mainflingen (Germany) Radio 77.5 kHz
MSF	Rugby (UK) Radio 60 kHz
WWV	Ft. Collins (US) Radio 2.5, 5, 10, 15, 20 MHz
WWVB	Boulder (US) Radio 60 kHz
WWVH	Kauai Hawaii (US) Radio 2.5, 5, 10, 15 MHz
CHU	Ottawa (Canada) Radio 3330, 7335, 14670 kHz
LORC	LORAN-C radio navigation system
OMEG	OMEGA radio navigation system
GPS	Global Positioning Service

Figure 3 Reference Identifier Codes

Root Dispersion**:** This is a 32-bit unsigned fixed-point number indicating the maximum error in the server clock, in seconds with the fraction point between bits 15 and 16. This field is significant only in server messages, where the values range from zero to several hundred microseconds.

Reference Identifier**:** This is a 32-bit bit string identifying the particular reference source. This field is significant only in server messages, where for stratum 0 (kiss-o'-death message) and 1 (primary server), the value is a four-character ASCII string, left justified and zero padded to 32 bits. Primary (stratum 1) servers set their Reference Identifier to a code identifying the external reference source according to Figure 3 above. If the external reference is one of those listed, the associated code should be used.

Reference Timestamp: This field is the time the system clock was last set or corrected, in 64-bit timestamp format.

Originate Timestamp: This is the time at which the request departed the client for the server, in 64-bit timestamp format.

Receive Timestamp: This is the time at which the request arrived at the server or the reply arrived at the client, in 64-bit timestamp format.

Transmit Timestamp: This is the time at which the request departed the client or the reply departed the server, in 64-bit timestamp format.

Authenticator (optional): When the NTP authentication scheme is implemented, the Key Identifier and Message Digest fields contain the message authentication code (MAC) information defined in Appendix C of RFC 1305.

Below is an actual unicast request (poll) to 207.46.130.100, a stratum 1 server from the MCF 5272 processor 192.2.2.35:

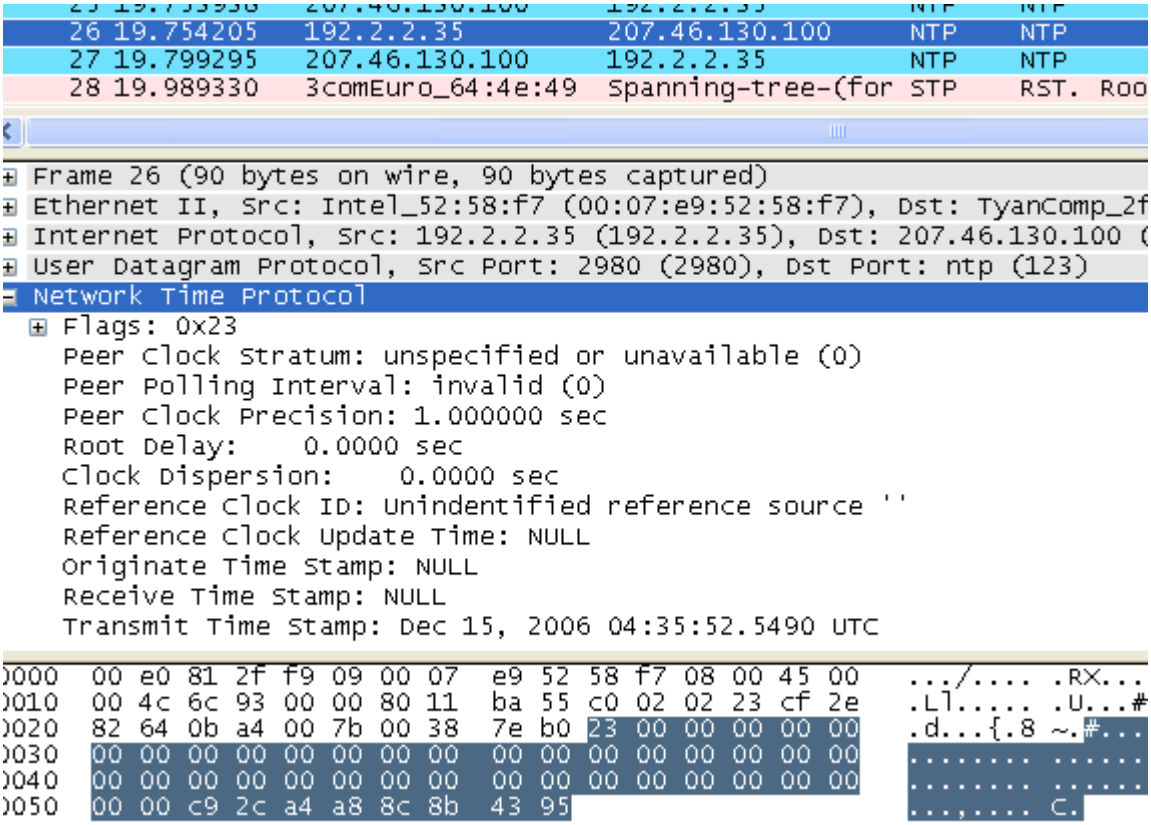

Figure 4a. Unicast server reply

Below is an actual unicast reply from 207.46.130.100, a stratum 1 server, responding to a unicast poll from the MCF 5272 processor 192.2.2.35:

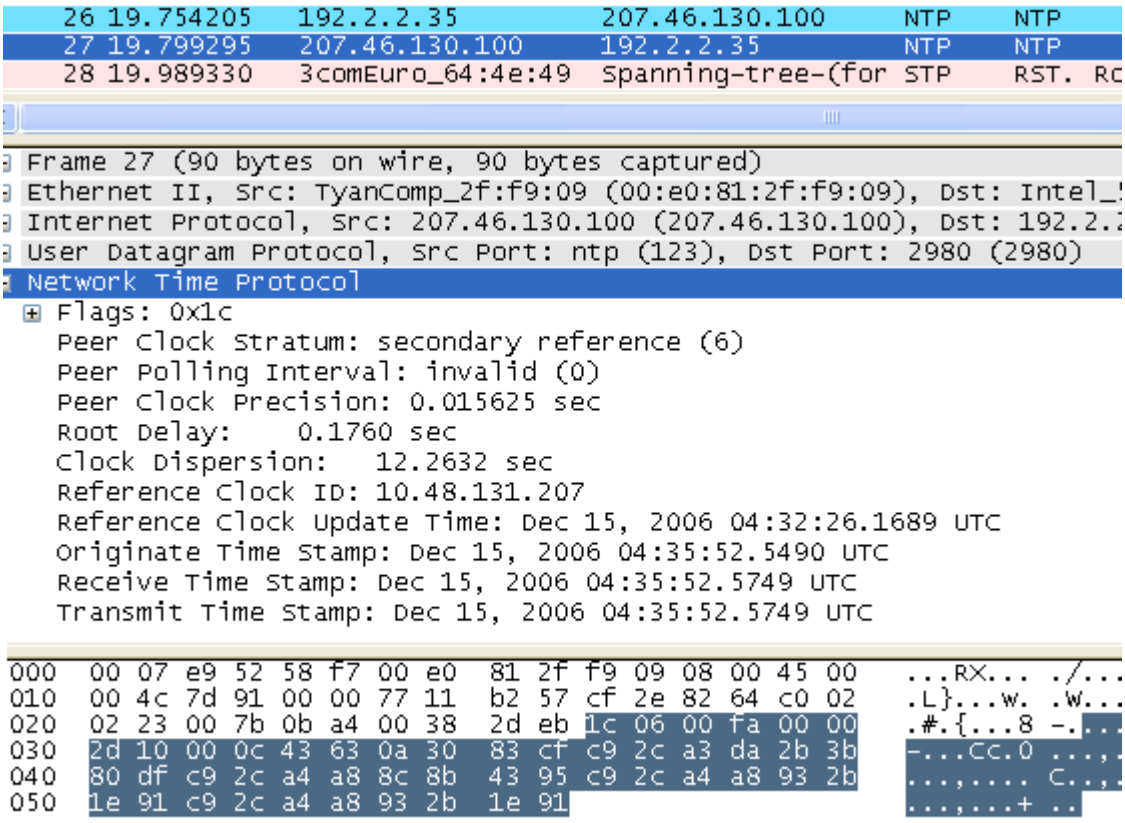

Figure 4b. Unicast server reply

SNTP Authentication

SNTPv4 includes certain optional extensions, including a public key-based authentication scheme designed specifically for broadcast and manycast applications. While authentication scheme is not formally published in an RFC, it is available in the Network Time Protocol Distribution Version 4 software (http://www.ntp.org/downloads.html). The SNTP Client provides the necessary fields in the NTP time message to contain authentication to both authenticate itself and authenticate incoming time data.

NetX SNTP Client Multi-Thread Support

The NetX SNTP Client services are not designed to be called from multiple sources. However, its active server lists are mutex protected so changes to this list are thread safe.

SNTP and NTP RFCs

NetX SNTP is compliant with RFC4330 and related RFCs.

Chapter 2

Installation and Use of NetX SNTP Client

This chapter contains a description of various issues related to installation, setup, and usage of the NetX SNTP Client.

Product Distribution

SNTP for NetX is shipped on a single CD-ROM compatible disk. The package includes two source files and a PDF file that contains this document, as follows:

nx_sntp.h	Header file for SNTP for NetX
nx_sntp_client.c	C Source file for SNTP Client for NetX
nx_sntp_client.h	Header file for SNTP Client for NetX
demo_netx_sntp_client.c	Demonstration SNTP Client application
nx_sntp.pdf	PDF description of SNTP Client for NetX

NetX SNTP Client Installation

In order to use SNTP for NetX, the entire distribution mentioned previously should be copied to the same directory where NetX is installed. For example, if NetX is installed in the directory "*threadx**arm7**green*" then the *nx_sntp.h*, *nx_sntp_client.c*, and *nx_sntp_client.h* files should be copied into this directory.

Using NetX SNTP Client

Using NetX SNTP Client is easy. Basically, the application code must include *nx_sntp.h* and *nx_sntp_client.h* after it includes *tx_api.h, fx_api.h,* and *nx_api.h*, in order to use ThreadX, FileX, and NetX, respectively. Once *nx_sntp.h* and *nx_sntp_client.h* is included, the application code is then able to make the SNTP function calls specified later in this guide. The application must also include *nx_sntp_client.c* in the build process. These files must be compiled in the same manner as other application files and its object form must be linked along with the files of the application. This is all that is required to use NetX SNTP.

Note that since SNTP utilizes NetX UDP services, UDP must be enabled with the *nx_udp_enable* call prior to using SNTP.

Small Example System

An example of how easy it is to use NetX SNTP is described in Figure 1. that appears below. In this example, the SNTP include file *nx_sntp.h* is brought in at line 14. Next, the SNTP Client is created in "*tx_application_define*" at line 157. Note that the SNTP Client control block "*demo_client* " was defined as a global variable at line 49 previously. After successful creation, an SNTP Client is started at line 182. At line 199, the SNTP Client is initialized for either unicast or broadcast operation. At line 287, the local clock is initialized. The the SNTP Client is ready to run. The Client begins unicast or broadcast operation at line 307. Note that this application runs on an MFC5272 Coldfire processor and requires Internet or network access.

```
1    /*
2       demo_netx_sntp_client.c
3
4       This is a small demo of the NetX SNTP Client on the high-performance NetX UDP/IP stack.
5       This demo relies on Thread, NetX and SNTP Client API to execute the Simple Network Time
6       Protocol for the unicast and broadcast Client.
7
8     */
9
10
11   #include  <stdio.h>
12   #include  "nx_api.h"
13   #include  "nx_ip.h"
14   #include  #nx_sntp.h"
15   #include  "nx_sntp_client.h"
16
17
18   /* Network driver/configuration for mcf5272. */
19
20   VOID    nx_etherDriver_mcf5272(NX_IP_DRIVER *driver_req_ptr);
21
22   /* Utilities for accessing and modifying device time. These assume the
23      processor is the local device clock. */
24
25   ULONG           base_ticks;
26   NX_SNTP_TIME    baseNTPtime;
27
28
29   /* Application defined services of the NetX SNTP Client. */
30
31   UINT get_local_device_time(NX_SNTP_TIME *time_ptr);
32   UINT set_local_device_time(NX_SNTP_TIME *time_ptr);
33   UINT leap_second_handler(NX_SNTP_CLIENT *client_ptr, UINT leap_indicator);
34   UINT kiss_of_death_handler(NX_SNTP_CLIENT *client_ptr, NX_SNTP_TIME_MESSAGE *server_time_msg_ptr);
35
36
37   /* Local services (not part of SNTP Client configuration). */
38
39   UINT convert_ticks_NTPTime(ULONG ticks, NX_SNTP_TIME *time_ptr);
40   UINT initialize_local_device_time(NX_SNTP_TIME *NTPtime, ULONG ticks);
41
42
43   /* Set up client thread and network resources. */
44
```

```
45   NX_PACKET_POOL      client_packet_pool;
46   NX_IP               client_ip;
47   NX_UDP_SOCKET       client_socket;
48   TX_THREAD           demo_client_thread;
49   NX_SNTP_CLIENT      demo_client;
50
51
52
53   /* Set up client thread entry point. */
54   void    demo_client_thread_entry(ULONG info);
55
56   /* Define main entry point.  */
57   int main()
58   {
59       /* Enter the ThreadX kernel.  */
60       tx_kernel_enter();
61   }
62
63   /* Define what the initial system looks like.  */
64   void    tx_application_define(void *first_unused_memory)
65   {
66
67   UINT       status;
68   UCHAR    *free_memory_pointer;
69
70
71       free_memory_pointer = (UCHAR *)first_unused_memory;
72
73       /* Create client packet pool. */
74       status =  nx_packet_pool_create(&client_packet_pool, "SNTP Client Packet Pool",
75                                       NX_SNTP_CLIENT_PACKET_SIZE, free_memory_pointer,
76                                       NX_SNTP_CLIENT_PACKET_POOL_SIZE);
77
78       /* Check for errors. */
79       if (status != NX_SUCCESS)
80       {
81
82           NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Error creating packet pool. Status: 0x%x\n\r", status));
83
84           return;
85       }
86
87       /* Initialize the NetX system. */
88       nx_system_initialize();
89
90       /* Update pointer to unallocated (free) memory. */
91       free_memory_pointer =  free_memory_pointer + NX_SNTP_CLIENT_PACKET_POOL_SIZE;
92
93       /* Create Client IP instances */
94       status = nx_ip_create(&client_ip, "SNTP IP Instance", NX_SNTP_CLIENT_IP_ADDRESS,
95                             0xFFFFFF00UL, &client_packet_pool,  nx_etherDriver_mcf5272,
96                             free_memory_pointer, NX_SNTP_CLIENT_IP_STACK_SIZE,
97                             NX_SNTP_CLIENT_IP_THREAD_PRIORITY);
98
99       /* Check for error. */
100      if (status != NX_SUCCESS)
101      {
102
103          NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Error creating IP instance. Status: 0x%x\n\r", status));
104
105          return;
106      }
107
108      free_memory_pointer =  free_memory_pointer + NX_SNTP_CLIENT_IP_STACK_SIZE;
109
110      /* Enable ARP and supply ARP cache memory. */
111      status =  nx_arp_enable(&client_ip, (void **) free_memory_pointer, 2048);
112
113      /* Check for error. */
114      if (status != NX_SUCCESS)
115      {
```

```
116
117         NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Client error enabling ARP. Status 0x%x\n\r", status));
118
119         return;
120     }
121
122     /* Update pointer to unallocated (free) memory. */
123     free_memory_pointer = free_memory_pointer + 2048;
124
125     /* Enable UDP for client. */
126     status =  nx_udp_enable(&client_ip);
127
128     /* Check for error. */
129     if (status != NX_SUCCESS)
130     {
131
132         NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Client error enabling UDP. Status 0x%x\n\r", status));
133
134         return;
135     }
136
137     /* Create the client thread */
138     status = tx_thread_create(&demo_client_thread, "Client_thread", demo_client_thread_entry,
139                               (ULONG)(&demo_client), free_memory_pointer,
140                               NX_SNTP_CLIENT_STACK_SIZE, NX_SNTP_CLIENT_THREAD_PRIORITY,
141                               NX_SNTP_CLIENT_PREEMPTION_THRESHOLD,
142                               NX_SNTP_CLIENT_THREAD_TIME_SLICE, TX_DONT_START);
143
144     /* Check for errors */
145     if (status != TX_SUCCESS)
146     {
147
148         NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Error creating client thread. Status: 0x%x ", status));
149
150         return;
151     }
152
153     /* Update pointer to unallocated (free) memory. */
154     free_memory_pointer = free_memory_pointer + NX_SNTP_CLIENT_STACK_SIZE;
155
156     /* Create the SNTP Client to run in unicast mode with test mode turned off. */
157     status =  nx_sntp_client_create(&demo_client, &client_ip, &client_packet_pool,
158                                     NX_SNTP_CLIENT_TIME_TO_LIVE, NX_SNTP_CLIENT_MAX_QUEUE_DEPTH,
159                                     NX_SNTP_CLIENT_UDP_PORT, UNICAST_MODE,
                                        NX_SNTP_CLIENT_MIN_TIME_ADJUSTMENT,
160                                     NX_SNTP_CLIENT_MAX_TIME_ADJUSTMENT,
                                        NX_SNTP_CLIENT_EXP_BACKOFF_RATE,
161                                     NX_SNTP_CLIENT_MAX_TIME_LAPSE, NX_SNTP_CLIENT_BAD_UPDATE_LIMIT,
162                                     NX_SNTP_CLIENT_MAX_ROOT_DISPERSION,
                                        NX_SNTP_CLIENT_IGNORE_MAX_ADJUST_STARTUP,
163                                     NX_SNTP_CLIENT_RUN_IN_TEST_MODE,
164                                     get_local_device_time,
165                                     set_local_device_time,
166                                     NULL /* No apply_custom_sanity_checks callback */,
167                                     NULL /* no adjust_local_device_time callback */,
168                                     leap_second_handler,
169                                     kiss_of_death_handler,
170                                     NULL /* no random_number_generator callback */);
171
172     /* Check for error. */
173     if (status != NX_SUCCESS)
174     {
175
176         NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Error creating SNTP client. Status 0x%x\r\n", status));
177
178         /* Bail out!*/
179         return;
180     }
181
182     tx_thread_resume(&demo_client_thread);
183
```

```
184     return;
185 }
186
187
188 /* Define the client thread.  */
189 void    demo_client_thread_entry(ULONG info)
190 {
191
192 UINT                status;
193 NX_SNTP_CLIENT      *client_ptr;
194
195
196     client_ptr = (NX_SNTP_CLIENT *)info;
197
198     /* Set up client time updates depending on client mode. */
199     if (client_ptr -> operating_mode == UNICAST_MODE)
200     {
201         /* Initialize the Client for unicast mode with manycast turned off. */
202         status = nx_sntp_client_initialize_unicast(client_ptr,
                                                    NX_SNTP_CLIENT_UNICAST_SERVER_TIMEOUT,
203                                                 NX_SNTP_CLIENT_UNICAST_POLL_INTERVAL, NX_TRUE,
                                                    NX_FALSE,
204                                                 NX_SNTP_CLIENT_MANYCAST_ADDRESS,
205                                                 NX_SNTP_UDP_UNICAST_SERVER_ADDRESSES);
206     }
207     else
208     {
209         /* Initialize the Client for broadcast mode with multicast turned off. */
210         status = nx_sntp_client_initialize_broadcast(client_ptr,
                                                      NX_SNTP_CLIENT_BROADCAST_SERVER_TIMEOUT,
211                                                   NX_SNTP_CLIENT_INITIAL_UNICAST_TIMEOUT,
212                                                   NX_TRUE, NX_FALSE,
                                                      NX_SNTP_CLIENT_BROADCAST_DOMAIN,
213                                                   NX_SNTP_CLIENT_MULTICAST_ADDRESS,
214                                                   NX_SNTP_UDP_BROADCAST_SERVER_ADDRESSES);
215     }
216
217     /* Check for error. */
218     if (status != NX_SUCCESS)
219     {
220
221         NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Unable to initialize client update task. Status
                                           0x%x\r\n", status));
222
223         return;
224     }
225
226     /* Create a udp socket to receive SNTP time data. */
227     status =  nx_udp_socket_create(client_ptr -> ip_ptr, &(client_ptr -> udp_socket),
228                                    NX_SNTP_CLIENT_UDP_SOCKET_NAME, NX_IP_NORMAL,
229                                    NX_FRAGMENT_OKAY, client_ptr -> time_to_live,
230                                    client_ptr -> max_queue_depth);
231
232     /* Check for error. */
233     if (status != NX_SUCCESS)
234     {
235
236         /* Log the event. */
237         NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Error creating socket. Status (0x%x)\r\n", status));
238
239         return;
240     }
241
242     /* Bind the UDP socket to the IP port.  */
243     status =  nx_udp_socket_bind(&(client_ptr -> udp_socket),
244                                  NX_SNTP_CLIENT_UDP_PORT, NX_WAIT_FOREVER);
245
246     /* Check for error. */
247     if (status != NX_SUCCESS)
248     {
249
```

```
250         /* Log the event. */
251         NX_SNTP_CLIENT_EVENT_LOG(MODERATE, ("Error binding socket to port. Status 0x%x", status));
252
253         /* Release threadX and NetX resources held by client.  */
254         nx_sntp_client_delete(client_ptr);
255
256         return;
257     }
258
259
260  /* This section initializes (sets) the local clock.  How the Client local clock should
261     be initialized is ENTIRELY up to the host application.  Below is one
262     approach to using the processor timer ticks
263     to substitute for an actual RTC and interface for setting
264     year/month/day/hour/minute/seconds  */
265
266
267     /* Set up baseline variables for using processor as local clock. */
268
269     /* This sets large offset in processor ticks (so we can adjust clock ahead or behind)
270        rather than simply setting base ticks to zero. */
271     base_ticks = client_ptr -> max_time_adjustment / NX_SNTP_MILLISECONDS_PER_TICK;
272     memset(&baseNTPtime, 0, sizeof(NX_SNTP_TIME));
273
274     /* Set a time benchmark of Feb 28, 11:19:05.2199 PST (Feb 28, 2007 19:19:05.2199 UTC).
275        This initial time will enable the Client to process the first actual server time
276        update.   */
277     baseNTPtime.seconds =  0xC9905429;
278     baseNTPtime.fraction = 0x3849BA5E;
279
280   /* Use the benchmark time to compute seconds up to the current time:
281        e.g. Mar 22, 2007 6:30pm PST (note that we don't worry about hours
          or minutes since this is merely a base line time and will be corrected
          on the first NTP time message from the NTP server: */ 282
282       baseNTPtime.seconds +=  22  * 84600 /*  days */
283                           +   0   * 3600  /*  hours */
284                           +   0   * 60;   /*  minutes */
285
286
287     initialize_local_device_time(&baseNTPtime, base_ticks);
288
289  /* At this point, the local clock must have some kind of reference time
290     for comparing with update time data from the server.  */
291
292
293     /* Check for valid (even if inaccurate) local time. */
294     status = (client_ptr -> get_local_device_time)(&(client_ptr -> local_ntp_time));
295
296     if (status != NX_SUCCESS)
297     {
298
299         NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Aborting SNTP Client task. No local time set. "
300                                           "Status 0x%x\r\n", status));
301
302         return;
303     }
304
305
306     /* Run whichever service the client is configured for. */
307     if (client_ptr -> operating_mode == UNICAST_MODE)
308     {
309
310         /* This includes (m)anycast configured clients. */
311         status = nx_sntp_client_run_unicast(client_ptr);
312     }
313     else
314     {
315         /* Do not restart the update timer. We're in a passive mode and
316            can only wait for updates. */
317
318         /* This includes multicast configured clients. */
```

```
319         status = nx_sntp_client_run_broadcast(client_ptr);
320     }
321
322
323     /* Log the Client delete status. */
324     NX_SNTP_CLIENT_EVENT_LOG(LOG, ("Client status on session termination: 0x%x \n\r", status));
325
326     NX_SNTP_CLIENT_EVENT_LOG(LOG, ("Deleting SNTP client task...\n\r"));
327
328     status = nx_sntp_client_delete(client_ptr);
329
330     return;
331 }
332
333
334 /* This application-defined handler for getting local time is required by SNTP
335    Client.  It is used in the Client API for getting the local time and providing
336    it to the Client in NTP time format. The default handler below uses the
337    device microprocessor timer ticks in lieu of a time controller like an RTC. */
338
339 UINT get_local_device_time(NX_SNTP_TIME *time_ptr)
340 {
341
342 UINT         status;
343 NX_SNTP_TIME elapsedNTPtime;
344 ULONG        current_ticks;
345 ULONG        elapsed_ticks;
346
347
348      /* Check for local clock not initialized.  */
349      if (!base_ticks || !baseNTPtime.seconds)
350      {
351
352          /* Log the error. */
353          NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Local device clock is not initialized!\r\n"));
354
355          /* Return the error condition. */
356          return NX_SNTP_INVALID_LOCAL_TIME;
357      }
358
359
360      /* Clear memory for storing current time data. */
361      memset(&elapsedNTPtime, 0, sizeof(NX_SNTP_TIME));
362      memset(time_ptr, 0, sizeof(NX_SNTP_TIME));
363
364      /* Compute elapsed time from timer ticks since base time. */
365      current_ticks = tx_time_get();
366      elapsed_ticks = current_ticks - base_ticks;
367
368      /* Convert elapsed timer ticks to an NTP time format. */
369      convert_ticks_NTPTime(elapsed_ticks, &elapsedNTPtime);
370
371      /* Add to the base NTP time to get the current time (in NTP time format). */
372      status = nx_sntp_client_utility_add_NTPtime(&baseNTPtime, &elapsedNTPtime, time_ptr);
373
374      /* Check for error. */
375      if (status != NX_SUCCESS)
376      {
377
378         NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Error converting ticks to NTP time. "
379                                           "Status 0x%x\r\n", status));
380
381         /* Return the error status. */
382         return status;
383      }
384
385      return NX_SUCCESS;
386 }
387
388
389 /* This application-defined handler for setting local time is required by SNTP
```

```
390     Client.  It is used in the Client API for setting the local time according to a
391     server time update in NTP time format. The default handler below uses the
392     device microprocessor timer ticks in lieu of a time controller like an RTC. */
393
394   UINT set_local_device_time(NX_SNTP_TIME *time_ptr)
395  {
396
397      /* Check for uninitialized local clock. */
398      if (!base_ticks)
399      {
400
401          NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Local clock is not initialized!\r\n"));
402
403          return NX_SNTP_INVALID_LOCAL_TIME;
404      }
405
406      memset(&baseNTPtime, 0, sizeof(NX_SNTP_TIME));
407      memcpy(&baseNTPtime, time_ptr, sizeof(NX_SNTP_TIME));
408
409      base_ticks = tx_time_get();
410
411      return NX_SUCCESS;
412  }
413
414
415  /* This application defined handler for handling an impending leap second is not
416     required by the SNTP Client. The default handler below only logs the event for
417     every time stamp received with the leap indicator set.  */
418
419  UINT leap_second_handler(NX_SNTP_CLIENT *client_ptr, UINT leap_indicator)
420  {
421
422      if (leap_indicator == 1)
423      {
424          NX_SNTP_CLIENT_EVENT_LOG(LOG, ("Leap Indicator Warning: last minute of current day will "
425                                         "have one extra second (61 seconds).\r\n"));
426      }
427      else if (leap_indicator == 2)
428      {
429          NX_SNTP_CLIENT_EVENT_LOG(LOG, ("Leap Indicator Warning: last minute of current day will "
430                                         "have one less second (59 seconds).\r\n"));
431
432      }
433      else if (leap_indicator == 3)
434      {
435          NX_SNTP_CLIENT_EVENT_LOG(LOG, ("Warning: Server is not (yet) synchronized!\r\n"));
436      }
437
438      /* Else no warning from leap second indicator. */
439
440      return NX_SUCCESS;
441  }
442
443  /* This application defined handler for handling a Kiss of Death packet is not
444     required by the SNTP Client. A KOD handler should determine
445     if the Client task should continue vs. abort sending/receiving time data
446     from its current time server, and if aborting if it should remove
447     the server from its active server list.
448
449     Note that the KOD list of codes is subject to change. The list
450     below is current at the time of this software release. */
451
452  UINT kiss_of_death_handler(NX_SNTP_CLIENT *client_ptr, NX_SNTP_TIME_MESSAGE *server_time_msg_ptr)
453  {
454
455  UINT    KOD_code;
456  UINT    remove_server_from_list = NX_FALSE;
457  UINT    status = NX_SUCCESS;
458
459
460      /* Convert the time message code to a known server Kiss of Death condition. */
```

```
461     nx_sntp_client_utility_convert_refID_KOD_code(
462                         server_time_msg_ptr -> reference_clock_id, &KOD_code);
463
464     /* Log the kiss of death code received. */
465     NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Client received kiss of death packet. "
466                                       "KOD code 0x%x.\n\r", KOD_code));
467
468     /* Handle kiss of death by code group. */
469     switch (KOD_code)
470     {
471
472         case NX_SNTP_KOD_RATE:
473         case NX_SNTP_KOD_NOT_INIT:
474         case NX_SNTP_KOD_STEP:
475
476             /* Find another server while this one is temporarily out of service.  */
477             status =  NX_SNTP_KOD_SERVER_NOT_AVAILABLE;
478
479             NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Server 0x%x is temporarily out of service.\r\n"
480                                               "Client must find another time server updates.\r\n",
481                                               client_ptr -> server_ip_address));
482         break;
483
484         case NX_SNTP_KOD_AUTH_FAIL:
485         case NX_SNTP_KOD_NO_KEY:
486         case NX_SNTP_KOD_CRYP_FAIL:
487
488             /* These indicate the server will not service client with time updates
489                without successful authentication. */
490
491             NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Server 0x%x is unable to provide time udpates "
492                                               "because of authentication issues.\r\n",
493                                               client_ptr -> server_ip_address));
494
495             remove_server_from_list =  NX_TRUE;
496
497         break;
498
499
500         default:
501
502             /* All other codes. Remove server before resuming time updates. */
503
504             NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Server 0x%x unable to provide client with time "
505                                               "update services.\r\n", client_ptr ->
                                                  server_ip_address));
506
507             remove_server_from_list =  NX_TRUE;
508         break;
509     }
510
511     /* Removing the server from the active server list? */
512     if (remove_server_from_list)
513     {
514
515         /* Let the caller know it has to bail on this server before resuming service. */
516         status = NX_SNTP_KOD_REMOVE_SERVER;
517     }
518
519     return status;
520 }
521
522
523 /* THIS IS NOT part of the Client API! It is only used
524    to converts timer ticks to NTP time. It is intended for use
525    with the host microprocessor in lieu of an independent time controller
526    (real time clock) on board the Client. */
527
528 UINT convert_ticks_NTPTime(ULONG ticks, NX_SNTP_TIME *time_ptr)
529 {
530
```

```
531  UINT  status;
532  ULONG milliseconds;
533
534      time_ptr -> seconds = ticks / NX_SNTP_TICKS_PER_SECOND;
535      milliseconds = (ticks % NX_SNTP_TICKS_PER_SECOND) ;
536
537      if (milliseconds > (0xFFFFFFFF / NX_SNTP_MILLISECONDS_PER_TICK))
538      {
539
540          NX_SNTP_CLIENT_EVENT_LOG(MODERATE, ("Overflow error converting ticks to "
541                                              "milliseconds.\r\n"));
542          return NX_SNTP_OVERFLOW_ERROR;
543      }
544
545      milliseconds *= NX_SNTP_MILLISECONDS_PER_TICK;
546
547      status = _nx_sntp_client_utility_msecs_to_fraction(milliseconds, &(time_ptr -> fraction));
548      if (status != NX_SUCCESS)
549      {
550          NX_SNTP_CLIENT_EVENT_LOG(MODERATE, ("Overflow error converting ticks to NTP time. "
551                                              "Status 0x%x\r\n", status));
552          return status;
553      }
554
555      return NX_SUCCESS;
556  }
557
558
559  /* THIS IS NOT part of the Client API! It is only used
560     for this particular demo to initialize the local 'clock'.
561     It sets the global baseline variables used in other local clock
562     services. */
563  UINT initialize_local_device_time(NX_SNTP_TIME *NTPtime, ULONG ticks)
564  {
565
566      memcpy(&baseNTPtime, NTPtime, sizeof(NX_SNTP_TIME));
567
568      /* Set the local clock timer ticks. */
569      tx_time_set(ticks);
570
571      /* Now we can set the base_ticks variable used in the
572         get local time service. . */
573      base_ticks = tx_time_get();
574
575      return NX_SUCCESS;
576  }
```

Figure 1 Example of using SNTP Client with NetX

Below in Figure 2 is a modification of the example shown in Figure 1 above to demonstrate how to use the multi home interface feature of NetX. Inserted below line 45-49 where NetX and ThreadX resource variables are created, the host's primary and secondary interface IP addresses are defined, as well as the host IP gateway address (optional) and server list of time servers. Notice that these are real IP addresses, and not the simulator IP addresses used by for the NetX ram driver demo, for purposes of demonstration.

After the Client IP instance is created in lines 94-104 using the primary client interface address, the second host interface is 'attached' to the main IP control block with the secondary address and in this case the same network driver in lines 112-119.

Lastly, once the client thread is running, the Client IP gateway is set at the top of the *demo_client_thread_entry* function in lines 200-204. This last step is **only** necessary if any of the

host's time servers are located on an off link network address and all packets must go through the host gateway to reach them.

At this point the IP task will be able to figure out which interface to send out packets to regardless if the host client connects to its time server through the primary or secondary interface. See the NetX User Guide for more specific information *on nx_ip_interface_attach* and *nx_ip_gateway_address_set.*

```
43   /* Set up client thread and network resources. */
44
45   NX_PACKET_POOL      client_packet_pool;
46   NX_IP               client_ip;
47   NX_UDP_SOCKET       client_socket;
48   TX_THREAD           demo_client_thread;
49   NX_SNTP_CLIENT      demo_client;
50
51   #define SERVER_IP_ADDRESS        "64.125.78.85  192.2.2.92"
52   #define CLIENT_PRIMARY_ADDRESS    IP_ADDRESS(192,68,1,10)
53   #define CLIENT_SECONDARY_ADDRESS IP_ADDRESS( 64,125,78,85)
54   #define GATEWAY_IP_ADDRESS       IP_ADDRESS(192,68,1,1)
55
56   #define MULTI_HOMED_DEVICE        1
```

…

```
93       /* Create Client IP instances */
94       status = nx_ip_create(&client_ip, "SNTP IP Instance", NX_SNTP_CLIENT_IP_ADDRESS,
95                             0xFFFFFF00UL, &client_packet_pool,  nx_etherDriver_mcf5272,
96                             free_memory_pointer, NX_SNTP_CLIENT_IP_STACK_SIZE,
97                             NX_SNTP_CLIENT_IP_THREAD_PRIORITY);
98
99       /* Check for error. */
100      if (status != NX_SUCCESS)
101      {
102
103          NX_SNTP_CLIENT_EVENT_LOG(SEVERE, ("Error creating IP instance. Status: 0x%x\n\r", status));
105          return;
106      }
107
108      free_memory_pointer =  free_memory_pointer + NX_SNTP_CLIENT_IP_STACK_SIZE;
109
110      #ifdef MULTI_HOMED_DEVICE
111      /* Create the second Client Interface.  */
112      status =  _nx_ip_interface_attach(&client_ip, "port_2", CLIENT_SECONDARY_ADDRESS,
113                  0xFFFFFF00UL, nx_etherDriver_mcf5485);
114
115      /* Check for IP attach errors.  */
116      if (status)
117      {
118          return;
119      }
120      #endif
```

…

```
188  /* Define the client thread.  */
189  void    demo_client_thread_entry(ULONG info)
190  {
191
192  UINT                status;
193  NX_SNTP_CLIENT      *client_ptr;
194
195
196      client_ptr = (NX_SNTP_CLIENT *)info;
197
198
199    /* For each off link SNTP server IP address, a next hop (e.g. gateway) must be established. */
```

```
200    status = nx_ip_gateway_address_set(client_ptr -> ip_ptr, GATEWAY_IP_ADDRESS);
201    if (status)
202    {
203
204        return;
205    }
...
```

Figure 2 Example of using a multi homed SNTP Client host with NetX (5.3 or later)

Configuration Options

There are several configuration options for defining the NetX SNTP Client. The following list describes each in detail:

Define	Meaning
NX_DISABLE_ERROR_CHECKING	Defined, this option removes the basic SNTP error checking. It is typically used after the application has been debugged.
NX_SNTP_CLIENT_DEBUG	This option sets the level of SNTP Client event logging, from logging ALL messages, to only logging SEVERE errors. To disable logging, set level to NONE. The default NetX SNTP Client level is set to MODERATE.
NX_SNTP_CLIENT_STACK_SIZE	This option sets the size of the Client thread stack. The default NetX SNTP Client size is 2048.
NX_SNTP_CLIENT_THREAD_TIME_SLICE	This option sets the time slice of the scheduler allows for Client thread execution. The default NetX SNTP Client size is TX_NO_TIME_SLICE.
NX_SNTP_CLIENT_ THREAD_PRIORITY	This option sets the Client thread priority. The NetX SNTP Client default value is 2.
NX_SNTP_CLIENT_PREEMPTION_THRESHOLD	This option sets the sets the level of priority at which the Client thread allows preemption. The default NetX SNTP Client value is set to NX_SNTP_CLIENT_ THREAD_PRIORITY.
NX_SNTP_CLIENT_IP_ADDRESS	

NX_SNTP_CLIENT_IP_STACK_SIZE This option sets the Client IP helper thread stack size. The default NetX SNTP Client size is 2048 bytes.

NX_SNTP_CLIENT_IP_THREAD_PRIORITY
This option sets Client IP helper thread priority. The default NetX SNTP Client value is NX_SNTP_CLIENT_THREAD_PRIORITY.

NX_SNTP_CLIENT_UDP_SOCKET_NAME
This option sets the UDP socket name. The NetX SNTP Client UDP socket name default is "SNTP Client socket."

NX_SNTP_CLIENT_UDP_PORT This sets the port which the Client socket will connect to the SNTP Server on. The default NetX SNTP Client is 123.

NX_SNTP_CLIENT_TIME_TO_LIVE Specifies the number of routers a Client packet can pass before it is discarded. The default NetX SNTP Client is set to 0x80.

NX_SNTP_CLIENT_MAX_QUEUE_DEPTH
Maximum number of UDP packets (datagrams) that can be queued in the NetX SNTP Client socket. Additional packets received mean the oldest packets are released. The default NetX SNTP Client is set to 5.

NX_SNTP_CLIENT_PACKET_HEADER_SIZE
This option sets aside the number of bytes of the packet size for Frame, IP, UDP and NetX header data. The default NetX SNTP Client is 60.

NX_SNTP_CLIENT_PACKET_SIZE Size of the UDP packet for sending time requests out. This includes UDP, IP, and Ethernet (Frame)

packet header data. The default NetX SNTP Client is 122 bytes.

NX_SNTP_CLIENT_PACKET_POOL_SIZE
Size of the SNTP Client packet pool. The NetX SNTP Client default is (10 * NX_SNTP_CLIENT_PACKET_SIZE).

NX_SNTP_CLIENT_PACKET_TIMEOUT Time out for NetX packet allocation. The default NetX SNTP Client packet timeout is 1 second.

NX_SNTP_CLIENT_ARP_CACHE_SIZE ARP cache memory size. Each ARP entry is 52 bytes, so the number of ARP entries is the memory size divided by 52. The default NetX SNTP Client ARP cache memory size is 1040 (20 entries).

NX_SNTP_CLIENT_SERVER_LIST_WAIT Client timeout to obtain the unicast and broadcast server list mutexes. The NetX SNTP Client server list mutex timeout default is 1 second.

NX_SNTP_CLIENT_MAX_SERVERS The maximum number of servers allowed on the Client active broadcast and unicast server list each. The NetX SNTP Client default is 6.

NX_SNTP_CLIENT_MAX_LOG_ENTRY Size of the buffer for displaying an NTP time, including optional title text. The default NetX SNTP Client is 50 bytes.

NX_SNTP_CLIENT_RUN_IN_TEST_MODE
This enables the NetX SNTP Client to run in ‘test’ mode, which means server time updates are not applied to the local device clock. The default NetX SNTP Client setting is disabled.

NX_SNTP_UDP_UNICAST_SERVER_ADDRESSES

Space delimited list of unicast server IP addresses in IP4 format which are required to initialize the NetX SNTP Client to run in unicast mode. A Client configured for manycast may leave this list empty. The default NetX SNTP Client setting is "192.2.2.35 192.2.2.100" which were test hosts in SNTP development.

NX_SNTP_CLIENT_MANYCAST_ADDRESS

IP address for a Client to send time requests to obtain a unicast time server using the manycast protocol. An empty string disables the Client from running in manycast mode, which is the default NetX SNTP Client setting.

NX_SNTP_CLIENT_BROADCAST_DOMAIN

IP address on the Client subnet for the Client operating in broadcast mode to listen for time updates from broadcast servers. The default NetX SNTP Client setting is "192.2.2.255" which was the test subnet domain.

NX_SNTP_UDP_BROADCAST_SERVER_ADDRESSES

Optional. Space delimited list of broadcast server IP addresses in IP4 format which are added to the active server list for the Client operating in broadcast mode. This list to verify incoming time broadcasts against known time servers on its own domain. The default NetX SNTP Client setting is "192.2.2.35 192.2.2.100" which were test hosts in SNTP development.

NX_SNTP_CLIENT_MULTICAST_ADDRESS

IP address for a Client to send time requests to obtain a broadcast time server address using the multicast

protocol. An empty string disables the Client from running in multicast mode, which is the default NetX SNTP Client setting.

NX_SNTP_CLIENT_NTP_VERSION SNTP version used by the Client The NetX SNTP Client API was based on Version 4.

NX_SNTP_CLIENT_MIN_NTP_VERSION Oldest SNTP version the Client will be able to work with. The NetX SNTP Client default is Version 3.

NX_SNTP_CLIENT_MIN_SERVER_STRATUM
The lowest level server (highest numeric stratum level) the Client will accept. The NetX SNTP Client default is 2.

NX_SNTP_CLIENT_MIN_TIME_ADJUSTMENT
The minimum time adjustment in milliseconds the Client will make to its local clock time. Time adjustments below this will be ignored. The NetX SNTP Client default is 10.

NX_SNTP_CLIENT_MAX_TIME_ADJUSTMENT
The maximum time adjustment in milliseconds the Client will make to its local clock time. For time adjustments above this amount, the local clock adjustment is limited to the maximum time adjustment. The NetX SNTP Client default is 180000 (3 minutes).

NX_SNTP_CLIENT_IGNORE_MAX_ADJUST_STARTUP
This enables the maximum time adjustment to be waived when the Client receives the first update from its time server. Thereafter, the maximum time adjustment is enforced. The intention is to get the Client in synch with the server clock

as soon as possible. The NetX SNTP Client default is enabled.

NX_SNTP_CLIENT_MAX_TIME_LAPSE Maximum amount of time the Client's host application is allowed to run without a valid time update. This value is based on the application's tolerance to local clock variation and rate of clock drift. The NetX SNTP Client default is 3 * NX_SNTP_CLIENT_UNICAST_SERVER_TIMEOUT (see below). This assumes unicast mode of operation and tolerates up to three servers to fail in succession before the NX_SNTP_CLIENT_MAX_TIME_LAPSE expires.

NX_SNTP_UPDATE_TIMEOUT_INTERVAL

The timer timeout (seconds) on the Client update timer. On every timer expiration, the update timer updates the time remaining on the Client *nx_sntp_update_time_remaining* decrementing it by the timer timeout interval. The NetX SNTP Client default is 1. The Client uses the *nx_sntp_update_time_remaining* field to know how much time is remaining, so a small value gives the Client task finer time resolution.

NX_SNTP_CLIENT_UNICAST_POLL_INTERVAL

The poll interval (seconds) on which the Client in unicast mode sends a time request to its time server. The NetX SNTP Client default is 3600.

NX_SNTP_CLIENT_UNICAST_SERVER_TIMEOUT

The maximum time interval (seconds) allowed without a valid time update received from the Client's unicast server before the Client switches to an alternative server. This value should be equal to or less than the NX_SNTP_CLIENT_MAX_TIME_LAPSE value. The NetX SNTP Client default is 3 *

NX_SNTP_CLIENT_UNICAST_POLL_INTERVAL to allow up to three consecutive missed server replies from its current server.

NX_SNTP_CLIENT_EXP_BACKOFF_RATE

The factor by which the unicast poll interval is increased. This is intended for the Client failing to receive a server time update, or receiving indications from the server that it is temporarily unavailable (e.g. not synchronized yet) for time update service. To disable this feature, set the back off rate to 1. The NetX SNTP Client default is 2.

NX_SNTP_CLIENT_BROADCAST_SERVER_TIMEOUT

The maximum time interval (seconds) allowed without a valid time update received from the Client's broadcast server before the Client needs switch to another server. This value should be less than the NX_SNTP_CLIENT_MAX_TIME_LAPSE to allow the Client to switch to another server on its subnet. The NetX SNTP Client default is 3600.

NX_SNTP_CLIENT_INITIAL_UNICAST_TIMEOUT

The receive time (seconds) for the Client in broadcast mode to wait for the server to reply to an initial unicast request. This value should be considerably less than the NX_SNTP_CLIENT_BROADCAST_SERVER_TIMEOUT Since the broadcast server may not be configured for unicast service and the Client should resume listening for broadcast time updates. The NetX SNTP Client default is 10.

NX_SNTP_CLIENT_MAX_ROOT_DISPERSION

The maximum server clock dispersion (microseconds), which is a measure of server clock precision, the Client will accept. To disable this requirement, set the maximum root

dispersion to zero. The NetX SNTP Client default is set to 500.

NX_SNTP_CLIENT_BAD_UPDATE_LIMIT

The limit on the number of consecutive bad updates received from the Client server in either broadcast or unicast mode. When this limit is reached, the Client stops accepting any more updates from the server and (should) switch to another server if possible. The limit should be set such that the NX_SNTP_CLIENT_MAX_TIME_LAPSE does not elapse while receiving bad updates. The NetX SNTP Client default is 3.

NX_SNTP_CURRENT_YEAR

This is set to the current year to display the local time in human readable format. The default value is zero.

Chapter 3

Description of NetX SNTP Client Services

This chapter contains a description of all NetX SNTP Client services (listed below) in alphabetic order.

In the “Return Values” section in the following API descriptions, values in **BOLD** are not affected by the **NX_DISABLE_ERROR_CHECKING** define that is used to disable API error checking, while non-bold values are completely disabled.

nx_sntp_client_add_server_IP_to_list
Add server IP address to active Client list

nx_sntp_client_add_servers_from_input_list
Add servers from input list to active Client list

nx_sntp_client_apply_sanity_checks
Apply sanity check to received time update

nx_sntp_client_calculate_roundtrip
Compute round trip time to server

nx_sntp_client_check_server_clock_dispersion
Check server clock dispersion for precision

nx_sntp_client_create
Create the SNTP Client

nx_sntp_client_delete
Delete the SNTP Client

nx_sntp_client_find_server_in_list
Find server in active Client list

nx_sntp_client_get_next_server
Get next server in active Client list

nx_sntp_client_get_server_roundtrip
Request and process time update for round trip time

nx_sntp_client_initialize_broadcast
Initialize Client for broadcast operation

nx_sntp_client_initialize_unicast
Initialize Client for unicast operation

nx_sntp_client_process_time_data
Apply server clock time to local Client time

nx_sntp_client_receive_time_update
Process packets received on Client socket

nx_sntp_client_remove_server_from_list
Remove server from active Client list

nx_sntp_client_reset_broadcast
Reinitialize Client to resume broadcast operation

nx_sntp_client_reset_unicast
Reinitialize Client to resume unicast operation

nx_sntp_client_run_broadcast
Receive time updates from server

nx_sntp_client_run_unicast
Send requests and receive time updates from server

nx_sntp_client_send_unicast_request
Send time request to an NTP Server

nx_sntp_client_utility_add_msecs_to_NTPtime
Add milliseconds to NTP time fraction

nx_sntp_client_utility_add_NTP time
Add two NTP times

nx_sntp_client_utility_convert_fraction_to_msecs
Convert NTP time fraction to milliseconds

nx_sntp_client_utility_convert_LONG_to_IP
Convert IP address from LONG to string

nx_sntp_client_utility_convert_refID_KOD_code
Convert a Reference ID to Kiss of Death code

nx_sntp_client_utility_display_NTP_time

Display NTP time in seconds

nx_sntp_client_utility_get_msec_diff
Get difference in milliseconds of two NTP times

nx_sntp_client_add_server_to_list

Add server to active Client server list

Prototype

```
UINT  nx_sntp_client_add_server_to_list(NX_SNTP_CLIENT *client_ptr,
                      UINT operating_mode, ULONG server_to_add);
```

Description

This service adds the input server IP to the Client's active server list (e.g. list of unicast or broadcast servers depending on Client's current mode of operation). If the server is already in the list, the service still returns successful completion status, no changes are made to the list.

Input Parameters

client_ptr	Pointer to SNTP Client control block
operating_mode	Client mode of operation (e.g. unicast)
server_to_add	IP address of server to add

Return Values

NX_SUCCESS	(0x00)	Successful SNTP Client deletion
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_SNTP_INVALID_IP_ADDRESS	(0xD10)	Invalid server IP address

Allowed From

Threads

Example

```
/* Add the server to the active Client list.  */
```

```
status =  nx_sntp_client_add_server_in_list(client_ptr, operating_mode,
                                                        server_to_add);

/* If status return NX_SUCCESS, the server was successfully added.*/
```

See Also

nx_sntp_client_find_server_in_list,
nx_sntp_client_add_servers_from_input_list,
nx_sntp_client_remove_from_list, nx_sntp_client_get_next_server,
nx_sntp_client_initialize_broadcast, nx_sntp_client_initialize_unicast

nx_sntp_client_add_servers_from_input_list

Add servers to active Client server list from an input list

Prototype

```
UINT  nx_sntp_client_add_servers_from_input_list(
                        NX_SNTP_CLIENT *client_ptr, UINT
                        operating_mode, CHAR *input_server_list);
```

Description

This service adds the server IP address from the input list to the Client's active server list (e.g. list of unicast or broadcast servers depending on Client's current mode of operation). The address is first converted from the string IP address to a ULONG data and the active list searched for the server already in the list. If a server is already in the list, the service continues to the next server to add without adding a duplicate server address entry. If at the end of the list the Client has no servers in its active list, and the Client intends to operate in unicast mode, the service returns an error.

Input Parameters

client_ptr	Pointer to SNTP Client control block
operating_mode	Client mode of operation (e.g. unicast)
input_server_list	List of server IP addresses to add

Return Values

NX_SUCCESS	(0x00)	Successful SNTP Client deletion
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_SNTP_NO_AVAILABLE_SERVERS	(0xD04)	No valid servers in Client list

Allowed From

Threads

Example

```
/* Add the servers from the specified list to the active Client list.  */
status =  nx_sntp_client_add_servers_from_input_list(client_ptr, operating_mode,
                                                     server_list);


/* If status return NX_SUCCESS, the server list was successfully added.*/
```

See Also

nx_sntp_client_add_server_IP_to_list, nx_sntp_client_remove_from_list, nx_sntp_client_get_next_server, nx_sntp_client_find_server_in_list, nx_sntp_client_initialize_broadcast, nx_sntp_client_initialize_unicast

nx_sntp_client_apply_sanity_checks

Apply sanity checks to a server time update

Prototype

```
UINT  nx_sntp_client_apply_sanity_checks(
                        NX_SNTP_CLIENT *client_ptr,
                        NX_SNTP_TIME_MESSAGE *client_time_msg_ptr,
                        NX_SNTP_TIME_MESSAGE *server_time_msg_ptr);
```

Description

This service applies a set of sanity checks recommended by the RFC 4330 for SNTP protocol to verify that a received server time update contains valid data. In unicast mode, the service requires both the Client send request as well as the server time update to match certain fields between these two NTP time messages.

The NetX SNTP Client API allows for the host application to define an additional user-defined set of sanity checks appropriate for the host environment. If the Client is configured with a user-defined set of sanity check callback, this service will call that service after completing its own set of checks.

Input Parameters

client_ptr	Pointer to SNTP Client control block
client_time_msg_ptr	Pointer to Client send request
server_time_msg_ptr	Pointer to server reply

Return Values

NX_SUCCESS	(0x00)	Successful SNTP Client deletion
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_SNTP_DUPE_SERVER_PACKET	(0xD14)	Duplicate server packet received
NX_SNTP_INVALID_SERVER_MODE		

(0xD0E) Invalid server mode for Client

NX_SNTP_INVALID_NTP_VERSION
(0xD0D) Invalid server NTP version

NX_SNTP_SERVER_CLOCK_NOT_SYNC
(0xD09) Server clock not synchronized

NX_SNTP_INVALID_SERVER_STRATUM
(0xD11) Server stratum not acceptable

NX_SNTP_INVALID_TIMESTAMP
(0xD18) Null or out of range server timestamp

Allowed From

Threads

Example

```
/* Apply sanity checks on the server update the Client has just received in
response to the Client sending the current_time_message_request. */
status =  nx_sntp_client_apply_sanity_checks(client_ptr,
               &(client_ptr -> current_time_message_request),
               &(client_ptr -> current_server_time_message));


/* If status return NX_SUCCESS, the server update passed all sanity checks.*/
```

See Also

nx_sntp_client_duplicate_update_check, nx_sntp_client_utility_get_msec_diff, nx_sntp_client_process_update_packet, nx_sntp_client_run_broadcast

nx_sntp_client_calculate_roundtrip

Calculate round trip time to server

Prototype

```
UINT  nx_sntp_client_calculate_roundtrip(
                        NX_SNTP_TIME *time_server_update_received,
                        NX_SNTP_TIME_MESSAGE *server_time_message,
                        LONG *roundtrip_time, LONG *offset_time);
```

Description

This service computes the round trip time to the server and the offset from the server clock to the Client clock using the formula specified in RFC 4330 for SNTP. Server offset is not used in the NetX SNTP Client API. Offset time can be a negative value, but round trip time must be zero or positive. Note this feature is only offered for NetX SNTP Clients operating in Unicast mode.

Input Parameters

time_server_update_received	Pointer to time update was received
server_time_message	Pointer to server time update
roundtrip_time	Pointer to computed round trip time
offset_time	Pointer to computed offset

Return Values

NX_SUCCESS	(0x00)	Successful SNTP Client deletion
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_INVALID_LOCAL_TIME	(0xD05)	Invalid time of update received
NX_SNTP_INVALID_RTT_TIME	(0xD19)	Invalid round trip time computed

Allowed From

Threads

Example

```
/* Calculate round trip time to server and save the computation to Client round
trip time and system (server) offset fields. */
status =  nx_sntp_client_calculate_roundtrip(
                                    &(client_ptr -> server_update_time),
                                    &(client_ptr -> current_server_time_message),
                                    &(client_ptr -> roundtrip_time_msec),
                                    &(client_ptr -> system_clock_offset_msec));

/* If status return NX_SUCCESS, a valid round trip time was calculated.*/
```

See Also

nx_sntp_client_utility_get_msec_diff, nx_sntp_client_process_update_packet nx_sntp_client_run_broadcast

nx_sntp_client_check_server_clock_dispersion

Check the reported server clock dispersion

Prototype

```
UINT  nx_sntp_client_check_server_clock_dispersion(
                    NX_SNTP_CLIENT *client_ptr,
                    ULONG server_clock_dispersion, UINT *dispersion_ok);
```

Description

This service converts the root dispersion field in the server time update, expressed as an exponent of two, to microseconds and compares that value against the Client *max_root_dispersion* parameter to verify the server clock has the required precision for the Client. If the server does not report its root dispersion, (which it is not required to do), the service routine returns successful completion. To disable this check, set the Client *max_root_dispersion* to NULL.

Input Parameters

client_ptr	Pointer to SNTP Client control block
server_clock_dispersion	NTP server clock dispersion
dispersion_ok	Pointer to flag indicating dispersion is ok

Return Values

NX_SUCCESS	(0x00)	Successful SNTP Client deletion
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input

Allowed From

Threads

Example

```
/* Check server root (clock) dispersion against the Client's required precision
and return the result in the dispersion_ok flag. */
status =  nx_sntp_client_check_server_clock_dispersion(client_ptr,
            client_ptr -> current_server_time_message.clock_dispersion,
            &dispersion_ok));

/* If status return NX_SUCCESS, server update either reported no root dispersion
or the non zero server root dispersion is below the Client's maximum root
dispersion accepted.*/
```

See Also

nx_sntp_client_process_update_packet

nx_sntp_client_create

Create an SNTP Client

Prototype

```
UINT nx_sntp_client_create(NX_SNTP_CLIENT *client_ptr, NX_IP *ip_ptr,
      NX_PACKET_POOL *packet_pool_ptr, ULONG time_to_live,
      UINT max_queue_depth, UINT port, UINT operating_mode,
      ULONG min_time_adjustment, ULONG max_time_adjustment, UINT
      exponential_backoff_rate, ULONG max_timelapse_without_update,
      UINT invalid_time_update_limit, ULONG max_root_dispersion,
      UINT ignore_max_adjust_on_startup, UINT test_mode,
      UINT (*get_local_device_time)(NX_SNTP_TIME *time_ptr),
      UINT (*set_local_device_time)(NX_SNTP_TIME *time_ptr),
      UINT (*apply_custom_sanity_checks)(NX_SNTP_CLIENT *client_ptr,
         NX_SNTP_TIME_MESSAGE *client_time_msg_ptr,
         NX_SNTP_TIME_MESSAGE *server_time_msg_ptr),
      UINT (*adjust_local_device_time)(LONG msecs),
      UINT (*leap_second_handler)(NX_SNTP_CLIENT *client_ptr, UINT
         indicator),
      UINT (*kiss_of_death_handler)(NX_SNTP_CLIENT *client_ptr,
         NX_SNTP_TIME_MESSAGE *server_time_msg),
      VOID (random_number_generator)(struct NX_SNTP_CLIENT_STRUCT
         *client_ptr, ULONG *rand);
```

Description

This service creates an SNTP Client instance.

Input Parameters

client_ptr	Pointer to SNTP Client control block
ip_ptr	Pointer to Client IP instance
packet_pool_ptr	Pointer to Client packet pool
time_to_live	Client UDP Packet time to live
max_queue_depth	Max number of packets in Client socket queue
port	NTP Server port
operating_mode	Client association with Server
min_time_adjustment	Minimum adjustment to local time
max_time_adjustment	Maximum adjustment to local time

exponential_backoff_rate	Poll interval rate of increase
max_timelapse_without_update	Maximum length of time allowed without a valid server update
invalid_time_update_limit	Limit on consecutive bad updates
max_root_dispersion	Maximum acceptable server dispersion
ignore_max_adjust_on_startup	No maximum limit on time adjustment on first update
test_mode	Enable test mode (server updates not applied to local time)
get_local_device_time	Callback for getting local time
set_local_device_time	Callback for setting local time
adjust_local_device_time	Callback for adjusting local time ahead or back
leap_second_handler	Callback for application response to impending leap second
kiss_of_death_handler	Callback for application response to receiving Kiss of Death packet
random_number_generator	Callback to random number generator service

Return Values

NX_SUCCESS	(0x00)	Successful Client creation
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input

Allowed From

Threads, Application code

Example

```
/* Create the SNTP Client. */
status =  nx_sntp_client_create(&demo_client, &client_ip, &client_packet_pool,
             NX_SNTP_CLIENT_TIME_TO_LIVE, NX_SNTP_CLIENT_MAX_QUEUE_DEPTH,
             NX_SNTP_CLIENT_UDP_PORT, BROADCAST_MODE,
             NX_SNTP_CLIENT_MIN_TIME_ADJUSTMENT,
             NX_SNTP_CLIENT_MAX_TIME_ADJUSTMENT, NX_SNTP_CLIENT_EXP_BACKOFF_RATE,
             NX_SNTP_CLIENT_MAX_TIME_LAPSE, NX_SNTP_CLIENT_BAD_UPDATE_LIMIT,
             NX_SNTP_CLIENT_MAX_ROOT_DISPERSION,
             NX_SNTP_CLIENT_IGNORE_MAX_ADJUST_STARTUP,
             NX_SNTP_CLIENT_RUN_IN_TEST_MODE,
             get_local_device_time, set_local_device_time,
             NULL /* No apply_custom_sanity_checks callback */,
             NULL /* no adjust_local_device_time callback */,
             leap_second_handler, kiss_of_death_handler,
             NULL /* no random_number_generator callback */);

/* If status is NX_SUCCESS an SNTP Client instance was successfully
   create.  */
```

See Also

nx_sntp_client_delete

nx_sntp_client_find_server_in_list

Find server in active Client server list

Prototype

```
UINT  nx_sntp_client_find_server_in_list(NX_SNTP_CLIENT *client_ptr,
            UINT operating_mode, ULONG server_to_find, INT *index);
```

Description

This service finds the input server IP in the Client's active server list (e.g. list of unicast or broadcast servers depending on Client's current mode of operation) and returns its location in the list. If not found, the service still returns successful completion status, but the index is set to -1.

Input Parameters

client_ptr	Pointer to SNTP Client control block
operating_mode	Client mode of operation (e.g. unicast)
server_to_find	IP address of server to find
index	Pointer to index into list where server found (or -1 if not found in list)

Return Values

NX_SUCCESS	(0x00)	Successful SNTP Client deletion
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Threads

Example

```
/* Find the server in the active Client list.  */
status =  nx_sntp_client_find_server_in_list(client_ptr, operating_mode,
                                             server_to_add, &list_index);
```

```
/* If index is non-negative the server was found in the active SNTP Client
list.*/
```

See Also

nx_sntp_client_add_server_IP_to_list,
nx_sntp_client_add_servers_from_input_list,
nx_sntp_client_remove_from_list,
nx_sntp_client_get_next_server, nx_sntp_client_initialize_broadcast,
nx_sntp_client_initialize_unicast

nx_sntp_client_get_next_server

Get next server in Client active server list

Prototype

```
UINT nx_sntp_client_get_next_server(NX_SNTP_CLIENT *client_ptr,
            UINT operating_mode, UINT *search_index, UINT wrap);
```

Description

This service retrieves the next server IP in the Client server list (e.g. unicast or broadcast mode). The search starts at the search_index location in the list till a server IP not equal to the Client current server is found. The wrap parameter enables the search to continue at the top of the list if a server has not been found yet. If a server is found, the search index is incremented by one before returning to the caller.

Input Parameters

client_ptr	Pointer to SNTP Client control block
operating_mode	Client mode of operation (e.g. unicast)
server_index	Pointer to index location where to start search
wrap	Enable continuing search back to the top of the list

Return Values

NX_SUCCESS	(0x00)	Client server available
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_ERROR_NO_AVAILABLE_SERVERS	(0xD0D)	Server list empty or no other servers in list

Allowed From

Threads

Example

```
/* Get the next server from Client active server list. */
status =  nx_sntp_client_get_next_server client_ptr,
                client_ptr -> operating_mode, &search_index, NX_TRUE);

/* If status is NX_SUCCESS a server was successfully located in Client list. */
```

See Also

nx_sntp_client_add_server_IP_to_list,
nx_sntp_client_add_servers_from_input_list,
nx_sntp_client_find_server_in_list, nx_sntp_client_remove_from_list,
nx_sntp_client_initialize_broadcast, nx_sntp_client_initialize_unicast

nx_sntp_client_get_server_roundtrip

Compute round trip time from Client to server

Prototype

```
UINT nx_sntp_client_get_server_roundtrip(NX_SNTP_CLIENT *client_ptr,
            ULONG initial_uni_timeout, UINT accept_incoming_address);
```

Description

This service is used by the Client in broadcast mode to send a unicast request to the first server it receives a time broadcast. If the server replies, the Client can calculate round trip time to the server (and offset from the server clock) before establishing this server as its broadcast time server. The service checks for a sensible round trip time is computed (e.g. greater than zero). Note this feature is only offered for NetX SNTP Clients operating in Unicast mode.

Input Parameters

client_ptr Pointer to SNTP Client control block

initial_uni_timeout Wait option for receiving server reply

accept_incoming_address
Automatically accept server IP in lieu of Client having established its current server yet

Return Values

NX_SUCCESS	(0x00)	Round trip time computed
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_SNTP_NO_UNICAST_FROM_SERVER	(0xD08)	No reply from server

Allowed From

Threads

Example

```
/* Attempt to get the server round trip time but only from the Client's current
    server. */
status =  nx_sntp_client_get_server_roundtip(&client_ptr,
                                    request_timeout, NX_FALSE);

/* If status is NX_SUCCESS if the server responded with valid time update and a
sensible round trip time was computed.  */
```

See Also

nx_sntp_client_send_unicast_request, nx_sntp_client_receive_time_update, nx_sntp_client_calculate_roundtrip, nx_sntp_client_process_update_time

nx_sntp_client_initialize_broadcast

Initialize the Client for broadcast operation

Prototype

```
UINT nx_sntp_client_initialize_broadcast(NX_SNTP_CLIENT *client_ptr,
            ULONG broadcast_timeout, ULONG initial_unicast_timeout,
            UINT send_initial_unicast, UINT client_requires_rtt, CHAR
            *broadcast_domain, CHAR *multicast_server_address,
            CHAR * broadcast_time_servers);
```

Description

This service initializes the Client for broadcast operation by setting up the necessary domain IP address and server time out, plus optional server input addresses (or multicast IP address for multicast discovery of broadcast server).

Input Parameters

client_ptr	Pointer to SNTP Client control block
broadcast_server_timeout	Timeout to receive next server broadcast
initial_unicast_timeout	Timeout to receive reply to unicast request
send_initial_unicast	Enable sending a unicast request after when update received
client_requires_rtt	Client requires a valid round trip time to process server time updates
broadcast_domain	IP address of Client broadcast subnet
multicast_server_address	IP address of Client multicast network
broadcast_time_servers	Starting list of known broadcast servers

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized

NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_INVALID_DOMAIN	(0xD03)	Invalid domain IP input

Allowed From

Threads

Example

```
/* Initialize the client for broadcast operation.  */
status =  nx_sntp_client_initialize_broadcast(client_ptr,
       NX_SNTP_CLIENT_BROADCAST_SERVER_TIMEOUT,
       NX_SNTP_CLIENT_INITIAL_UNICAST_TIMEOUT, NX_TRUE, NX_FALSE,
       NX_SNTP_CLIENT_BROADCAST_DOMAIN, NX_SNTP_CLIENT_MULTICAST_ADDRESS,
        NX_SNTP_UDP_BROADCAST_SERVER_ADDRESSES);

/* If status is NX_SUCCESS the Client was successfully initialized.  */
```

See Also

nx_sntp_client_reset_broadcast, nx_sntp_client_run_broadcast, nx_sntp_client_initialize_unicast, nx_sntp_client_reset_unicast, nx_sntp_client_run_unicast

nx_sntp_client_initialize_unicast

Delete directory on SNTP Server

Prototype

```
UINT nx_sntp_client_initialize_unicast(NX_SNTP_CLIENT * client_ptr,
       ULONG unicast_server_timeout, ULONG unicast_poll_interval,
       UINT randomize_wait_on_startup, UINT client_requires_rtt,
       CHAR * manycast_server_address, CHAR *unicast_time_servers);
```

Description

This service initializes the Client for unicast operation by setting up the Client's list of trusted unicast servers (or manycast IP address for manycast discovery of unicast servers), as well as unicast server time out and polling interval. The Client can be configured to wait for a random number of ticks if the host application has provided a random number generator, and to require a round trip time to server calculation be computed before updating local time.

Input Parameters

client_ptr	Pointer to SNTP Client control block
unicast_server_timeout	Timeout to receive next server unicast
unicast_poll_timeout	Wait interval between unicast requests
randomize_wait_on_startup	Enable random wait before first request
client_requires_rtt	Client requires a valid round trip time to process server time updates
manycast_server_address	IP address of Client manycast network
unicast_time_servers	Starting list of known unicast servers

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input

NX_SNTP_ERROR_NO_AVAILABLE_SERVERS
(0xD0D) No valid unicast servers in input list

Allowed From

Threads

Example

```
/* Initialize the Client for unicast operation.  */
status =  nx_sntp_client_initialize_unicast(&client_ptr,
     NX_SNTP_CLIENT_UNICAST_SERVER_TIMEOUT,
     NX_SNTP_CLIENT_UNICAST_POLL_INTERVAL, NX_TRUE, NX_FALSE,
     NX_SNTP_CLIENT_MANYCAST_ADDRESS, NX_SNTP_UDP_UNICAST_SERVER_ADDRESSES);

/* If status is NX_SUCCESS the Client is initialized for unicast operation.  */
```

See Also

nx_sntp_client_initialize_unicast, nx_sntp_client_reset_unicast, nx_sntp_client_run_unicast, nx_sntp_client_reset_broadcast, nx_sntp_client_run_broadcast

nx_sntp_client_process_time_data

Apply server time update to local time clock

Prototype

```
UINT nx_sntp_client_process_time_data(NX_SNTP_CLIENT *SNTP_client_ptr,
                  NX_SNTP_TIME_MESSAGE *server_time_message,
                  UINT first_update);
```

Description

This service processes time data received from the server (after the packet has been inspected and passed sanity checks) and determines if and how much to update local time clock based on the server clock time. The time difference with the server clock is not applied to local time if 1) it is below the minimum time adjustment, or 2) above the maximum time adjustment and it is not the first update from the server. In the latter case, the time adjustment is limited to the maximum time adjustment.

Input Parameters

client_ptr	Pointer to SNTP Client control block.
server_time_message	Pointer to server time update
first_update	Indicates if the time update is the first one received from the server

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_INVALID_LOCAL_TIME	(0xD05)	Time server was received was not recorded
NX_SNTP_BAD_SERVER_ROOT_DISPERSION	(0xD12)	Time server clock dispersion exceeds Client maximum dispersion setting

Allowed From

Threads

Example

```
/* Process the supplied time update received from the server as the first update
    from   the current server. */
  status =  nx_sntp_client_process_time_update(client_ptr,
              client_ptr -> current_server_time_message), first_update_pending);

/* If status is NX_SUCCESS, the client local clock time is updated to the server
    time.  */
```

See Also

nx_sntp_client_send_unicast_request, nx_sntp_client_receive_time_update, nx_sntp_client_calculate_roundtrip, nx_sntp_client_get_server_roundtrip

nx_sntp_client_receive_time_update

Process update packets received on Client socket

Prototype

```
UINT nx_sntp_client_receive_time_udpate(NX_SNTP_CLIENT *client_ptr,
            NX_SNTP_TIME_MESSAGE *time_message, ULONG timeout)
```

Description

This service receives packets on the Client socket and inspects them for valid UDP port, sender IP address and length. When a packet is received, the Client records the local time immediately to have a reference time point for updating the local clock with the server clock. The packet data is extracted into a server time message and returned to the caller before this service releases the packet back to the Client packet pool.

Input Parameters

client_ptr	Pointer to SNTP Client control block.
time_message	Pointer to server time update
timeout	Wait option to receive the update
accept_incoming_address	Accept the sender IP as the Client's current server

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_INVALID_SERVER_PORT	(0xD0F)	IP header sender's destination port is not what Client expects
NX_SNTP_INVALID_IP_ADDRESS	(0xD10)	IP header sender's source IP address is invalid

NX_SNTP_INVALID_LOCAL_TIME
(0xD05) Client is unable to obtain local time

NX_SNTP_INVALID_TIME_PACKET
(0xD0C) Update packet length is incorrect for an NTP time message

Allowed From

Threads

Example

```
/* Receive packets on the Client UDP socket, and inspect the packet header and
length for valid NTP time packet, and accept the sender IP e.g. manycast is
enabled.  */
status =  nx_sntp_client_receive_time_update(client_ptr,
              client_ptr -> current_server_time_message, receive_tieout, NX_TRUE);


/* If status is NX_SUCCESS, the Client has received a valid NTP time packet.  */
```

See Also

nx_sntp_client_process_update_packet,
nx_sntp_client_extract_time_message_from_packet,
nx_sntp_client_add_server_ULONG_to_list,
nx_sntp_client_find_server_in_list, nx_sntp_client_utility_server_in_domain,
nx_sntp_client_process_update_packet

nx_sntp_client_remove_server_from_list

Remove server from Client's active server list

Prototype

```
UINT nx_sntp_client_remove_server_from_list(NX_SNTP_CLIENT client_ptr,
                    UINT operating_mode, ULONG server_to_remove);
```

Description

This service removes the specified server from the Client's active server list. Active server list is indicated by the operating mode (e.g. unicast or broadcast). First the active list is searched for the server to remove. If not found, the service returns successful completion with no changes to the active list. Before any changes are made to the list, the service obtains an exclusive lock on the list, and releases the lock after the changes are made.

Input Parameters

client_ptr	Pointer to SNTP Client control block.
operating_mode	Client mode of operation (e.g. unicast)
server_to_remove	Server (IP address) to remove from list

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Threads

Example

```
/* Remove 'server_to_remove' from the Client's active server list. */
```

```
status =  nx_sntp_client_remove_server_from_list(client_ptr,
              client_ptr -> operating_mode, server_to_remove);


/* If status is NX_SUCCESS, server_to_remove has been removed from Client active
server list.  */
```

See Also

nx_sntp_client_add_server_IP_to_list,
nx_sntp_client_add_servers_from_input_list,
nx_sntp_client_find_server_in_list, nx_sntp_client_get_next_server

nx_sntp_client_reset_broadcast

Reinitialize the Client for resuming broadcast operation

Prototype

```
UINT nx_sntp_client_reset_broadcast(NX_SNTP_CLIENT *client_ptr);
```

Description

This service deactivates the Client update timer and clears the current server so the Client is ready to resume broadcast (or multicast) operation. It does not reset the time remaining on the Client *max_timelapse_without_update* because to do so requires receiving a valid server update.

Input Parameters

client_ptr	Pointer to SNTP Client control block.

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Threads

Example

```
/* Reinitialize the Client for broadcast operation. */
status =  nx_sntp_client_reset_broadcast(client_ptr);

/* If status is NX_SUCCESS, the Client pointed to by client_ptr is ready to
resume broadcast operation.  */
```

See Also

nx_sntp_client_initialize_broadcast, nx_sntp_client_run_broadcast, nx_sntp_client_initialize_unicast, nx_sntp_client_reset_unicast, nx_sntp_client_run_unicast

nx_sntp_client_reset_unicast

Reinitialize the Client for resuming unicast operation

Prototype

```
UINT nx_sntp_client_reset_unicast(NX_SNTP_CLIENT *client_ptr);
```

Description

This service deactivates the Client update timer and clears the current server so the Client is ready to resume unicast (or multicast) operation. It does not reset the time remaining on the Client *max_timelapse_without_update* because to do so requires receiving a valid server update.

Input Parameters

client_ptr	Pointer to SNTP Client control block.

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Threads

Example

```
/* Reinitialize the Client for unicast operation. */
status =  nx_sntp_client_reset_unicast(client_ptr);

/* If status is NX_SUCCESS, the Client pointed to by client_ptr is ready to
resume unicast operation.  */
```

See Also

nx_sntp_client_initialize_unicast, nx_sntp_client_run_unicast,
nx_sntp_client_initialize_broadcast, nx_sntp_client_reset_broadcast,
nx_sntp_client_run_broadcast

nx_sntp_client_run_broadcast

Run the Client in broadcast mode

Prototype

```
UINT nx_sntp_client_run_broadcast(NX_SNTP_CLIENT *client_ptr);
```

Description

This service runs the Client in broadcast mode. It waits to receive broadcasts from time servers on its specified domain. It keeps track of time remaining on both the broadcast server time out and the Client's maximum lapse time out. If the server time out expires before the maximum lapse time out, the Client exits this service in order to attempt to find another broadcast server on its subnet (or responding multicast server if multicast enabled).

Received time updates are checked for valid packet and NTP time data before being applied to the local clock time. Certain failed data checks will force the Client to terminate receiving updates from the server, while others simply indicate to reject the packet and continue waiting for a valid packet. Whether the server is removed from the active list rather than just switching to an alternate server is up to the Client host application.

Input Parameters

client_ptr	Pointer to SNTP Client control block.

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_CLIENT_NOT_INITIALIZED	(0xD01)	Client not initialized
NX_SNTP_INVALID_LOCAL_TIME		

	(0xD05)	Unable to obtain local time
NX_SNTP_OVER_BAD_UPDATE_LIMIT		
	(0xD13)	Limit on number of consecutive bad updates from the same server is reached
NX_SNTP_TIMED_OUT_ON_SERVER		
	(0xD16)	Broadcast server time out expired without a valid time update received
NX_SNTP_MAX_TIME_LAPSE_EXCEEDED		
	(0xD17)	Client max lapse time out expired without a valid time update received

Allowed From

Threads

Example

```
/* Start Client running in broadcast mode, receiving and processing broadcast
updates from time servers on its network domain.  */
status =  nx_sntp_client_run_broadcast(client_ptr);

/* This service runs as an infinite loop until an error or exception requires
    the Client to abort broadcast updates. Status indicates the nature of the
    error.  */
```

See Also

nx_sntp_client_initialize_broadcast, nx_sntp_client_reset_broadcast, nx_sntp_client_initialize_unicast, nx_sntp_client_reset_unicast, nx_sntp_client_run_unicast, nx_sntp_client_reset_current_time_message, nx_sntp_client_receive_time_update, nx_sntp_client_apply_sanity_checks, nx_sntp_client_get_server_roundtrip

nx_sntp_client_run_unicast

Run the Client in unicast mode

Prototype

```
UINT nx_sntp_client_run_unicast(NX_SNTP_CLIENT *client_ptr);
```

Description

This service runs the Client in unicast mode. While sending time requests and receiving unicast updates from its time server, it keeps track of time remaining on both the unicast server time out and the Client's maximum lapse time out. If the server times out expires before the maximum lapse time out, the Client exits this service in order to attempt to find another unicast server in its active server list (or invokes the manycast IP protocol if manycast is enabled).

Received time updates are checked for valid packet and NTP time data before being applied to the local clock time. Regardless if a received packet contains a valid time update, the Client will not poll the server for another update until the poll interval has expired. Certain failed data checks will force the Client to terminate receiving updates from the server, while others simply indicate to reject the packet and continue waiting for a valid packet. Whether the server is removed from the active list rather than just switching to an alternate server is up to the Client host application.

Input Parameters

client_ptr	Pointer to SNTP Client control block.

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_CLIENT_NOT_INITIALIZED	(0xD01)	Client not initialized

NX_SNTP_INVALID_LOCAL_TIME
(0xD05) Unable to obtain local time

NX_SNTP_OVER_BAD_UPDATE_LIMIT
(0xD13) Limit on number of consecutive bad updates from the same server is reached

NX_SNTP_TIMED_OUT_ON_SERVER
(0xD16) Unicast server time out expired without a valid time update received

NX_SNTP_MAX_TIME_LAPSE_EXCEEDED
(0xD17) Client max lapse time out expired without a valid time update received

Allowed From

Threads

Example

```
/* Start Client running in unicast mode, sending and receiving unicast updates
from its time server.  */
status =  nx_sntp_client_run_unicast(client_ptr);

/* This service runs as an infinite loop until an error or exception requires
    the Client to abort unicast updates. Status indicates the nature of the
    error.  */
```

See Also

nx_sntp_client_initialize_unicast, nx_sntp_client_reset_unicast, nx_sntp_client_initialize_broadcast, nx_sntp_client_reset_broadcast, nx_sntp_client_run_broadcast, nx_sntp_client_send_unicast_request, nx_sntp_client_receive_time_update, nx_sntp_client_apply_sanity_checks, nx_sntp_client_reset_current_time_message

nx_sntp_client_send_unicast_request

Send unicast request to unicast time server

Prototype

```
UINT nx_sntp_client_send_unicast_request(NX_SNTP_CLIENT *client_ptr,
                        NX_SNTP_TIME_MESSAGE *unicast_request);
```

Description

This service sends a unicast request, which is a NTP time message containing the Client's transmit time stamp and NTP version, to its current unicast server. The NTP time message is appended to a packet buffer and transmitted via the Client UDP socket. The service then releases the packet back to the Client packet pool.

Input Parameters

client_ptr	Pointer to SNTP Client control block
unicast_request	Pointer to the time message request

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Threads

Example

```
/* Send a unicast request, using the time message input to the Client's unicast
server.  */
status =  nx_sntp_client_send_unicast, request(client_ptr, time_message);

/* If status is NX_SUCCESS, the message was successfully transmitted.  */
```

See Also

nx_sntp_client_create_time_request_packet, nx_sntp_client_initialize_unicast, nx_sntp_client_receive_time_update, nx_sntp_client_apply_sanity_checks, nx_sntp_client_reset_current_time_message

nx_sntp_utility_add_msecs_to_NTPtime

Add milliseconds to an NTP time

Prototype

```
UINT nx_sntp_client_utility_add_msecs_to_NTPtime(NX_SNTP_TIME *timeA_ptr,
                                                 LONG msecs_to_add);
```

Description

This service adds the specified milliseconds to the NTP time pointed to by timeA_ptr. The milliseconds to add can be a negative number. However, the result cannot be a negative number (no such thing as negative time!). Also, this service checks for overflow and loss of sign before performing any addition operation.

Input Parameters

timeA_ptr	Pointer NTP time to add milliseconds to
msecs_to_add	milliseconds to add to NTP time

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_OVERFLOW_ERROR	(0xD1A)	Overflow error
NX_SNTP_INVALID_TIME	(0xD18)	Invalid time (e.g. negative)

Allowed From

Application code, Threads

Example

```
/* Add total_msecs_difference to the NTP time pointed to by the client_ptr
instance's local time.  */
```

```
status =  nx_sntp_client_utility_add_msecs_to_NTPtime (client_ptr ->
                                    local_ntp_time, total_msecs_difference);

/* If status is NX_SUCCESS, local time has been increased by the total
milliseconds.  */
```

See Also

nx_sntp_client_utility_add_NTPtime,
nx_sntp_client_utility_addition_overflow_check,
nx_sntp_client_utility_convert_fraction_to_msecs,
nx_sntp_client_utility_fraction_to_usecs,
nx_sntp_client_utility_get_msec_diff,
nx_sntp_client_utility_msecs_to_fraction

nx_sntp_client_utility_add_NTPtime

Add two NTP times to a sum NTP time

Prototype

```
UINT nx_sntp_client_utility_add_ NTPtime(NX_SNTP_TIME *timeA_ptr,
                  NX_SNTP_TIME *timeB_ptr NX_SNTP_TIME *sum_time_ptr,);
```

Description

This service adds the specified NTP times pointed to by timeA_ptr and timeB_ptr to a third NTP time, sum_time_ptr. The caller should NOT use timeA_ptr or timeB_ptr as the sum_time_ptr, since the latter is cleared in memory before the sum of the two operands are applied to it. This service checks for overflow before performing any addition operation.

Input Parameters

timeA_ptr Pointer NTP time to add milliseconds to

msecs_to_add milliseconds to add to NTP time

Return Values

NX_SUCCESS	(0x00)	Client successfully initialized
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_PARAM_ERROR	(0xD00)	Invalid non pointer input
NX_SNTP_OVERFLOW_ERROR	(0xD1A)	Overflow error

Allowed From

Application code, Threads

Example

```
/* Add the two NTP times pointed to by time A and time B pointers and apply the
sum to the NTP time pointed to by sum_time_ptr.  */
status =  nx_sntp_client_utility_add_ NTPtime (timeA_ptr, timeB_ptr,
                                    sum_time_ptr);


/* If status is NX_SUCCESS, sum_time_ptr contains the sum of time A and time B. */
```

See Also

nx_sntp_client_utility_add_msecs_to_NTPtime,
nx_sntp_client_utility_addition_overflow_check,
nx_sntp_client_utility_convert_fraction_to_msecs,
nx_sntp_client_utility_fraction_to_usecs,
nx_sntp_client_utility_get_msec_diff,
nx_sntp_client_utility_msecs_to_fraction

nx_sntp_client_utility_convert_fraction_to_msecs

Convert the fraction field in an NTP time to milliseconds

Prototype

```
UINT nx_sntp_client_utility_convert_fraction_to_msecs(
                    ULONG *milliseconds, NX_SNTP_TIME *time_ptr);
```

Description

This service converts the fraction in an NTP time, which is expressed in fixed point notation, to milliseconds. Note that the result must be positive milliseconds as there can not be negative time.

Input Parameters

milliseconds	Pointer to milliseconds converted
time_ptr	Pointer to NTP time

Return Values

NX_SUCCESS	(0x00)	Successful conversion
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Application code, Threads

Example

```
/* Convert the time fraction in the NTP time to milliseconds. */
status =  nx_sntp_client_utility_convert_fraction_to_msecs milliseconds, time_ptr);

/* If status is NX_SUCCESS, the fraction was successfully converted.  */
```

See Also

nx_sntp_client_utility_add_msecs_to_NTPtime,
nx_sntp_client_utility_add_ NTPtime,
nx_sntp_client_utility_addition_overflow_check,
nx_sntp_client_utility_fraction_to_usecs,
nx_sntp_client_utility_get_msec_diff,
nx_sntp_client_utility_msecs_to_fraction

nx_sntp_client_utility_convert_LONG_to_IP

Convert an IP as a ULONG data to an IP address string

Prototype

```
UINT nx_sntp_client_Utility_convert_LONG_to_IP(
                                        CHAR *buffer, ULONG IP_UL);
```

Description

This service converts an IP address stored as a ULONG to a string in the conventional IP4 format (e.g. 192.2.2.35). This service does not check if IP_UL is a valid IP address.

Input Parameters

buffer	Pointer to IP string buffer
IP_UL	IP address as a ULONG

Return Values

NX_SUCCESS	(0x00)	Successful conversion
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Application code, Threads

Example

```
/* Convert the manycast server address to an IP string stored in buffer. */
status = nx_sntp_client_utility_convert_LONG_to_IP( buffer,
                                    client_ptr -> manycast_server_address);

/* If status is NX_SUCCESS, the IP address was successfully converted.  */
```

See Also

nx_sntp_client_run_broadcast, nx_sntp_client_run_unicast,
nx_sntp_client_add_servers_from_input_list,
nx_sntp_client_remove_from_list,
nx_sntp_client_get_next_server

nx_sntp_client_utility_convert_refID_KOD_code

Convert an NTP Time Reference ID field to a Kiss of Death code

Prototype

```
UINT nx_sntp_client_utility_convert_refID_KOD_code(
                              UCHAR * reference_id, UINT *code_id);
```

Description

This service converts the Reference ID field in an NTP time data to a known Kiss of Death code by matching the reference ID string against the NetX SNTP Client API list of known reference ID codes. Note that the list of ‘known’ KOD codes is an actively changing list in NTP protocol.

Input Parameters

reference_id	Pointer to Reference ID
code_id	Pointer to coded Reference ID

Return Values

NX_SUCCESS	(0x00)	Successful conversion
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Application code, threads

Example

```
/* Convert the Reference ID to a known Kiss of Death code. */
status =  nx_sntp_client_utility_convert_refID_KOD_code(reference_id, code_id);

/* If status is NX_SUCCESS, the Reference ID was successfully converted.  */
```

See Also

nx_sntp_client_apply_sanity_checks

nx_sntp_client_utility_convert_seconds_to_date

Convert an NTP Time to month, day, year and time

Prototype

```
UINT nx_sntp_client_utility_convert_seconds_to_date(
                                NX_SNTP_TIME *current_NTP_time_ptr,
                                UINT current_year,
                                NX_SNTP_DATE_TIME *current_date_time_ptr);
```

Description

This service converts date and time in a NTP data to an NX_SNTP_DATE_TIME object defining year, month, day and time in the NTP time.

Input Parameters

current_NTP_time_ptr	Pointer to NTP time
current_year	Year contained in NTP time
current_date_time_ptr	Pointer to Date Time object

Return Values

NX_SUCCESS	(0x00)	Successful conversion
NX_SNTP_ERROR_CONVERTING_DATETIME	(0xD08)	Error converting Date Time to string

Allowed From

Application code, threads

Example

```
/* Convert the NTP time object into NX_SNTP_DATE_TIME date time object. */
status =  nx_sntp_client_utility_ convert_seconds_to_date
                                                (current_NTP_time_ptr,
                                                current_year,
                                                current_date_time_ptr);

/* If status is NX_SUCCESS, the NX_SNTP_DATE_TIME object was successfully
created and filled in from NTP time.  */
```

See Also

_nx_sntp_client_utility_display_date_time

nx_sntp_client_utility_display_date_time

Convert an NTP Time to Date and Time string

Prototype

```
UINT nx_sntp_client_utility_display_date_time (
                                CHAR *buffer, UINT length,
                                NX_SNTP_DATE_TIME *current_date_time_ptr);
```

Description

This service converts date and time in a NX_SNTP_DATE_TIME object into human readable format displaying date and time if the configurable option NX_SNTP_CURRENT_YEAR is defined e.g. 2008.

Input Parameters

buffer	Pointer to string buffer
length	Size of string buffer
current_date_time_ptr	Pointer to Date Time object

Return Values

NX_SUCCESS	(0x00)	Successful conversion
NX_SNTP_INVALID_DATETIME_BUFFER	(0xD07)	Invalid string buffer
NX_SNTP_ERROR_CONVERTING_DATETIME	(0xD08)	Error converting Date Time to string

Allowed From

Application code, threads

Example

```
/* Convert the NTP Date Time object into human readable date time format.  First
start with an actual NTP time, and convert seconds to date.  Then display the
date and time.  Note that the year is defined in the configurable option
NX_SNTP_CURRENT_YEAR. */
```

```
NTPTime.fraction = 0x3374C000;
NTPTime.seconds = 0xCA91C9C9;
status =  nx_sntp_client_utility_ convert_seconds_to_date
             (current_NTP_time_ptr, NX_SNTP_CURRENT_YEAR,
             current_date_time_ptr);


status =  nx_sntp_client_utility_display_date_time(buffer, length,
                                                   current_date_time_ptr);

/* If status is NX_SUCCESS, the buffer string was successfully created.  */
```

See Also

_nx_sntp_client_utility_convert_seconds_to_date

nx_sntp_client_utility_display_NTP_time

Display an NTP time in seconds and fractions of a second

Prototype

```
UINT nx_sntp_client_utility_display_NTP_time (
                              NX_SNTP_TIME *time_ptr, CHAR *title);
```

Description

If the Client has enabled logging, this service converts the input NTP time to seconds and fractions of a second prepended with a default text “Time: “ and the optional title argument. If logging is not enabled, nothing is displayed and the service returns successful completion. The intended usage of this service is to log time updates on the local host. The size of the buffer is a configurable option. If the final size of the output text exceeds the size of the buffer only a truncated title is prepended to the full time display.

Input Parameters

time_ptr	Pointer to NTP time to display
title	Pointer to optional text to display

Return Values

NX_SUCCESS	(0x00)	Successful conversion
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Application code, threads

Example

```
/* Display the NTP time with title text indicating this is an hourly update. */
status =  nx_sntp_client_utility_display_NTP_time(time_ptr, hourly_update_ptr);

/* If status is NX_SUCCESS, the time was successfully display.  */
```

See Also

nx_sntp_client_utility_convert_fraction_to_msecs

nx_sntp_client_utility_get_msec_diff

Get the time difference in milliseconds between two NTP times

Prototype

```
UINT nx_sntp_client_utility_get_msec_diff(
                            NX_SNTP_TIME *timeA_ptr,
                            NX_SNTP_TIME *timeB_ptr, LONG
                            *total_difference_msecs);
```

Description

This service subtracts the time pointed to by timeB_ptr from the time pointed to by timeA_ptr and returns the result in milliseconds. There is no requirement for a positive difference in time.

Input Parameters

timeA_ptr	Pointer to NTP time to subtract from
timeB_ptr	Pointer to NTP time to subtract
total_difference_msecs	Pointer to difference in milliseconds

Return Values

NX_SUCCESS	(0x00)	Successful conversion
NX_PTR_ERROR	(0x16)	Invalid pointer input
NX_SNTP_SIGN_ERROR	(0xD1B)	Loss of sign error
NX_SNTP_OVERFLOW_ERROR	(0xD1A)	Overflow error

Allowed From

Application code, threads

Example

```
/* Subtract the Receive time from the Transmit time as per RFC protocol and
return the result in milliseconds. */
status =  nx_sntp_client_utility_get_msecs_diff(
                                    &(server_time_message -> transmit_time),
```

```
                                        &(server_time_message -> receive_time),
                                        & msecs_difference);

    /* If status is NX_SUCCESS, the time difference was successfully computed.  */
```

See Also

nx_sntp_client_utility_convert_fraction_to_msecs, nx_sntp_client_calculate_roundtrip, nx_sntp_client_utility_get_msec_diff, nx_sntp_client_process_time_data

Appendix A. Round trip time calculation

The NTP protocol does not require setting the Transmit Timestamp field in the Client request. However the SNTP Client API performs this task as it is highly recommended by RFC 4330 for unicast and manycast modes. It allows a simple calculation to determine the round trip time between a server and the Client.

To calculate the round trip time, the Client sets the Transmit Timestamp field in the request when it sends the time request**.** Note that for this purpose, the local clock need not be synchronized. The server copies this field to the Originate Timestamp in the reply and sets the Receive Timestamp and Transmit Timestamp fields when it receives and transmits the reply respectively.

When a server reply is received, the Client gets the local time, which is the Destination Timestamp variable.

Timestamp Name	ID	When Generated
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 round trip time *d* and system clock offset *t* are defined as:

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

Note that in general both round trip time and offset are signed quantities and can be less than zero. However, a round trip time less than zero is possible only in symmetric (server to server) modes, which the SNTP Client is forbidden to use.

Appendix B. Fatal Error Codes

The following error codes will result in the SNTP Client aborting time updates with the current server. It is up to the host application to decide if the server should be removed from the SNTP Client list of available servers, or simply switch to the next available server on the list. The definition of each error status is defined in *nx_sntp.h.* The API to manipulate the SNTP Client list is shown below. More information is available in **Chapter 4 Description of SNTP Client Services.**

_nx_sntp_client_get_next_server
_nx_sntp_client_remove_server_from_list

When the SNTP Client returns an error from the list below to the host application, the Server should probably be removed. Note that the NX_SNTP_KOD_REMOVE_SERVER error status is left to the SNTP Client kiss of death handler (callback function) to set:

NX_SNTP_KOD_REMOVE_SERVER	0xD0C
NX_SNTP_SERVER_AUTH_FAIL	0xD0D
NX_SNTP_INVALID_NTP_VERSION	0xD11
NX_SNTP_INVALID_SERVER_MODE	0xD12
NX_SNTP_INVALID_SERVER_STRATUM	0xD15

When the SNTP Client returns an error from the list below to the host application, the Server may only temporarily be unable to provide valid time updates and need not be removed:

NX_SNTP_NO_UNICAST_FROM_SERVER	0xD09
NX_SNTP_SERVER_CLOCK_NOT_SYNC	0xD0A
NX_SNTP_KOD_SERVER_NOT_AVAILABLE	0xD0B
NX_SNTP_OVER_BAD_UPDATE_LIMIT	0xD17
NX_SNTP_BAD_SERVER_ROOT_DISPERSION	0xD16
NX_SNTP_INVALID_RTT_TIME	0xD21
NX_SNTP_KOD_SERVER_NOT_AVAILABLE	0xD24