Document title:	**Feil! Fant ikke referansekilden.**
Short title:	**Feil! Fant ikke referansekilden.**
Author(s):	**Feil! Fant ikke referansekilden.**
Classification:	**Feil! Fant ikke referansekilden.**
Revision number:	**Feil! Fant ikke referansekilden.**
Revision history:	

Summary

This is a top level software design document for a demonstration version of a software package tha*t* implements the IEEE1588 Time Synchronization Protocol.

Table of contents

1 Introduction

This document contains the top level software design document for a new product. The contents of this document are *Restricted*.

1.1 Document Purpose

This document contains the functional specifications for an implementation of time synchronization according to the IEEE 1588 standard.

1.2 Abbreviations

PTP	Precision Time Protocol, another name for the IEEE1588 protocol

1.3 References

[1] *1588 – IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Document no. ISBN 0-7381-3369-8 SH95023.

1.4 Definitions

Time Server	A PTP device in the PTP_MASTER state
Time Client	A PTP device in the PTP_SLAVE state

2 A Clock Object For IEEE1588

The time representation in IEE1588 diverges from the time representation used in GPS and NTP in that it uses a "discontinuous" representation:

```
struct TimeRepresentation {
  unsigned long seconds;
  long nanoseconds; }
```

Here the sign of the time representation resides in the nanosecond part. A negative value in the nanosecond part means that the whole time representation is negative.

The discontinuity stems from the fact that maximum value of a signed 32-bit number is slightly above $2 \cdot 10^9$ and it is therefore possible to specify a value of up to more than two seconds in the nanosecond part. GPS and NTP also use a 32 bit representation for the seconds, but the lower 32 bit specifies a binary fraction of a second. Fortunately it is fairly easy to convert between the two representations at the cost of a considerable amount of CPU resources.

2.1 The Algorithms

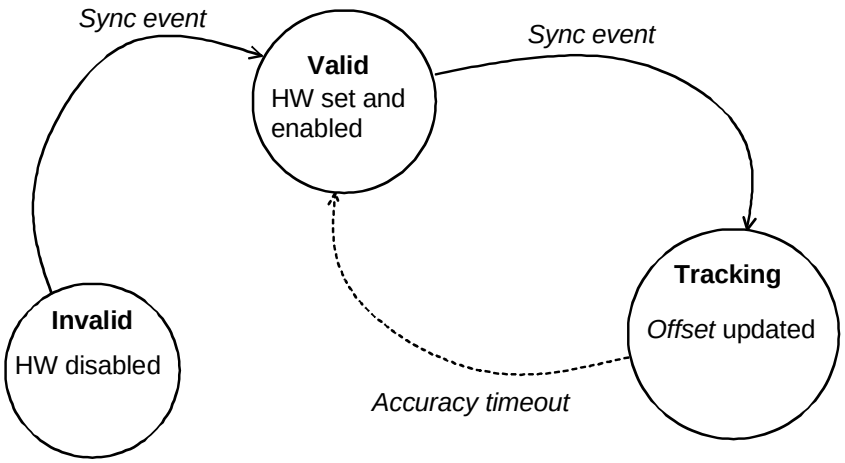

Figure 1. State map of the Real Time clock

2.2 The Methods

2.2.1 RTCinit

RTCinit *takes no parameters and returns nothing.*

- Set all local variables to their initial state
- Retrieve the system clock stratum and identifier
- Retrieve the system clock resolution.
- If the system clock resolution is better than 10μs:

- Call the system clock routine twice to determine the time spent in that routine. Save this time as one calibration constant.
- Call *getRTCvalue* twice to determine the time spent in that routine. Save this time as another calibration constant.

2.2.2 getRTCvalue

getRTCvalue *takes one parameter and returns a 64 bit Real Time Clock value (an RTC time stamp). If the parameter is >0, the returned time corresponds to the actual time of the return of the method*[1]*. If the parameter is <0, the returned time corresponds to the time the method was invoked. If the parameter is 0, the returned time corresponds to the uncorrected time that the system clock returned. A returned value of all 1's means that the returned value is invalid.*

- Call the system clock routine to get the system time
- Correct the time according to the parameter.
- Convert the time to IEEE 1588 representation

2.2.3 *convertRTChw*

convertRTChw *takes a 32 bit parameter and returns a 64 bit Real Time Clock value corresponding to the hardware clock value of the input parameter.*

- Call the system clock routine to get the system time
- Substitute the parameter in the correct place in the system time
- If the resulting time is greater than the system time, increase the part of the resulting time that is more significant than the substituted parameter.
- Convert that time to IEEE 1588 representation

2.2.4 *synchRTChw*

synchRTChw takes two parameters, one 32 bit parameter corresponding to a value of the hardware clock and one 64 bit Real Time Clock value which is the synchronizing value corresponding to that hardware clock value.

2.2.5 *synchRTCsw*

synchRTCsw *takes two parameters, one 64 bit Real Time Clock referring to an internal time value and one 64 bit Real Time Clock value which is the synchronizing value corresponding to that internal time value.*

- Store the values and update internal correction parameters

[1] If interrupts are enabled when this method is used, the returned value may be off by a large amount of time.

2.2.6 *getRTCstate*

getRTCstate *takes no parameters and returns the internal state of the Real Time Clock (Invalid, Valid or Tracking).*

2.2.7 *SetFreqCorr*

SetFreqCorr takes one parameter, 32 bit signed value containing a correction factor for the Real Time Clock crystal oscillator.

2.2.8 *SetRelFreqCorr*

SetRelFreqCorr takes one parameter, 32 bit signed value (representing a value between -0.5 and 0.5) containing a relative correction factor for the Real Time Clock crystal oscillator.

2.3 Calculating the Correction Parameters

Let *T* stand for the PTP timestamp and let *t* stand for the value of the internal clock when the PTP message containing *T* was received.

1. Calculate *P* as the value of *T* after it has been corrected for transmission delay and converted to a binary + binary fraction representation.
2. Save *P* in a temporary location
3. Calculate $\Delta = P - t$.
4. Define a two-entry array of records (struct) containing the difference between global and local time (as defined above) plus the corresponding local time.
5. Copy entry 0 in the array to entry 1.
6. Place (Δ, t) in array entry 0
7. Calculate the slope as $\alpha = \frac{\Delta_0 - \Delta_1}{t_0 - t_1}$
8. Copy the stored value of *P* to P_0

2.4 Calculate the Corrected Time Value

1. Fetch the system time *t*
2. If necessary, convert *t* to a binary + binary fraction representation.
3. Calculate $P = \alpha(t - t_0) + P_0$
4. Convert *P* to PTP representation as *T*.

3 Win32 Clock Internals

3.1 GetSystemTimeAsFileTime

The GetSystemTimeAsFileTime function obtains the current system date and time. The information is in Coordinated Universal Time (UTC) format.

```
VOID GetSystemTimeAsFileTime(
    LPFILETIME lpSystemTimeAsFileTime    // pointer to a file
time structure
   );
```

Parameters

lpSystemTimeAsFileTime

Pointer to a FILETIME structure to receive the current system date and time in UTC format.

Return Values

This function does not return a value.

Remarks

The GetSystemTimeAsFileTime function is equivalent to the following code sequence:

```
FILETIME ft;
SYSTEMTIME st;
GetSystemTime(&st);
SystemTimeToFileTime(&st,&ft);
```

See Also

FILETIME, GetSystemTime, SYSTEMTIME, SystemTimeToFileTime

3.2 Handling FileTime

3.2.1 FILETIME

The FILETIME structure is a 64-bit value representing the number of 100-nanosecond intervals since January 1, 1601.

```
typedef struct _FILETIME { // ft
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME;
```

Members

dwLowDateTime

Specifies the low-order 32 bits of the file time.

dwHighDateTime

Specifies the high-order 32 bits of the file time.

Remarks

It is not recommended that you add and subtract values from the FILETIME structure to obtain relative times. Instead, you should

- Copy the resulting FILETIME structure to a LARGE_INTEGER structure.
- Use normal 64-bit arithmetic on the LARGE_INTEGER value.

Additional remarks

In order to convert FILETIME to PTP time:

1. Subtract the FILETIME difference between 1/1/1601 and 1/1/1970 from the FILETIME (the number is 116445600000000000 or 0x19DB2A7FFA84000).
2. Shift the resulting value 8 bits to the left.
3. Do a fix-point multiply of the high and low half of that result by 1.6777216.

3.2.2 LARGE_INTEGER

The LARGE_INTEGER structure is used to represent a 64-bit signed integer value.

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
        LONG  HighPart;
    };
    LONGLONG QuadPart;
  } LARGE_INTEGER;
```

Members

LowPart

Specifies the low-order 32 bits.

HighPart

Specifies the high-order 32 bits.

QuadPart

Specifies a 64-bit signed integer.

Remarks

The LARGE_INTEGER structure is actually a union. If your compiler has built-in support for 64-bit integers, use the QuadPart member to store the 64-bit integer. Otherwise, use the LowPart and HighPart members to store the 64-bit integer.

4 VxWorks Clock Internals

4.1 The VxWorks System Time - clockLib

4.1.1 NAME

clockLib *- clock library (POSIX)*

4.1.2 ROUTINES

clock_getres*( ) - get the clock resolution (POSIX)*

clock_setres*( ) - set the clock resolution*

clock_gettime*( ) - get the current time of the clock (POSIX)*

clock_settime*( ) - set the clock to a specified time (POSIX)*

4.1.3 DESCRIPTION

This library provides a clock interface, as defined in the IEEE standard, POSIX 1003.1b.

A clock is a software construct that keeps time in seconds and nanoseconds. The clock has a simple interface with three routines: **clock_settime*( )***, **clock_gettime*( )***, *and* **clock_getres*( )***. The non-POSIX routine* **clock_setres*( )*** *is provided (temporarily) so that* ***clockLib*** *is informed if there are changes in the system clock rate (e.g., after a call to* **sysClkRateSet*( )***).*

Times used in these routines are stored in the timespec structure:

```
struct timespec
    {
    time_t      tv_sec;    /* seconds */
    long        tv_nsec;   /* nanoseconds (0 -1,000,000,000) */
    };
```

4.1.4 IMPLEMENTATION

Only one clock_id *is supported, the required* ***CLOCK_REALTIME****. Conceivably, additional "virtual" clocks could be supported, or support for additional auxiliary clock hardware (if available) could be added.*

4.1.5 INCLUDE FILES

timers.h

4.2 Using the VxWorks Clock Library

1. The date origin for the POSIX 1003.1b timestamps is very obscurely documented in the VxWorks documentation (read: impossible to find). A perusal of POSIX-related forums on the Internet supplied some data indicating that the origin was 1/1/1970, the same origin as the PTP clock.
2. POSIX 1003.1b timers ignore leap seconds, and therefore differ from UTC by about 11s (2003).
3. The *timespec* structure resembles the PTP *TimeRepresentation* structure very much. In the timeserver case this means that the result from clock_*gettime* can be used with PTP after a trivial *copy* procedure. A time client must start by converting the result to linear time and doing the calculations specified in 2.4.

5 General Considerations

5.1 Initial Values

Some system-related variables must be initialized before the PTP sync system starts. Some of these are:

- The "Recommended State" constant used in the Protocol Engine State Machine.
- The default data set.

5.1.1 The Default Data Set Members

Variable	**Contents**
clockCommTech	The clock communication technology. Our case it is Ethernet - PTP_ETHER.
ClockUuid	This is the name of the clock implementation. The maximum name length is 6 characters, it will be WINPTP.
clockPort	
ClockStratum	In our case it will usually be 3.
clockIdentifier	The clock accuracy. In our case it will usually be HAND, INIT or DEFLT.
clockVariance	May be calculated online or loaded as an initial value. The default value for a standard crystal clock should be about –8127.
clockFollowupCapable	Indicates whether we are able to create a follow-up message with a greater precision than the standard sync message. Will probably be FALSE on a Windows implementation.
preferred	In our case it should usually be initialized to FALSE.
initializable	The value should be initialized to TRUE if there is some automatic way of initializing the clock upon power-up.
externalTiming	A standard implementation will have FALSE here (there will be no dedicated pin on which to present a timing signal).
isBoundaryClock	Will only be TRUE if we have at least two communication interfaces with PTP capability on each. In our case this means FALSE.
syncInterval	Should be kept at the default value (1).
subdomainName	A maximum of 16 characters. The first five are defined in the

Variable	Contents
	standard, The rest will be filled with PICKMASTER and the appropriate number of zeros to make a total of 16.
numberPorts	1 for an ordinary clock, otherwise equal to the number of ports.
numberForeignRecords	The implementation is limited by the amount of available memory, but the variable should be initialized to 8.

5.2 Network Startup Issues

Since PTP uses multicast communication, the network startup code is somewhat more involved than a normal startup sequence.

1. Create a socket with parameters AF_INET and SOCK_DGRAM.
2. Bind the socket to INADDR_ANY and the PTP event port (319).
3. Create a multicast structure with the PTP primary group address (224.0.1.129)
4. Execute a *setsockopt* with parameters IPPROTO_IP, IP_ADD_MEMBERSHIP and the multicast structure.
5. Repeat 1 – 4 using the PTP general port (320).
6. Spawn one thread to *listen* to each of the ports.

5.3 Network Shutdown Issues

1. Execute a *setsockopt* with parameters IPPROTO_IP, IP_DROP_MEMBERSHIP and the multicast structure.
2. Close the socket.