Network Working Group
Request for Comments: 1589
Category: Informational

D. Mills
University of Delaware
March 1994

A Kernel Model for Precision Timekeeping

Status of this Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Overview

This memorandum describes an engineering model which implements a precision time-of-day function for a generic operating system. The model is based on the principles of disciplined oscillators and phase-lock loops (PLL) often found in the engineering literature. It has been implemented in the Unix kernel for several workstations, including those made by Sun Microsystems and Digital Equipment. The model changes the way the system clock is adjusted in time and frequency, as well as provides mechanisms to discipline its frequency to an external precision timing source. The model incorporates a generic system-call interface for use with the Network Time Protocol (NTP) or similar time synchronization protocol. The NTP Version 3 daemon xntpd operates with this model to provide synchronization limited in principle only by the accuracy and stability of the external timing source.

This memorandum does not obsolete or update any RFC. It does not propose a standard protocol, specification or algorithm. It is intended to provoke comment, refinement and alternative implementations. While a working knowledge of NTP is not required for an understanding of the design principles or implementation of the model, it may be helpful in understanding how the model behaves in a fully functional timekeeping system. The architecture and design of NTP is described in [1], while the current NTP Version 3 protocol specification is given in RFC-1305 [2] and a subset of the protocol, the Simple Network Time Protocol (SNTP), in RFC-1361 [4].

The model has been implemented in three Unix kernels for Sun Microsystems and Digital Equipment workstations. In addition, for the Digital machines the model provides improved precision to one microsecond (us). Since these specific implementations involve modifications to licensed code, they cannot be provided directly. Inquiries should be directed to the manufacturer's representatives. However, the engineering model for these implementations, including a

simulator with code segments almost identical to the implementations, but not involving licensed code, is available via anonymous FTP from host louie.udel.edu in the directory pub/ntp and compressed tar archive kernel.tar.Z. The NTP Version 3 distribution can be obtained via anonymous ftp from the same host and directory in the compressed tar archive xntp3.3g.tar.Z, where the version number shown as 3.3g may be adjusted for new versions as they occur.

1. Introduction

This memorandum describes a model and programming interface for generic operating system software that manages the system clock and timer functions. The model provides improved accuracy and stability for most workstations and servers using the Network Time Protocol (NTP) or similar time synchronization protocol. This memorandum describes the principles of design and implementation of the model. Related technical reports discuss the design approach, engineering analysis and performance evaluation of the model as implemented in Unix kernels for Sun Microsystems and Digital Equipment workstations. The NTP Version 3 daemon xntpd operates with these implementations to provide improved accuracy and stability, together with diminished overhead in the operating system and network. In addition, the model supports the use of external timing sources, such as precision pulse-per-second (PPS) signals and the industry standard IRIG timing signals. The NTP daemon automatically detects the presence of the new features and utilizes them when available.

There are three prototype implementations of the model presented in this memorandum, one each for the Sun Microsystems SPARCstation with the SunOS 4.1.x kernel, Digital Equipment DECstation 5000 with the Ultrix 4.x kernel and Digital Equipment 3000 AXP Alpha with the OSF/1 V1.x kernel. In addition, for the DECstation 5000/240 and 3000 AXP Alpha machines, a special feature provides improved precision to 1 us (Sun 4.1.x kernels already do provide 1-us precision). Other than improving the system clock accuracy, stability and precision, these implementations do not change the operation of existing Unix system calls which manage the system clock, such as gettimeofday(), settimeofday() and adjtime(); however, if the new features are in use, the operations of gettimeofday() and adjtime() can be controlled instead by new system calls ntp_gettime() and ntp_adjtime() as described below.

A detailed description of the variables and algorithms is given in the hope that similar functionality can be incorporated in Unix kernels for other machines. The algorithms involve only minor changes to the system clock and interval timer routines and include interfaces for application programs to learn the system clock status and certain statistics of the time synchronization process. Detailed

installation instructions are given in a specific README files included in the kernel distributions.

In this memorandum, NTP Version 3 and the Unix implementation xntp3 are used as an example application of the new system calls for use by a synchronization daemon. In principle, the new system calls can be used by other protocols and implementations as well. Even in cases where the local time is maintained by periodic exchanges of messages at relatively long intervals, such as using the NIST Automated Computer Time Service, the ability to precisely adjust the system clock frequency simplifies the synchronization procedures and allows the telephone call frequency to be considerably reduced.

2. Design Approach

While not strictly necessary for an understanding or implementation of the model, it may be helpful to briefly describe how NTP operates to control the system clock in a client workstation. As described in [1], the NTP protocol exchanges timestamps with one or more peers sharing a synchronization subnet to calculate the time offsets between peer clocks and the local clock. These offsets are processed by several algorithms which refine and combine the offsets to produce an ensemble average, which is then used to adjust the local clock time and frequency. The manner in which the local clock is adjusted represents the main topic of this memorandum. The goal in the enterprise is the most accurate and stable system clock possible with the available kernel software and workstation hardware.

In order to understand how the new software works, it is useful to review how most Unix kernels maintain the system time. In the Unix design a hardware counter interrupts the kernel at a fixed rate: 100 Hz in the SunOS kernel, 256 Hz in the Ultrix kernel and 1024 Hz in the OSF/1 kernel. Since the Ultrix timer interval (reciprocal of the rate) does not evenly divide one second in microseconds, the Ultrix kernel adds 64 microseconds once each second, so the timescale consists of 255 advances of 3906 us plus one of 3970 us. Similarly, the OSF/1 kernel adds 576 us once each second, so its timescale consists of 1023 advances of 976 us plus one of 1552 us.

2.1. Mechanisms to Adjust Time and Frequency

In most Unix kernels it is possible to slew the system clock to a new offset relative to the current time by using the adjtime() system call. To do this the clock frequency is changed by adding or subtracting a fixed amount (tickadj) at each timer interrupt (tick) for a calculated number of ticks. Since this calculation involves dividing the requested offset by tickadj, it is possible to slew to a new offset with a precision only of tickadj, which is

usually in the neighborhood of 5 us, but sometimes much more. This results in a roundoff error which can accumulate to an unacceptable degree, so that special provisions must be made in the clock adjustment procedures of the synchronization daemon.

In order to implement a frequency-discipline function, it is necessary to provide time offset adjustments to the kernel at regular adjustment intervals using the adjtime() system call. In order to reduce the system clock jitter to the regime considered in this memorandum, it is necessary that the adjustment interval be relatively small, in the neighborhood of 1 s. However, the Unix adjtime() implementation requires each offset adjustment to complete before another one can be begun, which means that large adjustments must be amortized in possibly many adjustment intervals. The requirement to implement the adjustment interval and compensate for roundoff error considerably complicates the synchronizing daemon implementation.

In the new model this scheme is replaced by another that represents the system clock as a multiple-word, precision-time variable in order to provide very precise clock adjustments. At each timer interrupt a precisely calibrated quantity is added to the kernel time variable and overflows propagated as required. The quantity is computed as in the NTP local clock model described in [3], which operates as an adaptive-parameter, first-order, type-II phase-lock loop (PLL). In principle, this PLL design can provide precision control of the system clock oscillator within 1 us and frequency to within parts in 10^11. While precisions of this order are surely well beyond the capabilities of the CPU clock oscillator used in typical workstations, they are appropriate using precision external oscillators as described below.

The PLL design is identical to the one originally implemented in NTP and described in [3]. In this design the software daemon simulates the PLL using the adjtime() system call; however, the daemon implementation is considerably complicated by the considerations described above. The modified kernel routines implement the PLL in the kernel using precision time and frequency represention, so that these complications are avoided. A new system call ntp_adjtime() is called only as each new time update is determined, which in NTP occurs at intervals of from 16 s to 1024 s. In addition, doing frequency compensation in the kernel means that the system time runs true even if the daemon were to cease operation or the network paths to the primary synchronization source fail.

In the new model the new ntp_adjtime() operates in a way similar to the original adjtime() system call, but does so independently

of adjtime(), which continues to operate in its traditional fashion. When used with NTP, it is the design intent that settimeofday() or adjtime() be used only for system time adjustments greater than +-128 ms, although the dynamic range of the new model is much larger at +-512 ms. It has been the Internet experience that the need to change the system time in increments greater than +-128 ms is extremely rare and is usually associated with a hardware or software malfunction or system reboot.

The easiest way to set the time is with the settimeofday() system call; however, this can under some conditions cause the clock to jump backward. If this cannot be tolerated, adjtime() can be used to slew the clock to the new value without running backward or affecting the frequency discipline process. Once the system clock has been set within +-128 ms, the ntp_adjtime() system call is used to provide periodic updates including the time offset, maximum error, estimated error and PLL time constant. With NTP the update interval depends on the measured dispersion and time constant; however, the scheme is quite forgiving and neither moderate loss of updates nor variations in the update interval are serious.

2.2 Daemon and Application Interface

Unix application programs can read the system clock using the gettimeofday() system call, which returns only the system time and timezone data. For some applications it is useful to know the maximum error of the reported time due to all causes, including clock reading errors, oscillator frequency errors and accumulated latencies on the path to a primary synchronization source. However, in the new model the PLL adjusts the system clock to compensate for its intrinsic frequency error, so that the time errors expected in normal operation will usually be much less than the maximum error. The programming interface includes a new system call ntp_gettime(), which returns the system time, as well as the maximum error and estimated error. This interface is intended to support applications that need such things, including distributed file systems, multimedia teleconferencing and other real-time applications. The programming interface also includes the new system call ntp_adjtime() mentioned previously, which can be used to read and write kernel variables for time and frequency adjustment, PLL time constant, leap-second warning and related data.

In addition, the kernel adjusts the maximum error to grow by an amount equal to the oscillator frequency tolerance times the elapsed time since the last update. The default engineering parameters have been optimized for update intervals in the order

of 64 s. For other intervals the PLL time constant can be adjusted to optimize the dynamic response over intervals of 16-1024 s. Normally, this is automatically done by NTP. In any case, if updates are suspended, the PLL coasts at the frequency last determined, which usually results in errors increasing only to a few tens of milliseconds over a day using room-temperature quartz oscillators of typical modern workstations.

While any synchronization daemon can in principle be modified to use the new system calls, the most likely will be users of the NTP Version 3 daemon xntpd. The xntpd code determines whether the new system calls are implemented and automatically reconfigures as required. When implemented, the daemon reads the frequency offset from a file and provides it and the initial time constant via ntp_adjtime(). In subsequent calls to ntp_adjtime(), only the time offset and time constant are affected. The daemon reads the frequency from the kernel using ntp_adjtime() at intervals of about one hour and writes it to a system file. This information is recovered when the daemon is restarted after reboot, for example, so the sometimes extensive training period to learn the frequency separately for each system can be avoided.

2.3. Precision Clocks for DECstation 5000/240 and 3000 AXP Alpha

The stock microtime() routine in the Ultrix kernel returns system time to the precision of the timer interrupt interval, which is in the 1-4 ms range. However, in the DECstation 5000/240 and possibly other machines of that family, there is an undocumented IOASIC hardware register that counts system bus cycles at a rate of 25 MHz. The new microtime() routine for the Ultrix kernel uses this register to interpolate system time between timer interrupts. This results in a precision of 1 us for all time values obtained via the gettimeofday() and ntp_gettime() system calls. For the Digital Equipment 3000 AXP Alpha, the architecture provides a hardware Process Cycle Counter and a machine instruction rpcc to read it. This counter operates at the fundamental frequency of the CPU clock or some submultiple of it, 133.333 MHz for the 3000/400 for example. The new microtime() routine for the OSF/1 kernel uses this counter in the same fashion as the Ultrix routine.

In both the Ultrix and OSF/1 kernels the gettimeofday() and ntp_gettime() system call use the new microtime() routine, which returns the actual interpolated value, but does not change the kernel time variable. Therefore, other routines that access the kernel time variable directly and do not call either gettimeofday(), ntp_gettime() or microtime() will continue their present behavior. The microtime() feature is independent of other features described here and is operative even if the kernel PLL or

new system calls have not been implemented.

The SunOS kernel already includes a system clock with 1-us resolution; so, in principle, no microtime() routine is necessary. An existing kernel routine uniqtime() implements this function, but it is coded in the C language and is rather slow at 42-85 us per call. A replacement microtime() routine coded in assembler language is available in the NTP Version 3 distribution and is much faster at about 3 us per call.

2.4. External Time and Frequency Discipline

The overall accuracy of a time synchronization subnet with respect to Coordinated Universal Time (UTC) depends on the accuracy and stability of the primary synchronization source, usually a radio or satellite receiver, and the system clock oscillator of the primary server. As discussed in [5], the traditional interface using an RS232 protocol and serial port precludes the full accuracy of the radio clock. In addition, the poor stability of typical CPU clock oscillators limits the accuracy, whether or not precision time sources are available. There are, however, several ways in which the system clock accuracy and stability can be improved to the degree limited only by the accuracy and stability of the synchronization source and the jitter of the operating system.

Many radio clocks produce special signals that can be used by external equipment to precisely synchronize time and frequency. Most produce a pulse-per-second (PPS) signal that can be read via a modem-control lead of a serial port and some produce a special IRIG signal that can be read directly by a bus peripheral, such as the KSI/Odetics TPRO IRIG SBus interface, or indirectly via the audio codec of some workstations, as described in [5]. In the NTP Version 3 distribution, the PPS signal can be used to augment the less precise ASCII serial timecode to improve accuracy to the order of microseconds. Support is also included in the distribution for the TPRO interface as well as the audio codec; however, the latter requires a modified kernel audio driver contained in the bsd_audio.tar.Z distribution in the same host and directory as the NTP Version 3 distribution mentioned previously.

2.4.1. PPS Signal

The NTP Version 3 distribution includes a special ppsclock module for the SunOS 4.1.x kernel that captures the PPS signal presented via a modem-control lead of a serial port. Normally, the ppsclock module produces a timestamp at each transition of the PPS signal and provides it to the synchronization daemon

for integration with the serial ASCII timecode, also produced by the radio clock. With the conventional PLL implementation in either the daemon or the kernel as described above, the accuracy of this scheme is limited by the intrinsic stability of the CPU clock oscillator to a millisecond or two, depending on environmental temperature variations.

The ppsclock module has been modified to in addition call a new kernel routine hardpps() once each second. The kernel routine compares the timestamp with a sample of the CPU clock oscillator to develop a frequency offset estimate. This offset is used to discipline the oscillator frequency, nominally to within a few parts in 10^8, which is about two orders of magnitude better than the undisciplined oscillator. The new feature is conditionally compiled in the code described below only if the PPS_SYNC option is used in the kernel configuration file.

When using the PPS signal to adjust the time, there is a problem with the SunOS implementation which is very delicate to fix. The Sun serial port interrupt routine operates at interrupt priority level 12, while the timer interrupt routine operates at priority 10. Thus, it is possible that the PPS signal interrupt can occur during the timer interrupt routine, with result that a tick increment can be missed and the returned time early by one tick. It may happen that, if the CPU clock oscillator is within a few ppm of the PPS oscillator, this condition can persist for two or more successive PPS interrupts. A useful workaround has been to use a median filter to process the PPS sample offsets. In this filter the sample offsets in a window of 20 samples are sorted by offset and the six highest and six lowest outlyers discarded. The average of the eight samples remaining becomes the output of the filter.

The problem is not nearly so serious when using the PPS signal to discipline the frequency of the CPU clock oscillator. In this case the quantity of interest is the contents of the microseconds counter only, which does not depend on the kernel time variable.

2.4.2. External Clocks

It is possible to replace the system clock function with an external bus peripheral. The TPRO device mentioned previously can be used to provide IRIG-synchronized time with a precision of 1 us. A driver for this device tprotime.c and header file tpro.h are included in the kernel.tar.Z distribution mentioned previously. Using this device the system clock is read directly

from the interface; however, the device does not record the year, so special provisions have to be made to obtain the year from the kernel time variable and initialize the driver accordingly. This feature is conditionally compiled in the code described below only if the EXT_CLOCK option is used in the kernel configuration file.

While the system clock function is provided directly by the microtime() routine in the driver, the kernel time variable must be disciplined as well, since not all system timing functions use the microtime() routine. This is done by measuring the difference between the microtime() clock and kernel time variable and using the difference to adjust the kernel PLL as if the adjustment were provided by an external peer and NTP.

A good deal of error checking is done in the TPRO driver, since the system clock is vulnerable to a misbehaving radio clock, IRIG signal source, interface cables and TPRO device itself. Unfortunately, there is no easy way to utilize the extensive diversity and redundancy capabilities available in the NTP synchronization daemon. In order to avoid disruptions that might occur if the TPRO time is far different from the kernel time variable, the latter is used instead of the former if the difference between the two exceeds 1000 s; presumably in that case operator intervention is required.

2.4.3. External Oscillators

Even if a source of PPS or IRIG signals is not available, it is still possible to improve the stability of the system clock through the use of a specialized bus peripheral. In order to explore the benefits of such an approach, a special SBus peripheral caled HIGHBALL has been constructed. The device includes a pair of 32-bit hardware counters in Unix timeval format, together with a precision, oven-controlled quartz oscillator with a stability of a few parts in 10^9. A driver for this device hightime.c and header file high.h are included in the kernel.tar.Z distribution mentioned previously. This feature is conditionally compiled in the code described below only if the EXT_CLOCK and HIGHBALL options are used in the kernel configuration file.

Unlike the external clock case, where the system clock function is provided directly by the microtime() routine in the driver, the HIGHBALL counter offsets with respect to UTC must be provided first. This is done using the ordinary kernel PLL, but controlling the counter offsets directly, rather than the

kernel time variable. At first, this might seem to defeat the purpose of the design, since the jitter and wander of the synchronization source will affect the counter offsets and thus the accuracy of the time. However, the jitter is much reduced by the PLL and the wander is small, especially if using a radio clock or another primary server disciplined in the same way. In practice, the scheme works to reduce the incidental wander to a few parts in 10^8, or about the same as using the PPS signal.

As in the previous case, the kernel time variable must be disciplined as well, since not all system timing functions use the microtime() routine. However, the kernel PLL cannot be used for this, since it is already in use providing offsets for the HIGHBALL counters. Therefore, a special correction is calculated from the difference between the microtime() clock and the kernel time variable and used to adjust the kernel time variable at the next timer interrupt. This somewhat roundabout approach is necessary in order that the adjustment does not cause the kernel time variable to jump backwards and possibly lose or duplicate a timer event.

2.5 Other Features

It is a design feature of the NTP architecture that the system clocks in a synchronization subnet are to read the same or nearly the same values before during and after a leap-second event, as declared by national standards bodies. The new model is designed to implement the leap event upon command by an ntp_adjtime() argument. The intricate and sometimes arcane details of the model and implementation are discussed in [3] and [5]. Further details are given in the technical summary later in this memorandum.

3. Technical Summary

In order to more fully understand the workings of the model, a stand-alone simulator kern.c and header file timex.h are included in the kernel.tar.Z distribution mentioned previously. In addition, a complete C program kern_ntptime.c which implements the ntp_gettime() and ntp_adjtime() functions is provided, but with the vendor-specific argument-passing code deleted. Since the distribution is somewhat large, due to copious comments and ornamentation, it is impractical to include a listing of these programs in this memorandum. In any case, implementors may choose to snip portions of the simulator for use in new kernel designs, but, due to formatting conventions, this would be difficult if included in this memorandum.

The kern.c program is an implementation of an adaptive-parameter, first-order, type-II phase-lock loop. The system clock is implemented using a set of variables and algorithms defined in the simulator and driven by explicit offsets generated by a driver program also included in the program. The algorithms include code fragments almost identical to those in the machine-specific kernel implementations and operate in the same way, but the operations can be understood separately from any licensed source code into which these fragments may be integrated. The code fragments themselves are not derived from any licensed code. The following discussion assumes that the simulator code is available for inspection.

3.1. PLL Simulation

The simulator operates in conformance with the analytical model described in [3]. The main() program operates as a driver for the fragments hardupdate(), hardclock(), second_overflow(), hardpps() and microtime(), although not all functions implemented in these fragments are simulated. The program simulates the PLL at each timer interrupt and prints a summary of critical program variables at each time update.

There are three defined options in the kernel configuration file specific to each implementation. The PPS_SYNC option provides support for a pulse-per-second (PPS) signal, which is used to discipline the frequency of the CPU clock oscillator. The EXT_CLOCK option provides support for an external kernel-readable clock, such as the KSI/Odetics TPRO IRIG interface or HIGHBALL precision oscillator, both for the SBus. The TPRO option provides support for the former, while the HIGHBALL option provides support for the latter. External clocks are implemented as the microtime() clock driver, with the specific source code selected by the kernel configuration file.

3.1.1. The hardupdate() Fragment

The hardupdate() fragment is called by ntp_adjtime() as each update is computed to adjust the system clock phase and frequency. Note that the time constant is in units of powers of two, so that multiplies can be done by simple shifts. The phase variable is computed as the offset divided by the time constant. Then, the time since the last update is computed and clamped to a maximum (for robustness) and to zero if initializing. The offset is multiplied (sorry about the ugly multiply) by the result and divided by the square of the time constant and then added to the frequency variable. Note that all shifts are assumed to be positive and that a shift of a signed quantity to the right requires a little dance.

With the defines given, the maximum time offset is determined by the size in bits of the long type (32 or 64) less the SHIFT_UPDATE scale factor (12) or at least 20 bits (signed). The scale factor is chosen so that there is no loss of significance in later steps, which may involve a right shift up to SHIFT_UPDATE bits. This results in a time adjustment range over +-512 ms. Since time_constant must be greater than or equal to zero, the maximum frequency offset is determined by the SHIFT_USEC scale factor (16) or at least 16 bits (signed). This results in a frequency adjustment range over +-31,500 ppm.

In the addition step, the value of offset * mtemp is not greater than MAXPHASE * MAXSEC = 31 bits (signed), which will not overflow a long add on a 32-bit machine. There could be a loss of precision due to the right shift of up to 12 bits, since time_constant is bounded at 6. This results in a net worst-case frequency resolution of about .063 ppm, which is not significant for most quartz oscillators. The worst case could be realized only if the NTP peer misbehaves according to the protocol specification.

The time_offset value is clamped upon entry. The time_phase variable is an accumulator, so is clamped to the tolerance on every call. This helps to damp transients before the oscillator frequency has been determined, as well as to satisfy the correctness assertions if the time synchronization protocol or implementation misbehaves.

3.1.2. The hardclock() Fragment

The hardclock() fragment is inserted in the hardware timer interrupt routine at the point the system clock is to be incremented. Previous to this fragment the time_update variable has been initialized to the value computed by the adjtime() system call in the stock Unix kernel, normally plus/minus the tickadj value, which is usually in the order of 5 us. The time_phase variable, which represents the instantaneous phase of the system clock, is advanced by time_adj, which is calculated in the second_overflow() fragment described below. If the value of time_phase exceeds 1 us in scaled units, time_update is increased by the (signed) excess and time_phase retains the residue.

Except in the case of an external oscillator such as the HIGHBALL interface, the hardclock() fragment advances the system clock by the value of tick plus time_update. However, in the case of an external oscillator, the system clock is obtained directly from the interface and time_update used to

discipline that interface instead. However, the system clock must still be disciplined as explained previously, so the value of clock_cpu computed by the second_overflow() fragment is used instead.

3.1.3. The second_overflow() Fragment

The second_overflow() fragment is inserted at the point where the microseconds field of the system time variable is being checked for overflow. Upon overflow the maximum error time_maxerror is increased by time_tolerance to reflect the maximum time offset due to oscillator frequency error. Then, the increment time_adj to advance the kernel time variable is calculated from the (scaled) time_offset and time_freq variables updated at the last call to the hardclock() fragment.

The phase adjustment is calculated as a (signed) fraction of the time_offset remaining, where the fraction is added to time_adj, then subtracted from time_offset. This technique provides a rapid convergence when offsets are high, together with good resolution when offsets are low. The frequency adjustment is the sum of the (scaled) time_freq variable, an adjustment necessary when the tick interval does not evenly divide one second fixtick and PPS frequency adjustment pps_ybar (if configured).

The scheme of approximating exact multiply/divide operations with shifts produces good results, except when an exact calculation is required, such as when the PPS signal is being used to discipling the CPU clock oscillator frequency, as described below. As long as the actual oscillator frequency is a power of two in seconds, no correction is required. However, in the SunOS kernel the clock frequency is 100 Hz, which results in an error factor of 0.78. In this case the code increases time_adj by a factor of 1.25, which results in an overall error less than three percent.

On rollover of the day, the leap-second state machine described below determines whether a second is to be inserted or deleted in the timescale. The microtime() routine insures that the reported time is always monotonically increasing.

3.1.4. The hardpps() Fragment

The hardpps() fragment is operative only if the PPS_SYNC option is specified in the kernel configuration file. It is called from the serial port driver or equivalent interface at the on-time transition of the PPS signal. The fragment operates as a

first-order, type-I frequency-lock loop (FLL) controlled by the difference between the frequency represented by the pps_ybar variable and the frequency of the hardware clock oscillator.

In order to avoid calling the microtime() routine more than once for each PPS transition, the interface requires the calling program to capture the system time and hardware counter contents at the on-time transition of the PPS signal and provide a pointer to the timestamp (Unix timeval) and counter contents as arguments to the hardpps() call. The hardware counter contents can be determined by saving the microseconds field of the system time, calling the microtime() routine, and subtracting the saved value. If a counter overflow has occured during the process, the resulting microseconds value will be negative, in which case the caller adds 1000000 to normalize the microseconds field.

The frequency of the hardware oscillator can be determined from the difference in hardware counter readings at the beginning and end of the calibration interval divided by the duration of the interval. However, the oscillator frequency tolerance, as much as 100 ppm, may cause the difference to exceed the tick value, creating an ambiguity. In order to avoid this ambiguity, the hardware counter value at the beginning of the interval is increased by the current pps_ybar value once each second, but computed modulo the tick value. At the end of the interval, the difference between this value and the value computed from the hardware counter is used as a control signal sample for the FLL.

Control signal samples which exceed the frequency tolerance are discarded, as well as samples resulting from excessive interval duration jitter. Surviving samples are then processed by a three-stage median filter. The signal which drives the FLL is derived from the median sample, while the average of differences between the other two samples is used as a measure of dispersion. If the dispersion is below the threshold pps_dispmax, the median is used to correct the pps_ybar value with a weight expressed as a shift PPS_AVG (2). In addition to the averaging function, pps_disp is increased by the amount pps_dispinc once each second. The result is that, should the dispersion be exceptionally high, or if the PPS signal fails for some reason, the pps_disp will eventually exceed pps_dispmax and raise an alarm.

Initially, an approximate value for pps_ybar is not known, so the duration of the calibration interval must be kept small to avoid overflowing the tick. The time difference at the end of

the calibration interval is measured. If greater than a fraction tick/4, the interval is reduced by half. If less than this fraction for four successive calibration intervals, the interval is doubled. This design automatically adapts to nominal jitter in the PPS signal, as well as the value of tick. The duration of the calibration interval is set by the pps_shift variable as a shift in powers of two. The minimum value PPS_SHIFT (2) is chosen so that with the highest CPU oscillator frequency 1024 Hz and frequency tolerance 100 ppm the tick will not overflow. The maximum value PPS_SHIFTMAX (8) is chosen such that the maximum averaging time is about 1000 s as determined by measurements of Allan variance [5].

Should the PPS signal fail, the current frequency estimate pps_ybar continues to be used, so the nominal frequency remains correct subject only to the instability of the undisciplined oscillator. The procedure to save and restore the frequency estimate works as follows. When setting the frequency from a file, the time_freq value is set as the file value minus the pps_ybar value; when retrieving the frequency, the two values are added before saving in the file. This scheme provides a seamless interface should the PPS signal fail or the kernel configuration change. Note that the frequency discipline is active whether or not the synchronization daemon is active. Since all Unix systems take some time after reboot to build a running system, usually by that time the discipline process has already settled down and the initial transients due to frequency discipline have damped out.

3.1.4. External Clock Interface

The external clock driver interface is implemented with two routines, microtime(), which returns the current clock time, and clock_set(), which furnishes the apparent system time derived from the kernel time variable. The latter routine is called only when the clock is set using the settimeofday() system call, but can be called from within the driver, such as when the year rolls over, for example.

In the stock SunOS kernel and modified Ultrix and OSF/1 kernels, the microtime() routine returns the kernel time variable plus an interpolation between timer interrupts based on the contents of a hardware counter. In the case of an external clock, such as described above, the system clock is read directly from the hardware clock registers. Examples of external clock drivers are in the tprotime.c and hightime.c routines included in the kernel.tar.Z distribution.

The external clock routines return a status code which indicates whether the clock is operating correctly and the nature of the problem, if not. The return code is interpreted by the ntp_gettime() system call, which transitions the status state machine to the TIME_ERR state if an error code is returned. This is the only error checking implemented for the external clock in the present version of the code.

The simulator has been used to check the PLL operation over the design envelope of +-512 ms in time error and +-100 ppm in frequency error. This confirms that no overflows occur and that the loop initially converges in about 15 minutes for timer interrupt rates from 50 Hz to 1024 Hz. The loop has a normal overshoot of a few percent and a final convergence time of several hours, depending on the initial time and frequency error.

3.2. Leap Seconds

It does not seem generally useful in the user application interface to provide additional details private to the kernel and synchronization protocol, such as stratum, reference identifier, reference timestamp and so forth. It would in principle be possible for the application to independently evaluate the quality of time and project into the future how long this time might be "valid." However, to do that properly would duplicate the functionality of the synchronization protocol and require knowledge of many mundane details of the platform architecture, such as the subnet configuration, reachability status and related variables. For the curious, the ntp_adjtime() system call can be used to reveal some of these mysteries.

However, the user application may need to know whether a leap second is scheduled, since this might affect interval calculations spanning the event. A leap-warning condition is determined by the synchronization protocol (if remotely synchronized), by the timecode receiver (if available), or by the operator (if awake). This condition is set by the synchronization daemon on the day the leap second is to occur (30 June or 31 December, as announced) by specifying in a ntp_adjtime() system call a clock status of either TIME_DEL, if a second is to be deleted, or TIME_INS, if a second is to be inserted. Note that, on all occasions since the inception of the leap-second scheme, there has never been a deletion occasion, nor is there likely to be one in future. If the value is TIME_DEL, the kernel adds one second to the system time immediately following second 23:59:58 and resets the clock status to TIME_OK. If the value is TIME_INS, the kernel subtracts one second from the system time immediately following second 23:59:59 and resets the clock status to TIME_OOP, in effect causing system

time to repeat second 59. Immediately following the repeated second, the kernel resets the clock status to TIME_OK.

Depending upon the system call implementation, the reported time during a leap second may repeat (with the TIME_OOP return code set to advertise that fact) or be monotonically adjusted until system time "catches up" to reported time. With the latter scheme the reported time will be correct before and shortly after the leap second (depending on the number of microtime() calls during the leap second), but freeze or slowly advance during the leap second itself. However, Most programs will probably use the ctime() library routine to convert from timeval (seconds, microseconds) format to tm format (seconds, minutes,...). If this routine is modified to use the ntp_gettime() system call and inspect the return code, it could simply report the leap second as second 60.

3.3. Clock Status State Machine

The various options possible with the system clock model described in this memorandum require a careful examination of the state transitions, status indications and recovery procedures should a crucial signal or interface fail. In this section is presented a prototype state machine designed to support leap second insertion and deletion, as well as reveal various kinds of errors in the synchronization process. The states of this machine are decoded as follows:

TIME_OK If an external clock is present, it is working properly and the system clock is derived from it. If no external clock is present, the synchronization daemon is working properly and the system clock is synchronized to a radio clock or one or more peers.

TIME_INS An insertion of one second in the system clock has been declared following the last second of the current day, but has not yet been executed.

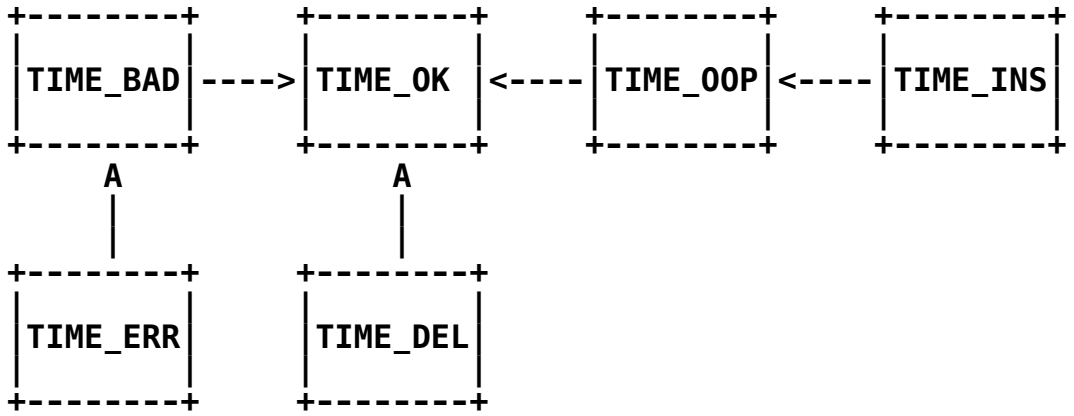
TIME_DEL A deletion of the last second of the current day has been declared, but not yet executed.

TIME_OOP An insertion of one second in the system clock has been declared following the last second of the current day. The second is in progress, but not yet completed. Library conversion routines should interpret this second as 23:59:60.

TIME_BAD Either (a) the synchronization daemon has declared the protocol is not working properly, (b) all sources of outside synchronization have been lost or (c) an external clock is present and it has just become operational following a non-operational condition.

TIME_ERR An external clock is present, but is in a non-operational condition.

In all except the TIME_ERR state the system clock is derived from either an external clock, if present, or the kernel time variable, if not. In the TIME_ERR state the external clock is present, but not working properly, so the system clock may be derived from the kernel time variable. The following diagram indicates the normal transitions of the state machine. Not all valid transitions are shown.



The state machine makes a transition once each second at an instant where the microseconds field of the kernel time variable overflows and one second is added to the seconds field. However, this condition is checked at each timer interrupt, which may not exactly coincide with the actual instant of overflow. This may lead to some interesting anomalies, such as a status indication of a leap second in progress (TIME_OOP) when actually the leap second had already expired.

The following state transitions are executed automatically by the kernel:

any state -> TIME_ERR This transition occurs when an external clock is present and an attempt is made to read it when in a non-operational condition.

TIME_INS -> TIME_OOP	This transition occurs immediately following second 86,400 of the current day when an insert-second event has been declared.
TIME_OOP -> TIME_OK	This transition occurs immediately following second 86,401 of the current day; that is, one second after entry to the TIME_OOP state.
TIME_DEL -> TIME_OK	This transition occurs immediately following second 86,399 of the current day when a delete-second event has been declared.

The following state transitions are executed by specific ntp_adjtime() system calls:

TIME_OK -> TIME_INS	This transition occurs as the result of a ntp_adjtime() system call to declare an insert-second event.
TIME_OK -> TIME_DEL	This transition occurs as the result of a ntp_adjtime() system call to declare a delete-second event.
any state -> TIME_BAD	This transition occurs as the result of a ntp_adjtime() system call to declare loss of all sources of synchronization or in other cases of error.

The following table summarizes the actions just before, during and just after a leap-second event. Each line in the table shows the UTC and NTP times at the beginning of the second. The left column shows the behavior when no leap event is to occur. In the middle column the state machine is in TIME_INS at the end of UTC second 23:59:59 and the NTP time has just reached 400. The NTP time is set back one second to 399 and the machine enters TIME_OOP. At the end of the repeated second the machine enters TIME_OK and the UTC and NTP times are again in correspondence. In the right column the state machine is in TIME_DEL at the end of UTC second 23:59:58 and the NTP time has just reached 399. The NTP time is incremented, the machine enters TIME_OK and both UTC and NTP times are again in correspondence.

No Leap		Leap Insert		Leap Delete	
UTC	NTP	UTC	NTP	UTC	NTP
23:59:58	398	23:59:58	398	23:59:58	398
23:59:59	399	23:59:59	399	00:00:00	400
00:00:00	400	23:59:60	399	00:00:01	401
00:00:01	401	00:00:00	400	00:00:02	402
00:00:02	402	00:00:01	401	00:00:03	403

To determine local midnight without fuss, the kernel code simply finds the residue of the time.tv_sec (or time.tv_sec + 1) value mod 86,400, but this requires a messy divide. Probably a better way to do this is to initialize an auxiliary counter in the settimeofday() routine using an ugly divide and increment the counter at the same time the time.tv_sec is incremented in the timer interrupt routine. For future embellishment.

4. Programming Model and Interfaces

This section describes the programming model for the synchronization daemon and user application programs. The ideas are based on suggestions from Jeff Mogul and Philip Gladstone and a similar interface designed by the latter. It is important to point out that the functionality of the original Unix adjtime() system call is preserved, so that the modified kernel will work as the unmodified one, should the new features not be in use. In this case the ntp_adjtime() system call can still be used to read and write kernel variables that might be used by a synchronization daemon other than NTP, for example.

4.1. The ntp_gettime() System Call

The syntax and semantics of the ntp_gettime() call are given in the following fragment of the timex.h header file. This file is identical, except for the SHIFT_HZ define, in the SunOS, Ultrix and OSF/1 kernel distributions. (The SHIFT_HZ define represents the logarithm to the base 2 of the clock oscillator frequency specific to each system type.) Note that the timex.h file calls the syscall.h system header file, which must be modified to define the SYS_ntp_gettime system call specific to each system type. The kernel distributions include directions on how to do this.

```
/*
 * This header file defines the Network Time Protocol (NTP)
 * interfaces for user and daemon application programs. These are
 * implemented using private system calls and data structures and
 * require specific kernel support.
 *
 * NAME
 *   ntp_gettime - NTP user application interface
 *
 * SYNOPSIS
 *   #include <sys/timex.h>
 *
 *   int system call(SYS_ntp_gettime, tptr)
 *
 *   int SYS_ntp_gettime     defined in syscall.h header file
 *   struct ntptimeval *tptr pointer to ntptimeval structure
 *
 * NTP user interface - used to read kernel clock values
 * Note: maximum error = NTP synch distance = dispersion + delay /
 * 2
 * estimated error = NTP dispersion.
 */
struct ntptimeval {
     struct timeval time;    /* current time */
     long maxerror;          /* maximum error (us) */
     long esterror;          /* estimated error (us) */
};
```

The ntp_gettime() system call returns three values in the ntptimeval structure: the current time in unix timeval format plus the maximum and estimated errors in microseconds. While the 32-bit long data type limits the error quantities to something more than an hour, in practice this is not significant, since the protocol itself will declare an unsynchronized condition well below that limit. In the NTP Version 3 specification, if the protocol computes either of these values in excess of 16 seconds, they are clamped to that value and the system clock declared unsynchronized.

Following is a detailed description of the ntptimeval structure members.

```
struct timeval time;    /* current time */
```

This member returns the current system time, expressed as a Unix timeval structure. The timeval structure consists of two 32-bit words; the first returns the number of seconds past 1 January 1970, while the second returns the number of microseconds.

```
long maxerror;          /* maximum error (us) */
```

This member returns the time_maxerror kernel variable in microseconds. See the entry for this variable in section 5 for additional information.

```
long esterror;          /* estimated error (us) */
```

This member returns the time_esterror kernel variable in microseconds. See the entry for this variable in section 5 for additional information.

4.2. The ntp_adjtime() System Call

The syntax and semantics of the ntp_adjtime() call are given in the following fragment of the timex.h header file. Note that, as in the ntp_gettime() system call, the syscall.h system header file must be modified to define the SYS_ntp_adjtime system call specific to each system type.

```
/*
 * NAME
 *   ntp_adjtime - NTP daemon application interface
 *
 * SYNOPSIS
 *   #include <sys/timex.h>
 *
 *   int system call(SYS_ntp_adjtime, mode, tptr)
 *
 *   int SYS_ntp_adjtime     defined in syscall.h header file
 *   struct timex *tptr      pointer to timex structure
 *
 * NTP daemon interface - used to discipline kernel clock
 * oscillator
 */
struct timex {
    int mode;               /* mode selector */
    long offset;            /* time offset (us) */
    long frequency;         /* frequency offset (scaled ppm) */
    long maxerror;          /* maximum error (us) */
    long esterror;          /* estimated error (us) */
    int status;             /* clock command/status */
    long time_constant;     /* pll time constant */
    long precision;         /* clock precision (us) (read only)
                             */
    long tolerance;         /* clock frequency tolerance (scaled
                             * ppm) (read only) */
    /*
     * The following read-only structure members are implemented
     * only if the PPS signal discipline is configured in the
     * kernel.
     */
    long ybar;              /* frequency estimate (scaled ppm) */
    long disp;              /* dispersion estimate (scaled ppm)
                             */
    int shift;              /* interval duration (s) (shift) */
    long calcnt;            /* calibration intervals */
    long jitcnt;            /* jitter limit exceeded */
    long discnt;            /* dispersion limit exceeded */
};
```

The ntp_adjtime() system call is used to read and write certain time-related kernel variables summarized in this and subsequent sections. Writing these variables can only be done in superuser mode. To write a variable, the mode structure member is set with one or more bits, one of which is assigned each of the following variables in turn. The current values for all variables are returned in any case; therefore, a mode argument of zero means to return these values without changing anything.

Following is a description of the timex structure members.

```
int mode;               /* mode selector */
```

This is a bit-coded variable selecting one or more structure members, with one bit assigned each member. If a bit is set, the value of the associated member variable is copied to the corresponding kernel variable; if not, the member is ignored. The bits are assigned as given in the following fragment of the timex.h header file. Note that the precision and tolerance are determined by the kernel and cannot be changed by ntp_adjtime().

```
/*
 * Mode codes (timex.mode)
 */
#define ADJ_OFFSET      0x0001    /* time offset */
#define ADJ_FREQUENCY   0x0002    /* frequency offset */
#define ADJ_MAXERROR    0x0004    /* maximum time error */
#define ADJ_ESTERROR    0x0008    /* estimated time error */
#define ADJ_STATUS      0x0010    /* clock status */
#define ADJ_TIMECONST   0x0020    /* pll time constant */
```

```
long offset;            /* time offset (us) */
```

If selected, this member replaces the value of the time_offset kernel variable in microseconds. The absolute value must be less than MAXPHASE microseconds defined in the timex.h header file. See the entry for this variable in section 5 for additional information.

If within range and the PPS signal and/or external oscillator are configured and operating properly, the clock status is automatically set to TIME_OK.

```
long time_constant;     /* pll time constant */
```

If selected, this member replaces the value of the time_constant kernel variable. The value must be between zero and MAXTC defined in the timex.h header file. See the entry for this variable in section 5 for additional information.

```
long frequency;         /* frequency offset (scaled ppm) */
```

If selected, this member replaces the value of the time_frequency kernel variable. The value is in ppm, with the integer part in the high order 16 bits and fraction in the low order 16 bits. The absolute value must be in the range less than MAXFREQ ppm defined in the timex.h header file. See the entry for this variable in section 5 for additional information.

```
long maxerror;          /* maximum error (us) */
```

If selected, this member replaces the value of the time_maxerror kernel variable in microseconds. See the entry for this variable in section 5 for additional information.

```
long esterror;          /* estimated error (us) */
```

If selected, this member replaces the value of the time_esterror kernel variable in microseconds. See the entry for this variable in section 5 for additional information.

```
int status;             /* clock command/status */
```

If selected, this member replaces the value of the time_status kernel variable. See the entry for this variable in section 5 for additional information.

In order to set this variable by ntp_adjtime(), either (a) the current clock status must be TIME_OK or (b) the member value is TIME_BAD; that is, the ntp_adjtime() call can always set the clock to the unsynchronized state or, if the clock is running correctly, can set it to any state. In any case, the ntp_adjtime() call always returns the current state in this member, so the caller can determine whether or not the request succeeded.

```
long time_constant;     /* pll time constant */
```

If selected, this member replaces the value of the time_constant kernel variable. The value must be between zero and MAXTC defined in the timex.h header file. See the entry for this variable in section 5 for additional information.

```
long precision;         /* clock precision (us) (read only) */
```

This member returns the time_precision kernel variable in microseconds. The variable can be written only by the kernel. See the entry for this variable in section 5 for additional information.

```
long tolerance;         /* clock frequency tolerance (scaled ppm)
                           */
```

This member returns the time_tolerance kernel variable in microseconds. The variable can be written only by the kernel. See the entry for this variable in section 5 for additional information.

```
long ybar;              /* frequency estimate (scaled ppm) */
```

This member returns the pps_ybar kernel variable in microseconds. The variable can be written only by the kernel. See the entry for this variable in section 5 for additional information.

```
long disp;              /* dispersion estimate (scaled ppm) */
```

This member returns the pps_disp kernel variable in microseconds. The variable can be written only by the kernel. See the entry for this variable in section 5 for additional information.

```
int shift;              /* interval duration (s) (shift) */
```

This member returns the pps_shift kernel variable in microseconds. The variable can be written only by the kernel. See the entry for this variable in section 5 for additional information.

```
long calcnt;            /* calibration intervals */
```

This member returns the pps_calcnt kernel variable in microseconds. The variable can be written only by the kernel. See the entry for this variable in section 5 for additional information.

```
long jitcnt;            /* jitter limit exceeded */
```

This member returns the pps_jittcnt kernel variable in microseconds. The variable can be written only by the kernel. See the entry for this variable in section 5 for additional information.

```
long discnt;            /* dispersion limit exceeded */
```

This member returns the pps_discnt kernel variable in microseconds. The variable can be written only by the kernel. See the entry for this variable in section 5 for additional information.

4.3. Command/Status Codes

The kernel routines use the system clock status variable time_status, which records whether the clock is synchronized, waiting for a leap second, etc. The value of this variable is returned as the result code by both the ntp_gettime() and ntp_adjtime() system calls. In addition, it can be explicitly read and written using the ntp_adjtime() system call, but can be written only in superuser mode. Values presently defined in the timex.h header file are as follows:

```
/*
 * Clock command/status codes (timex.status)
 */
#define TIME_OK    0    /* clock synchronized */
#define TIME_INS   1    /* insert leap second */
#define TIME_DEL   2    /* delete leap second */
#define TIME_OOP   3    /* leap second in progress */
#define TIME_BAD   4    /* kernel clock not synchronized */
#define TIME_ERR   5    /* external oscillator not
                           synchronized */
```

A detailed description of these codes as used by the leap-second state machine is given later in this memorandum. In case of a negative result code, the kernel has intercepted an invalid address or (in case of the ntp_adjtime() system call), a superuser violation.

5. Kernel Variables

This section contains a list of kernel variables and a detailed description of their function, initial value, scaling and limits.

5.1. Interface Variables

The following variables are read and set by the ntp_adjtime() system call. Additional automatic variables are used as temporaries as described in the code fragments.

```
int time_status = TIME_BAD;
```

This variable controls the state machine used to insert or delete leap seconds and show the status of the timekeeping system, PPS signal and external oscillator, if configured.

```
long time_offset = 0;
```

This variable is used by the PLL to adjust the system time in small increments. It is scaled by (1 << SHIFT_UPDATE) (12) in microseconds. The maximum value that can be represented is about +-512 ms and the minimum value or precision is a few parts in 10^10 s.

```
long time_constant = 0;      /* pll time constant */
```

This variable determines the bandwidth or "stiffness" of the PLL. The value is used as a shift between zero and MAXTC (6), with the effective PLL time constant equal to a multiple of (1 << time_constant) in seconds. For room-temperature quartz oscillator the recommended default value is 2, which corresponds to a PLL time constant of about 900 s and a maximum update interval of about 64 s. The maximum update interval scales directly with the time constant, so that at the maximum time constant of 6, the update interval can be as large as 1024 s.

Values of time_constant between zero and 2 can be used if quick convergence is necessary; values between 2 and 6 can be used to reduce network load, but at a modest cost in accuracy. Values above 6 are appropriate only if an external oscillator is present.

```
long time_tolerance = MAXFREQ; /* frequency tolerance (ppm) */
```

This variable represents the maximum frequency error or tolerance in ppm of the particular CPU clock oscillator and is a property of the architecture; however, in principle it could change as result of the presence of external discipline signals, for instance. It is expressed as a positive number greater than zero in parts-per-million (ppm).

The recommended value of MAXFREQ is 200 ppm is appropriate for room-temperature quartz oscillators used in typical workstations. However, it can change due to the operating condition of the PPS signal and/or external oscillator. With either the PPS signal or external oscillator, the recommended value for MAXFREQ is 100 ppm.

```
long time_precision = 1000000 / HZ; /* clock precision (us) */
```

This variable represents the maximum error in reading the system clock in microseconds. It is usually based on the number of microseconds between timer interrupts, 10000 us for the SunOS kernel, 3906 us for the Ultrix kernel, 976 us for the OSF/1 kernel. However, in cases where the time can be interpolated between timer interrupts with microsecond resolution, such as in the unmodified SunOS kernel and modified Ultrix and OSF/1 kernels, the precision is specified as 1 us. In cases where a PPS signal or external oscillator is available, the precision can depend on the operating condition of the signal or oscillator. This variable is determined by the kernel for use by the synchronization daemon, but is otherwise not used by the kernel.

```
long time_maxerror = MAXPHASE; /* maximum error */
```

This variable establishes the maximum error of the indicated time relative to the primary synchronization source in microseconds. For NTP, the value is initialized by a ntp_adjtime() call to the synchronization distance, which is equal to the root dispersion plus one-half the root delay. It is increased by a small amount (time_tolerance) each second to reflect the clock frequency tolerance. This variable is computed by the synchronization daemon and the kernel, but is otherwise not used by the kernel.

```
long time_esterror = MAXPHASE; /* estimated error */
```

This variable establishes the expected error of the indicated time relative to the primary synchronization source in microseconds. For NTP, the value is determined as the root dispersion, which represents the best estimate of the actual error of the system clock based on its past behavior, together with observations of multiple clocks within the peer group. This variable is computed by the synchronization daemon and returned in system calls, but is otherwise not used by the kernel.

5.2. Phase-Lock Loop Variables

The following variables establish the state of the PLL and the residual time and frequency offset of the system clock. Additional automatic variables are used as temporaries as described in the code fragments.

```
long time_phase = 0;          /* phase offset (scaled us) */
```

The time_phase variable represents the phase of the kernel time variable at each tick of the clock. This variable is scaled by (1 << SHIFT_SCALE) (23) in microseconds, giving a maximum adjustment of about +-256 us/tick and a resolution less than one part in 10^12.

```
long time_offset = 0;         /* time offset (scaled us) */
```

The time_offset variable represents the time offset of the CPU clock oscillator. It is recalculated as each update to the system clock is received via the hardupdate() routine and at each second in the seconds_overflow routine. This variable is scaled by (1 << SHIFT_UPDATE) (12) in microseconds, giving a maximum adjustment of about +-512 ms and a resolution of a few parts in 10^10 s.

```
long time_freq = 0;           /* frequency offset (scaled ppm) */
```

The time_freq variable represents the frequency offset of the CPU clock oscillator. It is recalculated as each update to the system clock is received via the hardupdate() routine. It can also be set via ntp_adjtime() from a value stored in a file when the synchronization daemon is first started. It can be retrieved via ntp_adjtime() and written to the file about once per hour by the daemon. The time_freq variable is scaled by (1 << SHIFT_KF) (16) ppm, giving it a maximum value well in excess of the limit of +-256 ppm imposed by other constraints. The precision of this representation (frequency resolution) is parts in 10^11, which is adequate for all but the best external oscillators.

```
time_adj = 0;                 /* tick adjust (scaled 1 / HZ) */
```

The time_adj variable is the adjustment added to the value of tick at each timer interrupt. It is computed once each second from the time_offset, time_freq and, if the PPS signal is present, the ps_ybar variable once each second.

```
long time_reftime = 0;       /* time at last adjustment (s) */
```

This variable is the seconds portion of the system time on the last update received by the hardupdate() routine. It is used to compute the time_freq variable as the time since the last update increases.

```
int fixtick = 1000000 % HZ;  /* amortization factor */
```

In the Ultrix and OSF/1 kernels, the interval between timer interrupts does not evenly divide the number of microseconds in the second. In order that the clock runs at a precise rate, it is necessary to introduce an amortization factor into the local timescale. In the original Unix code, the value of fixtick is amortized once each second, introducing an additional source of jitter; in the new model the value is amortized at each tick of the system clock, reducing the jitter by the reciprocal of the clock oscillator frequency. This is not a new kernel variable, but a new use of an existing kernel variable.

5.3. Pulse-per-second (PPS) Frequency-Lock Loop Variables

The following variables are used only if a pulse-per-second (PPS) signal is available and connected via a modem-control lead, such as produced by the optional ppsclock feature incorporated in the serial port driver. They establish the design parameters of the PPS frequency-lock loop used to discipline the CPU clock oscillator to an external PPS signal. Additional automatic variables are used as temporaries as described in the code fragments.

```
long pps_usec;          /* microseconds at last pps */
```

The pps_usec variable is latched from a high resolution counter or external oscillator at each PPS interrupt. In determining this value, only the hardware counter contents are used, not the contents plus the kernel time variable, as returned by the microtime() routine.

```
long pps_ybar = 0;      /* pps frequency offset estimate */
```

The pps_ybar variable is the average CPU clock oscillator frequency offset relative to the PPS disciplining signal. It is scaled in the same units as the time_freq variable.

```
pps_disp = MAXFREQ;     /* dispersion estimate (scaled ppm) */
```

The pps_disp variable represents the average sample dispersion measured over the last three samples. It is scaled in the same units as the time_freq variable.

```
pps_dispmax = MAXFREQ / 2; /* dispersion threshold */
```

The pps_dispmax variable is used as a dispersion threshold. If pps_disp is less than this threshold, the median sample is used to update the pps_ybar estimate; if not, the sample is discarded.

```
pps_dispinc = MAXFREQ >> (PPS_SHIFT + 4); /* pps dispersion
increment/sec */
```

The pps_dispinc variable is the increment to add to pps_disp once each second. It is computed such that, if no PPS samples have arrived for several calibration intervals, the value of pps_disp will exceed the pps_dispmax threshold and raise an alarm.

```
int pps_mf[] = {0, 0, 0};    /* pps median filter */
```

The pps-mf[] array is used as a median filter to detect and discard jitter in the PPS signal.

```
int pps_count = 0;           /* pps calibrate interval counter */
```

The pps_count variable measures the length of the calibration interval used to calculate the frequency. It normally counts from zero to the value 1 << pps_shift.

```
pps_shift = PPS_SHIFT;       /* interval duration (s) (shift) */
```

The pps_shift variable determines the duration of the calibration interval, 1 << pps_shift s.

```
pps_intcnt = 0;              /* intervals at current duration */
```

The pps_intcnt variable counts the number of calibration intervals at the current interval duration. It is reset to zero after four intervals and when the interval duration is changed.

```
long pps_calcnt = 0;         /* calibration intervals */
```

The pps_calcnt variable counts the number of calibration intervals.

```
long pps_jitcnt = 0;          /* jitter limit exceeded */
```

The pps_jitcnt variable counts the number of resets due to excessive jitter or frequency offset. These resets are usually due to excessive noise in the PPS signal or interface.

```
long pps_discnt = 0;          /* dispersion limit exceeded */
```

The pps_discnt variable counts the number of calibration intervals where the dispersion is above the pps_dispmax limit. These resets are usually due to excessive frequency wander in the PPS signal source.

5.4. External Oscillator Variables

The following variables are used only if an external oscillator (HIGHBALL or TPRO) is present. Additional automatic variables are used as temporaries as described in the code fragments.

```
int clock_count = 0;          /* CPU clock counter */
```

The clock_count variable counts the seconds between adjustments to the kernel time variable to discipline it to the external clock.

```
struct timeval clock_offset; /* HIGHBALL clock offset */
```

The clock_offset variable defines the offset between system time and the HIGHBALL counters.

```
long clock_cpu = 0;           /* CPU clock adjust */
```

The clock_cpu variable contains the offset between the system clock and the HIGHBALL clock for use in disciplining the kernel time variable.

6. Architecture Constants

Following is a list of the important architecture constants that establish the response and stability of the PLL and provide maximum bounds on behavior in order to satisfy correctness assertions made in the protocol specification. Additional definitions are given in the timex.h header file.

6.1. Phase-lock loop (PLL) definitions

The following defines establish the performance envelope of the PLL. They establish the maximum phase error (MAXPHASE), maximum frequency error (MAXFREQ), minimum interval between updates (MINSEC) and maximum interval between updates (MAXSEC). The intent of these bounds is to force the PLL to operate within predefined limits in order to satisfy correctness assertions of the synchronization protocol. An excursion which exceeds these bounds is clamped to the bound and operation proceeds normally. In practice, this can occur only if something has failed or is operating out of tolerance, but otherwise the PLL continues to operate in a stable mode.

MAXPHASE must be set greater than or equal to CLOCK.MAX (128 ms), as defined in the NTP specification. CLOCK.MAX establishes the maximum time offset allowed before the system time is reset,

rather than incrementally adjusted. Here, the maximum offset is clamped to MAXPHASE only in order to prevent overflow errors due to defective programming.

MAXFREQ reflects the manufacturing frequency tolerance of the CPU oscillator plus the maximum slew rate allowed by the protocol. It should be set to at least the intrinsic frequency tolerance of the oscillator plus 100 ppm for vernier frequency adjustments. If the kernel frequency discipline code is installed (PPS_SYNC), the CPU oscillator frequency is disciplined to an external source, presumably with negligible frequency error.

```
#define MAXPHASE 512000      /* max phase error (us) */
#ifdef PPS_SYNC
#define MAXFREQ 100          /* max frequency error (ppm) */
#else
#define MAXFREQ 200          /* max frequency error (ppm) */
#endif /* PPS_SYNC */
#define MINSEC 16            /* min interval between updates (s)
                                */
#define MAXSEC 1200          /* max interval between updates (s)
                                */
```

6.2. Pulse-per-second (PPS) Frequency-lock Loop (FLL) Definitions

The following defines and declarations are used only if a pulse-per-second (PPS) signal is available and connected via a modem-control lead, such as produced by the optional ppsclock feature incorporated in the serial port driver. They establish the design parameters of the frequency-lock loop (FLL) used to discipline the CPU clock oscillator to the PPS oscillator.

PPS_AVG is the averaging constant used to update the FLL from frequency samples measured for each calibration interval. PPS_SHIFT and PPS_SHIFTMAX are the minimum and maximem, respectively, of the calibration interval represented as a power of two. The PPS_DISPINC is the initial increment to pps_disp at each second.

```
#define PPS_AVG 2            /* pps averaging constant (shift) */
#define PPS_SHIFT 2          /* min interval duration (s) (shift)
                                */
#define PPS_SHIFTMAX 6       /* max interval duration (s) (shift)
                                */
#define PPS_DISPINC 0        /* dispersion increment (us/s) */
```

6.3. External Oscillator Definitions

The following definitions and declarations are used only if an external oscillator (HIGHBALL or TPRO) is configured on the system.

```
#define CLOCK_INTERVAL 30    /* CPU clock update interval (s) */
```

7. References

[1] Mills, D., "Internet time synchronization: the Network Time Protocol", IEEE Trans. Communications COM-39, 10 (October 1991), 1482- 1493. Also in: Yang, Z., and T.A. Marsland (Eds.). Global States and Time in Distributed Systems, IEEE Press, Los Alamitos, CA, 91-102.

[2] Mills, D., "Network Time Protocol (Version 3) specification, implementation and analysis", RFC 1305, University of Delaware, March 1992, 113 pp.

[3] Mills, D., "Modelling and analysis of computer network clocks", Electrical Engineering Department Report 92-5-2, University of Delaware, May 1992, 29 pp.

[4] Mills, D., "Simple Network Time Protocol (SNTP)", RFC 1361, University of Delaware, August 1992, 10 pp.

[5] Mills, D., "Precision synchronizatin of computer network clocks", Electrical Engineering Department Report 93-11-1, University of Delaware, November 1993, 66 pp.

Security Considerations

Security issues are not discussed in this memo.

Author's Address

David L. Mills
Electrical Engineering Department
University of Delaware
Newark, DE 19716

Phone: (302) 831-8247
EMail: mills@udel.edu