1. Introduction

This report 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 computers using the Network Time Protocol (NTP) or similar time synchronization protocol. This report describes the design principles and implementations of the model, while related technical reports discuss the design approach, engineering analysis and performance evaluation of the model as implemented in Unix kernels for modern 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 report, 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 μ s (stock Sun kernels already do provide this 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 that operate upon them 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 report, 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, these 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 [LEV89], 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 computer. As described in [MIL91], 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 report. The goal in the enterprise is the most accurate and stable system clock possible with the available computer hardware and kernel software.

In order to understand how the new model works, it is useful to review how most Unix kernels maintain the system clock. 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 kernel adds 64 us 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 timer interrupts. 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 larger. 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 consistent with the model, 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 over 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 a one 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 this variable and overflows propagated as required. The new model operates in two modes, depending on the interval between updates. At intervals less than about 1024 s, it operates as an adaptive-parameter, first-order, type-II phase-lock loop (PLL) as described in [MIL92b]. However, this type of discipline is not suitable for update intervals greater than 1024 s.

The second mode is appropriate for update intervals greater than 1024 s, as used in the Automatic Computer Time Service (ACTS), a telephone time system operated by NIST. In this mode the discipline operates as a hybrid phase/frequency-lock loop (FLL), in which the frequency is estimated directly, rather than inferred from phase observations. In principle, the hybrid PLL/FLL 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.

In the original NTP design, the software daemon *xntpd* 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/FLL in the kernel using precision time and frequency representations, 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 clock runs true even if the daemon were to cease operation or the network paths to the primary synchronization source fail.

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 [MIL92b], 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 [MIL92b]. In the original 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 representations, 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 clock 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 clock 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 clock 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 backwards. 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 and time constant depend on the measured delay and dispersion; 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 the 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 error 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 a new system call ntp_adjtime(), 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 indicated maximum error to grow by an amount equal to the maximum 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. As shown in [MIL93], this is near the optimum interval for NTP used with ordinary room-temperature quartz oscillators. 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 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 system 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 oscillator can be avoided.

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

The stock microtime() routine in the Ultrix kernel for Digital Equipment MIPS-based workstations 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 and 175.000 MHz for the 3000/400, for example. The new microtime() routine for the OSF/1 kernel automatically determines the counter rate and uses it in the same fashion as the Ultrix routine. Support for this feature is conditionally compiled in the kernel only if the MICRO option is used in the kernel configuration file.

In both the Ultrix and OSF/1 kernels the gettimeofday() and ntp_gettime() system call use the new microtime() routine, which returns the interpolated value to 1-us resolution, 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 on a SPARCstation IPC. 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. Note that, as explained later, this routine should be called at an interrupt priority level not greater than that of the timer interrupt routine. Otherwise, it is possible to miss a tick increment, with result the time returned can be late by one tick. This is always true in the case of gettimeofday() and ntp_gettime(), but might not be true in other cases, such as when using the PPS signal described later in this report.

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 CPU clock oscillator of the primary server. As discussed in [MIL93], the traditional interface using a ASCII serial timecode and RS232 port precludes the full accuracy of most radio clocks. In addition, the poor frequency 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 interface and 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 [MIL93]. In the NTP Version 3 daemon *xntpd*, the PPS signal can be used to augment the less precise ASCII serial timecode to improve accuracy to the order of a few tens of microseconds. Support is also included in the NTP distribution for the TPRO interface, as well as the audio codec; however, the latter requires a modified kernel audio driver contained in the compressed tar archive bsd_audio.tar.Z in the same host and directory as the NTP Version 3 distribution mentioned previously.

2.4.1. PPS Signal

The most convenient way to interface a PPS signal to a computer is usually with a serial port and RS232-compatible signal; however, the PPS signal produced by most radio clocks and laboratory instruments is usually a TTL pulse signal. Therefore, some kind of level converter/pulse generator is necessary to adapt the PPS signal to a serial port. An example design, including schematic and printed-circuit board artwork, is in the compressed tar archive gadget.tar.Z in the same host and directory as the NTP Version 3 distribution mentioned previously. There are several ways the PPS signal can be used in conjunction with the NTP Version 3 daemon *xntpd*, as described in [MIL93] and in the documentation included in the distribution.

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 synchroni-

zation 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 in [MIL93], 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. In addition, the Ultrix 4.3 kernel has been modified to provide a similar functionality. The hardpps() routine compares the timestamp with a sample of the CPU clock oscillator in order to discipline the oscillator to the time and frequency of the PPS signal. Using this method, the time accuracy is improved to typically 20 us or less and frequency stability 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 some kernels which is very difficult to fix. The serial port interrupt routine often operates at an interrupt priority level above the timer interrupt routine. Thus, as explained below, it is possible that a tick increment can be missed and the time returned late by one tick. It may happen that, if the CPU clock oscillator frequency is close to the PPS oscillator frequency (less than a few ppm), this condition can persist for two or more successive PPS interrupts. A useful workaround in the code is to use a glitch detector and median filter to process the PPS sample offsets. The glitch detector suppresses offset bursts greater than half the tick interval and which last less than 30 successive PPS interrupts. The median filter ranks the offsets in a moving window of three samples and uses the median as the output and the difference between the other two as a dispersion measure.

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 technical information 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 been made to obtain the year from the kernel time variable and initialize the driver accordingly. Support for this feature is conditionally compiled in the kernel only if the EXT_CLOCK and TPRO options are 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 time difference between the microtime() clock and kernel time variable and using it 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 practical 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 called 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 technical information distribution mentioned previously. Support for this feature is conditionally compiled in the kernel 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 [MIL92b] and [MIL93]. Further details are given in the technical summary later in this report.

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 technical information distribution mentioned previously. In addition, an example kernel module kern_ntptime.c which implements the ntp_gettime() and ntp_adjtime() system calls is included. Neither of these programs incorporate licensed code. Since the distribution is somewhat large, due to copious comments and ornamentation, it is impractical to include a listing of these programs in this report. 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 report.

In the kern.c program, the system clock is implemented using a set of variables and algorithms defined in the simulator and driven by explicit offsets generated by the main() routine 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/FLL Simulation

The simulator kern.c operates in conformance with the analytical models described in [MIL92b] and [MIL94]. The main() program operates as a driver for the routines hardupdate(), hardpps() and microtime(), and the code fragments hardclock and second_overflow, although not all functions implemented in these routines and fragments are simulated. The program simulates the selected mode, PLL or FLL, at each timer interrupt and prints a summary of critical program variables at each time update. The simulator is not fancy; in its present form, the mode and various operational features are selected by changing defines and in some cases the code itself and rebuilding the program.

There are four 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 can be used to discipline the time and frequency of the CPU clock oscillator. The EXT_CLOCK option provides support for an external kernel-readable clock. External clocks are implemented as various versions of the microtime() clock driver, with the specific driver selected by an option in the kernel configuration file. The TPRO option selects the KSI/Odetics TPRO IRIG interface for the SBus, while the HIGHBALL option selects the HIGHBALL precision oscillator interface for the SBus.

The kernel code can operate in various modes and with various features enabled or disabled, as selected by the ntp_adjtime() system call, which is not simulated here. The bits of the time_status variable are used to control these functions and record error conditions as they exist. The programming interface is described later in this report. In addition, the PPS signal is carefully monitored for error conditions which can affect accuracy, stability and reliability.

In following sections the operation of each routine and code fragment is described in exhaustive detail. The intent is not only to describe how the algorithms work, but also to demonstrate assertions on the ability of the algorithms to work correctly over the entire range of input variables and algorithm states. Although the most demanding proofs involve machines with an intrinsic word size of 32 bits, such as those using 32-bit SPARC and MIPS processors, the assertion is made at the outset that the proofs apply equally to machines with larger word sizes, including the 64-bit Alpha processor. It should also be pointed out at the outset that the routines and fragments are in fact virtually identical to those used in the SunOS, Ultrix and OSF/1 kernels mentioned previously. From all available evidence, their operations in the simulator and the actual machine are identical.

3.2. The hardupdate() Routine

The hardupdate() routine is called by the ntp_adjtime() system call to adjust the system clock phase and frequency. The offset variable is passed in the hardupdate() argument. The phase adjustment time_offset is computed as offset scaled by SHIFT_UPDATE (12), which is sufficient to protect the low-order bits in later operations, and leaves 20 bits on a 32-bit machine to represent the phase

adjustment in microsecond units. The result is clamped to a maximum MAXPHASE that can be defined as high as 512 ms, but in practice is much lower at about 128 ms.

The current frequency is represented by the variable time_freq. This variable is scaled by SHIFT_USEC (16), which leaves 16 bits on a 32-bit machine to represent the frequency in ppm units. The method of frequency adjustment depends on whether the PLL or FLL mode is selected (by the STA_FLL bit in the status word). In FLL mode, the adjustment is calculated directly from offset and the time since last update. The result updates time_freq, which in this mode is exponentially averaged with time constant SHIFT_KH (2). In PLL mode, the adjustment is calculated as the product of offset and time since last update divided by the frequency gain factor SHIFT_KF (16) and the square of the time constant time_constant. The result is added directly to time_freq. Note that gain factors and time constants are powers of two, so that most multiply/divide operations can be done by simple shifts. In either mode, time_freq is clamped not to exceed the frequency tolerance MAXFREQ, which can be defined as high as 512 ppm, but usually is much lower in the order of 300 ppm. Note that all shifts are assumed to be positive and that a shift of a signed quantity to the right requires a little dance.

It is necessary to carefully evaluate the possibility of overflow and loss of significance in the above operations, especially in the case of 32-bit machines. The signed offset provided by hardclock() is clamped upon entry not to exceed MAXPHASE (512000) or 20 bits (including sign) and shifted left by SHIFT_UPDATE (12) bits. The resulting value of time_offset cannot overflow a 32-bit word. In FLL mode, time_offset is used directly; in PLL mode, time_offset is shifted right SHIFT_KG (6) plus time_constant in bits. Since time_constant is positive and limited to MAXTC (6), no significance is lost in the process.

In PLL mode, the interval since last update cannot exceed MAXSEC (1024). Thus, the intermediate product time_offset times this interval requires at most 20 bits. Since time_constant is greater than or equal to zero and SHIFT_KF (16) is greater than or equal to SHIFT_USEC (16), the shift is always to the right. The maximum value of the adjustment cannot overflow a 32-bit word and the clamp involving time_tolerance cannot produce anomalous results. In FLL mode, the interval since last update is at least MINSEC (16). The intermediate term, time_offset divided by this interval, requires only 16 bits; therefore, this quantity left shifted by SHIFT_USEC (16) cannot overflow a 32 bit word.

The STA_PLL, STA_FLL and STA_PPSTIME status bits, which are set by the ntp_adjtime() system call, serve to enable or inhibit the kernel PLL/FLL and PPS time-discipline functions. The STA_PPSSIGNAL status bit is set by the hardpps() code fragment when the PPS signal is present and operating within nominal bounds. Time discipline from the PPS signal operates only if both the STA_PPSTIME and STA_PPSSIGNAL bits are set; otherwise, the discipline operates from the offset given in the ntp_adjtime() system call. In the intended mode of operation, the synchronization daemon sets STA_PLL to enable the PLL when first initialized, then sets STA_PPSTIME when reliable synchronization to within MAXPHASE has been achieved with either a radio clock or external peer. The daemon can detect and indicate this condition for monitoring purposes by noting that both STA_PPSTIME and STA_PPSSIGNAL are set.

3.2.1. The hardclock Fragment

The hardclock fragment is inserted in the timer interrupt routine at the point the system clock is to be incremented by the timer interrupt interval, or tick. Previous to this fragment the time_update

variable has been initialized to the value of tick, the value of which depends on the particular kernel. Optionally, the stock Unix adjtime() system call can be used to augment time_update by the (signed) value of the kernel variable tickadj, which is usually in the order of 5 us. This adjustment is outside the PLL/FLL discipline loop and therefore does not affect the system clock frequency. The adjtime() code to do this is part of the licensed Unix kernel and not normally used with the kernel modifications described here. However, this feature allows the stock Unix functionality to be preserved when the modified kernel functions are not in use.

Regardless of whether the adjtime() or ntp_adjtime() system calls are in use, 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 units scaled by SHIFT_SCALE (22), time_update is increased by the (signed) excess and time_phase retains the residue.

In those cases where a PPS signal is connected by a serial port operating at an interrupt priority level greater than the timer interrupt, special consideration should be given the location of the hardclock fragment in the timer interrupt routine. The system clock should be advanced as early in the routine as possible, preferably before the hardware timer interrupt flag is cleared. This reduces or eliminates the possibility that the microtime() routine may latch the time after the flag is cleared, but before the system clock is advanced, which results in a returned time late by one tick.

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.

3.2.2. The second_overflow Fragment

The second_overflow fragment is inserted in the timer interrupt routine at the point after the hardclock fragment, where the microseconds field of the system time variable has been incremented, and then checked if greater than 1000000 (one second). If not, the second_overflow fragment has no effect. If true, this fragment first runs the leap-second state machine described below. Then, the maximum error time_maxerror is increased by time_tolerance. This represents the increment necessary to satisfy correctness assertions described in the specification, but is otherwise not used by the kernel.

Next, the increment time_adj to advance the kernel time variable at each timer interrupt is calculated from the phase (time_offset) and frequency (time_freq) variables previously computed by the hardclock fragment. In FLL mode, the phase increment is equal to the value of time_offset itself; while, in PLL mode, the increment is equal to the value of time_offset divided by the product of the phase gain factor SHIFT_KG (6) times time_constant. In either case, the phase increment is clamped so as not to exceed the maximum slew rate, which occurs at the maximum offset MAXPHASE (512000) divided by the minimum update interval MINSEC (16) and scaled by SHIFT_UPDATE (12). The result requires no more than 28 bits, so cannot overflow a 32-bit word.

The actual phase adjustment is the increment calculated as above, which is then subtracted from time_offset, yielding a residual to be incorporated at the next seconds overflow. This technique provides a rapid convergence for large adjustments, together with good resolution for small ones. In FLL mode, the maximum slew rate clamp above insures that the phase correction rate is not larger

than that necessary to amortize the entire phase correction of as much as 512 ms before the next update. While in PLL mode the adjustments may not be entirely amortized by the time of the next update, the affect on loop stability and accuracy is very small.

Finally, the fraction point of the phase increment is aligned to SHIFT_SCALE (22), which requires a left shift of SHIFT_SCALE minus SHIFT_UPDATE (12) or 10 bits, and divided by the hardware timer frequency, represented as a right shift of SHIFT_HZ (7, 8 or 10, depending on the kernel) bits. In order to prevent overflow, the shifts are combined in one operation, which results in a net left shift of no more than 3 bits, and means the 28-bit result cannot overflow a 32-bit word. A safe shift requires SHIFT_SCALE be no less than the sum of SHIFT_UPDATE plus SHIFT_HZ, which is the case for SHIFT_HZ values up to 10 (1024 Hz). For interval timer frequencies greater than 1024 Hz, the various shifts will have to be adjusted; however, this is likely only for machines with word sizes greater than 32 bits, which considerably simplifies the analysis. As a matter of interest, the lower limit of SHIFT_HZ can be made as small as 6 (32 Hz) without overflow, should that ever be useful.

In both PLL and FLL modes, the clock frequency offset time_freq has already been calculated by the hardupdate() routine. The system clock frequency is maintained by adding time_freq to time_adj once each second. First, the fraction point of time_freq is aligned to SHIFT_SCALE (22), which requires a left shift of SHIFT_SCALE minus SHIFT_USEC or 6 bits. The result then must be divided by the hardware clock frequency as above, which results in a net right shift. As above, the shifts are combined in one operation, so the result cannot overflow a 32-bit word. Note that, in the case the tick does not exactly divide the second in microseconds, an auxiliary variable fixtick is used to trim the frequency to account for the remainder. The sum of the phase and frequency contributions is then divided by the number of timer ticks per second, which becomes the final value of time_adj.

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 discipline the CPU clock oscillator frequency as described below. As long as the actual oscillator frequency is a power of two in Hz, 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.

To complete the analysis, the above operations can be seen to conserve the microsecond resolution provided in the hardupdate() routine argument. The resolution of the frequency variable time_freq is 16 bits in ppm, which is comparable to the stability of a cesium oscillator. The resolution of the resolution of the phase variable time_adj depends on the timer frequency and decreases as the frequency increases. At a frequency of 1024 Hz, for example, the resolution is 12 bits in ppm, which is less than even the best temperature-stabilized quartz crystal oscillator.

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.

3.2.3. 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 code operates as a first-order, type-I, frequency-lock loop (FLL) controlled by the difference between the frequency represented by the pps_freq variable and the

frequency of the hardware clock oscillator. It also provides offsets to the hardupdate() fragment in order to discipline the system clock time.

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 are determined by saving the microseconds field of the system time, calling the microtime() routine, and subtracting the saved value. If a microseconds overflow has occurred during the process, the resulting microseconds value will be negative, in which case the caller adds 1000000 to normalize the microseconds field.

In order to avoid large jitter when the PPS interrupt occurs during the timer interrupt routine before the system clock is advanced, a glitch detector is used. The detector latches when an offset exceeds a threshold tick/2 and stays latched until either a subsequent offset is less than the threshold or a specified interval MAXGLITCH (30 s) has elapsed. As long as the detector remains latched, it outputs the offset immediately preceding the latch, rather than the one received.

A three-stage median filter is used to suppress jitter less than the glitch threshold. The median sample drives the PLL, while the difference between the other two samples represents the time dispersion. Time dispersion samples are averaged and used as a jitter estimate. If this estimate exceeds a threshold MAXTIME/2 (100 us), an error bit STA_PPSJITTER is raised in the status word.

The frequency of the hardware oscillator is 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_freq 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 the control signal for the FLL.

Control signal samples which exceed the frequency tolerance MAXFREQ (100 ppm) are discarded, as well as samples resulting from excessive interval duration jitter. In these cases an error bit STA_PPSERROR is raised in the status word. Surviving samples are then processed by a three-stage median filter. The median sample drives the FLL, while the difference between the other two samples represents the frequency dispersion. Frequency dispersion samples are averaged and used as a stability estimate. If this estimate is below a threshold MAXFREQ/4 (25 ppm), the median sample is used to correct the oscillator frequency pps_freq with a weight expressed as a shift PPS_AVG (2).

Initially, an approximate value for the oscillator frequency 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 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 [MIL93].

Should the PPS signal fail, the current frequency estimate pps_freq 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_freq 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.

3.2.4. 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.

3.3. 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 status bit of either STA_DEL, if a second is to be deleted, or STA_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, nor is there likely to be one in future. If the bit is STA_DEL, the kernel adds one second to the system time immediately following second 23:59:58 and resets the clock state to TIME_WAIT. If the bit is STA_INS, the kernel subtracts one second from the system time immediately following second 23:59:59 and resets the clock state 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_WAIT.

Following the leap operations, the clock remains in the TIME_WAIT state until both the STA_DEL and STA_INS status bits are reset. This provides both an unambiguous indication that a leap recently occurred, as well as time for the daemon or operator to clear the warning condition.

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.

3.3.1. Clock Status State Machine

The various options possible with the system clock model described in this report 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 a PPS signal or external clock is present, it is working properly and the system clock is derived from it. If not, 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_WAIT

The scheduled leap event has occurred, but the STA_DEL and STA_INS status bits have not yet been cleared.

TIME_ERROR

Either (a) the synchronization daemon has declared the protocol is not working properly, (b) all sources of outside synchronization have been lost or (c) a PPS signal or external clock is present, but not working properly.

In all states the system clock is derived from either a PPS signal or external clock, if present, or the kernel time variable, if not. If a PPS error condition is recognized, the PPS signal is disabled and ntp_adjtime() updates are used instead. If an external clock error condition is recognized, the external clock is disabled and the kernel time variable is used instead.

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 when the timer overflows, which may not coincide with the actual seconds increment. This may lead to some interesting anomalies, such as a status indication of a leap second in progress (TIME_OOP) when the leap second has already expired. This ambiguity is unavoidable, unless the timer interrupt is made synchronous with the system clock.

The following state transitions are executed automatically by the kernel at rollover of the microseconds field:

any state → TIME_ERROR

This transition occurs when an error condition is recognized and continues as long as the condition persists. The error indication overrides the normal state indication, but does not affect the actual clock state. Therefore, when the condition is cleared, the normal state indication resumes.

TIME_OK → TIME_DEL

This transition occurs if the STA_DEL bit is set in the status word.

TIME_OK → TIME_INS

This transition occurs if the STA_INS bit is set in the status word.

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_WAIT

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_WAIT

This transition occurs immediately following second 86,399 of the current day when a delete-second event has been declared.

Standard		Insert		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

Table 1. Comparison of UTC and NTP Timescales at Leap

TIME_WAIT → TIME_OK

This transition occurs when the STA_DEL and STA_INS bits are cleared by an ntp_adjtime() call.table summarizes the actions just before, during and just after a leap-second event.

Each line in Table 1 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.

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.

The kernel routines use the clock state variable time_state, 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. It is set implicitly by the STA_DEL and STA_INS status bits, as described previously. Values presently defined in the timex.h header file are as follows:

TIME_OK	0	no leap second warning
TIME_INS	1	insert leap second warning

TIME_DEL	2	delete leap second warning
TIME_OOP	3	leap second in progress
TIME_WAIT	4	leap second has occurred
TIME_ERROR	5	clock not synchronized

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.

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 ys/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 (ro) */
        long maxerror;                /* maximum error (us) (ro) */
        long esterror;                /* estimated error (us) (ro) */
};
```

The ntp_gettime() system call returns three read-only (ro) 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 (ro)

> This member is the current system time expressed as a Unix timeval structure. The timeval structure consists of two 32-bit words; the first is the number of seconds past 1 January 1970 assuming no intervening leap-second insertions or deletions, while the second is the number of microseconds within the second.

long maxerror (ro)

> This member is the value of the time_maxerror kernel variable, which represents 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 maximum clock frequency error. This variable is provided bu a ntp-adjtime() system call and modified by the kernel, but is otherwise not used by the kernel.

long esterror (ro)

> This member is the value of the time_esterror kernel variable, which represents 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 provided bu a ntp-adjtime() system call, but is otherwise not used by the kernel.

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. In the fragment, rw = read/write, ro = read-only, wo = write-only.

```
/*
 * NAME
 *   ntp_adjtime - NTP daemon application interface
 *
 * SYNOPSIS
 *   #include ys/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 {
        unsigned int mode;          /* mode selector (wo) */
        long offset;                /* time offset (us) (rw) */
        long frequency;             /* frequency offset (scaled ppm) (rw) */
        long maxerror;              /* maximum error (us) (rw) */
        long esterror;              /* estimated error (us) (rw) */
        int status;                 /* clock status bits (rw) */
        long constant;              /* pll time constant (rw) */
        long precision;             /* clock precision (us) (ro) */
        long tolerance;             /* clock frequency tolerance (scaled  ppm) (ro) */
        /*
         * The following read-only structure members are implemented
         * only if the PPS signal discipline is configured in the
         * kernel.
         */
        long ppsfreq;               /* pps frequency (scaled ppm) (ro) */
        long jitter;                /* pps jitter (us) (ro) */
        int shift;                  /* interval duration (s) (shift) (ro) */
        long stabil;                /* pps stability (scaled ppm) (ro) */
        long jitcnt;                /* jitter limit exceeded (ro) */
        long calcnt;                /* calibration intervals (ro) */
        long errcnt;                /* calibration errors (ro) */
        long stbcnt;                /* stability limit exceeded (ro) */
};
```

The ntp_adjtime() system call is used to read and write certain time-related kernel variables summarized below. 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.

mode (wo)

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, with the variable name indicated in parens. Note that the precision, tolerance and PPS variables are determined by the kernel and cannot be changed by ntp_adjtime().

```
MOD_OFFSET      0x0001      time offset (offset)
MOD_FREQUENCY 0x0002        frequency offset (frequency)
MOD_MAXERROR 0x0004         maximum time error (maxerror)
MOD_ESTERROR  0x0008        estimated time error (esterror)
```

MOD_STATUS	0x0010	clock status (status)
MOD_TIMECONST	0x0020	pll time constant (constant)
MOD_CLKB	0x4000	set clock B
MOD_CLKA	0x8000	set clock A

Note that the MOD_CLKA and MOD_CLKB bits are intended for those systems where more than one hardware clock is available for backup, such as in Tandem Non-Stop computers. Presumably, in such cases each clock would have its own oscillator and require a separate PLL for each. Refinements to this model are for further study. The interpretation of these bits is as follows:

offset (rw)

If selected, this member specifies the time adjustment, in microseconds. The absolute value must be less than MAXPHASE (128000) microseconds defined in the timex.h header file. On return, this member contains the residual offset remaining between a previously specified offset and the current system time, in microseconds.

frequency (rw)

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 (100) ppm defined in the timex.h header file.

The time_freq variable represents the frequency offset of the CPU clock oscillator. It is recalculated as each update to the system clock is determined by the offset member of the timex structure. It is usually set from a value stored in a file when the synchronization daemon is first started. The current value is usually retrieved via this member and written to the file about once per hour.

maxerror (rw)

If selected, this member replaces the value of the time_maxerror kernel variable, in microseconds. This is the same variable as in the ntp_getime() system call.

esterror (rw)

If selected, this member replaces the value of the time_esterror kernel variable, in microseconds. This is the same variable as in the ntp_getime() system call.

int status (rw)

If selected, this member replaces the value of the time_status kernel variable. This variable controls the state machine used to insert or delete leap seconds and shows the status of the timekeeping system, PPS signal and external oscillator, if configured.

STA_PLL	0x0001	enable PLL updates (rw)
STA_PPSFREQ	0x0002	enable PPS freq discipline (rw)
STA_PPSTIME	0x0004	enable PPS time discipline (rw)
STA_FLL	0x0008	select FLL mode (rw)

STA_INS	0x0010	insert leap (rw)
STA_DEL	0x0020	delete leap (rw)
STA_UNSYNC	0x0040	clock unsynchronized (rw)
STA_FREQHOLD	0x0080	frequency hold (rw)
STA_PPSSIGNAL	0x0100	PPS signal present (r)
STA_PPSJITTER	0x0200	PPS signal jitter exceeded (r)
STA_PPSWANDER	0x0400	PPS signal wander exceeded (r)
STA_PPSERROR	0x0800	PPS signal calibration error (r)
STA_CLOCKERR	0x1000	clock hardware fault (r)

The interpretation of these bits is as follows:

STA_PLL

set/cleared by the caller to enable PLL updates

STA_PPSFREQ

set/cleared by the caller to enable PPS frequency discipline

STA_PPSTIME

set/cleared by the caller to enable PPS time discipline

STA_FLL

set/cleared by the caller; set selects FLL mode, clear selects PLL mode.

STA_INS

set by the caller to insert a leap second at the end of the current day; cleared by the caller after the event

STA_DEL

set by the caller to delete a leap second at the end of the current day; cleared by the caller after the event

STA_UNSYNC

set/cleared by the caller to indicate clock unsynchronized (e.g., when no peers are reachable)

STA_FREQHOLD

set/cleared by the caller to disable frequency update.

STA_PPSSIGNAL

set/cleared by the hardpps() fragment to indicate PPS signal present

STA_PPSJITTER

set/cleared by the hardpps() fragment to indicates PPS signal jitter exceeded

STA_PPSWANDER

set/cleared by the hardpps() fragment to indicates PPS signal wander exceeded

STA_PPSERROR

set/cleared by the hardpps() fragment to indicates PPS signal calibration error

STA_CLOCKERR

set/cleared by the external hardware clock driver to indicate hardware fault

An error condition is raised when (a) either STA_UNSYNC or STA_CLOCKERR is set (loss of synchronization), (b) STA_PPSFREQ or STA_PPSTIME is set and STA_PPSSIGNAL is clear (loss of PPS signal), (c) STA_PPSTIME and STA_PPSJITTER are both set (jitter exceeded), (d) STA_PPSFREQ is set and either STA_PPSWANDER or STA_PPSERROR is set (wander exceeded). An error condition results in a system call return code of TIME_ERROR.

constant (rw)

If selected, this member replaces the value of the time_constant kernel variable. The value must be between zero and MAXTC (6) defined in the timex.h header file.

The time_constant 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 oscillators, 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 precision external oscillator is present.

precision (ro)

This is the current value of the time_precision kernel variable in microseconds.

The time_precision variable represents the maximum error in reading the system clock, in microseconds. It is usually based on the number of microseconds between timer interrupts (tick), 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 stock 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.

tolerance (ro)

This is the current value of the time_tolerance 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 time_tolerance variable represents the maximum frequency error in ppm of the particular CPU clock oscillator and is a property of the hardware; however, in principle it could change as result of the presence of external discipline signals, for instance.

The recommended value for time_tolerance MAXFREQ (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.

The following members are defined only if the PPS_SYNC option is specified in the kernel configuration file. These members are useful primarily as a monitoring and evaluation tool. These variables can be written only by the kernel.

ppsfreq (ro)

This is the current value of the pps_freq kernel variable, which is the CPU clock oscillator frequency offset relative to the PPS discipline signal. The value is in ppm, with the integer part in the high order 16 bits and fraction in the low order 16 bits.

jitter (ro)

This is the current value of the pps_jitter kernel variable, which is the average PPS time dispersion measured by the time-offset median filter, in microseconds.

shift (ro)

This is the current value of the pps_shift kernel variable, which determines the duration of the calibration interval as the value of 1 < pps_shift, in seconds.

stabil (ro)

This is the current value of the pps_stabil kernel variable, which is the average PPS frequency dispersion measured by the frequency-offset median filter. The value is in ppm, with the integer part in the high order 16 bits and fraction in the low order 16 bits.

jitcnt (ro)

This is the current value of the pps_jitcnt kernel variable, counts the number of PPS signals where the average jitter exceeds the threshold MAXTIME (200 us).

calcnt (ro)

This is the current value of the pps_calcnt kernel variable, which counts the number of frequency calibration intervals. The duration of these intervals can range from 4 to 256 seconds, as determined by the pps_shift kernel variable.

errcnt (ro)

This is the current value of the pps_errcnt kernel variable, which counts the number of frequency calibration cycles where (a) the apparent frequency offset is greater than MAXFREQ (100 ppm) or (b) the interval jitter exceeds tick * 2.

stbcnt (ro)

This is the current value of the pps_discnt kernel variable, which counts the number of calibration intervals where the average stability exceeds the threshold MAXFREQ / 4 (25 ppm).

5. References

[MIL91]Mills, D.L. 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.

[MIL92a]Mills, D.L. Network Time Protocol (Version 3) specification, implementation and analysis, RFC 1305, University of Delaware, March 1992, 113 pp.

[MIL92b]Mills, D.L. Modelling and analysis of computer network clocks, Electrical Engineering Department Report 92-5-2, University of Delaware, May 1992, 29 pp.

[MIL92c]Mills, D.L. Simple Network Time Protocol (SNTP), RFC 1361, University of Delaware, August 1992, 10 pp.

[MIL93]Mills, D.L. Precision synchronization of computer network clocks, Electrical Engineering Department Report 93-11-1, University of Delaware, November 1993, 66 pp.

[LEV89]Levine, J., M. Weiss, D. Davis, D. Allan, and D. Sullivan. The NIST automated computer time service. *J. Research National Institute of Standards and Technology 94, 5* (September-October 1989), 311-321.

[MIL94]Mills, D.L. Improved algorithms for synchronizing computer network clocks. *Proc. ACM SIGCOMM 94 Symposium* (London UK, September 1994), 317-327.