A Programming Interface for Precise Computer Clock Adjustments and Synchronization

Dennis Ferguson

(Version 0.31)

1. Introduction

Computer operating systems have provided mechanisms to maintain a system time-of-day clock, and provided methods for applications to use to sample the current time-of-day, since there were computer operating systems. As this facility has generally promised that the time returned to applications has a well-defined relationship to civil time[1] (i.e. the time reported on the radio which we set our wrist watches to), for example by referencing the time returned to a civil time epoch, operating systems have also provided methods for system utilities to use to set the computer's time-of-day clock to civil time. It is the case, however, that the methods provided for setting the computer's clock have essentially mimicked the process you would use to set your wristwatch, that is they solely allowed one to provide the operating system with an indication of what some external time source thought was correct "now" with the apparent presumption that once the computer was told that it would be able to provide an accurate time-of-day into the future. The POSIX[2] system call *clock_settime(2)*[3] provides such a facility, the only standard facility provided to enable the computer clock's time to return the time it promises, that is time referenced to a civil time epoch. The fallacy of this expectation is that even if you set the clock's time to be precisely correct when you make the system call (though you are unlikely to be able to do even this because of the ambiguity about when "now" actually is), the computer's extrapolation of this time into the future is, like your wristwatch, almost certainly dependent on counting cycles from an oscillator whose actual frequency not only differs from the nominal frequency specified for it but which in reality is a variable, moving target. While the frequency error will generally be relatively small the effect it will have on the time will be cumulative so that, like your wrist watch, if you wait a while you will likely find that the system returns time which is not nearly as accurate as it was when the clock was originally set. The longer you wait the less accurate it is likely to have become.

Ensuring that the quality of the time returned by the computer's clock does not continually degrade in this fashion makes it desirable to implement a process which continuously (or, at least, frequently) compares the computer's clock to some more accurate source of time, perhaps to an attached hardware device, like a GPS receiver, or to the clock in another

---

[1] For example, see http://en.wikipedia.org/wiki/Civil_time

[2] http://www.unix.org/

[3] http://pubs.opengroup.org/onlinepubs/009695399/functions/clock_settime.html

computer which is attached to such a device and may be queried using a network connection. When the clock in the local computer drifts noticeably out of synchronization with its reference it can be corrected to bring it back into alignment. Note that when the problem becomes not one of setting the clock to a time, as above, but rather comparing the clock to another clock, it becomes more convenient (if not imperative) to track the difference between the local time and the reference time expressed as a time offset. Time offsets are comparable between samples; the rate of change of the time offset over time is related to the magnitude of the error in the local clock's estimate of the clock oscillator's frequency, an error which is likely to be both relatively small as well as being interesting to measure. The BSD Unix *adjtime(2)*[4] system call allows time changes to be specified as a relative offset to the current time, which removes the ambiguity of "now" when correcting the time error; the time after the adjustment completes will be the same independent of the precise moment when the adjustment is performed. Additionally, *adjtime(2)* makes use of the latter property by applying the adjustment over time, rather than all at once, a feature which can permit the time to be adjusted (perhaps) without causing a discontinuity in the time returned (at the expense, of course, of lengthening the time it takes to return the clock to "correct" time; for this reason *clock_settime(2*) is often used instead when making large time changes, despite the problem of "now" it reintroduces). While this is good, however, it is incomplete. The reason a clock which was once accurately aligned to civil time will become inaccurate is because the operating system's understanding of the frequency of the oscillator driving its clock does not match the oscillator's actual frequency. While an adjustment mechanism to make a phase correction to the time is good to have to correct time errors accumulated in the past, making the clock keep better time in the future is best accomplished by correcting the frequency error which is causing the problem rather than making time adjustments repeatedly (and generally frequently) to compensate for it.

There has been (at least) one effort to address the latter deficiency, the *ntp_adjtime(2)* interface described in RFC 1589[5]. The difficulty with this is that, while the interface provides mechanisms to adjust both time (i.e. phase) and frequency errors, it does so in a way which embeds the mechanism in a policy for its use. Essentially the system call moves a small portion of a standard Network Time Protocol[6] implementation into the operating system's kernel, yielding a mechanism which is fully useful only to the remainder of a standard NTP application running outside the kernel. Even if one could argue that standard NTP is indeed the optimal way to keep the system's clock synchronized, however, I think this design is unfortunate if only for aesthetic reasons. An operating system should provide the general mechanisms required to perform necessary operations but, to the extent possible, leave the specific policy for using those mechanisms to the applications which need them. The *ntp_adjtime(2)* interface fails badly when evaluated by this standard; even just figuring out what it does requires an intimate familiarity with standard NTP, and much of what it does is of little use to anything other than standard NTP.

The situation thus seems to be as follows. The operating system interface standard we have, POSIX, provides no mechanism at all which is suitable for use by a process tasked with keeping a computer's time-of-day clock accurate by continuous monitoring and adjustment. The ad hoc interface *adjtime(2)*, which is not standardized but which is widely

---

[4] http://www.unix.com/man-page/FreeBSD/2/adjtime/

[5] http://tools.ietf.org/html/rfc1589

[6] http://tools.ietf.org/html/rfc5905

implemented, is incomplete; it implements phase adjustments to correct time errors which have already occurred but provides no mechanism to correct the frequency errors which are the underlying cause of those time errors and which, if corrected, would allow the system's clock to provide more accurate time in future. The *ntp_adjtime(2)* interface corrects this lack but does so in such an exceedingly application-specific way with features and side effects that it are not of general use.

This paper, then, attempts to correct this situation by describing a method for operating systems to provide applications with the ability to correct both time and frequency errors made by the system's clock in a way which provides appropriate mechanisms without burdening them with any particular usage policy. It does so by first defining a model for a computer's clock which describes the operations and adjustment functions which an implementation should seek to emulate. It then discusses properties of the model which are highly desirable for an implementation to maintain with as much precision as possible, and reinforces this by looking at adjustment strategies which may be enabled by this. It then suggests a programming interface which, if well implemented, meets the requirements and suggests a possible practical implementation of that programming interface. Note that this is very much a work in progress.

2. A Model Computer Clock

A diagrammatic model of an "ideal" computer clock is shown in Figure 1.

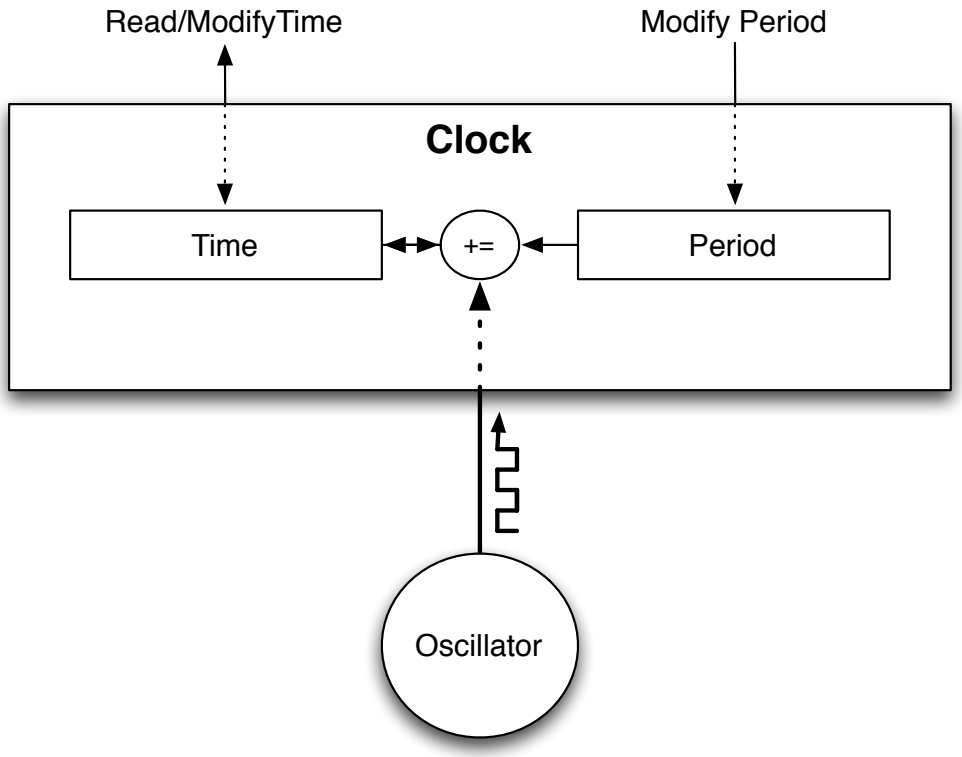

Figure 1: Clock Model

The clock is driven by a free-running oscillator. For each oscillator cycle the value of the "Period" register is added to the "Time" accumulator register. The "Time" register is made

available for applications to read; it is, in effect, the computer's time-of-day clock. For the "Time" register to be advanced accurately the "Period" register needs to be programmed to the actual period of the oscillator (i.e. the inverse of the oscillator's actual frequency) in the same units as used to express "Time". Note that, for the purpose of this definition, the "Time" and "Period" registers are assumed to be arbitrarily precise in the mathematical sense that any of the infinity of possible values for these registers may be represented. While real implementations of the model will approximate these quantities with finite representations, the model itself is defined exactly.

The model defines several adjustment operations which can be performed on the clock. The first of these are "atomic" operations which are performed at a single conceptual instant:

- A *Step* adjustment is performed by adding an offset to, or subtracting an offset from, the current contents of the "Time" register. This allows "Time" to be correctly aligned with respect to a civil time epoch, though it results in a phase discontinuity which may be observed by readers of the "Time" register.

- A *Rate* adjustment is performed by adding an offset to, or subtracting an offset from, the "Period" register. This provides a method to adjust the "Period" to more accurately match the current period of the oscillator. Note that to avoid applications having to know the actual frequency of the oscillator (and for other reasons which should soon become clear) the interface to this facility will likely instead specify the offset as a ratio of rates (requiring a multiply to determine the offset). In particular a *Relative Rate* adjustment specifies the offset as a fraction of the current value of the "Period" register, while an *Absolute Rate* adjustment will specify the value to set the "Period" register as a fraction of the nominal period of the oscillator, i.e. the default value the operating system would assign to the "Period" register.

Besides the atomic operations there are several operations which instead are completed some time after they are initiated:

- A *Slew* adjustment is performed by altering the "Period" register, as a *Rate* adjustment would, for a predetermined time interval (perhaps conceptually measured by counting oscillator cycles), after which the "Period" register is returned to its original value. The effect of a *Slew* adjustment is hence to add an offset to the "Time" register like a *Step* adjustment, but in a phase-continuous manner by allowing the offset to accumulate over a finite interval like *adjtime(2)*. Note that the interface to this adjustment facility is likely to allow the application to equivalently specify a time offset and a rate adjustment with which to accumulate this offset, with the adjustment facility computing the amount of time required to accumulate the exact offset specified.

- A *Leap* adjustment is performed by adding an offset to the "Time" register, like a *Step* adjustment would do immediately, but deferring the operation until the contents of the "Time" register reach some specified value. The intended purpose of a *Leap* adjustment is to provide a mechanism to handle leap seconds. To do this the *Leap* adjustment would be scheduled by the application ahead of midnight UTC, with the offset specified as ±1 second and the time for the operation set to precisely midnight.

- A *Sloop*, or "*Slewing Leap*", adjustment is a *Slew* adjustment whose initiation is delayed until the contents of the "Time" register reach some specified value, at which point the "Period" register is altered for a specified time interval. A *Sloop* hence provides an

alternative method of handling leap seconds, one which is useful when it is desired to avoid the non-monotonic behavior of a *Leap* adjustment. Note that while I am not personally convinced that this is the most useful way to handle leap second insertion (its use implies that there are no applications which care about precisely frequency-accurate time, and I doubt this), it is a common enough proposal for this purpose that it deserves support.

The existence of adjustment operations which will complete only after some discrete time interval has elapsed requires the model have an answer to an additional question: What happens when another adjustment is attempted before the previous *Slew*, *Leap* or *Sloop* adjustment has completed? The simplest answer, that no adjustment may be made until the previous adjustment has been completed, is the one chosen for this model. To avoid the loss of control of the clock which is a potential consequence of this choice, one additional clock operation is prudent:

- An *Abort* operation causes any in-progress *Slew*, *Leap* or *Sloop* adjustment to be immediately terminated, with the undone portion of the operation being made available to the caller. For a *Leap* adjustment the "undone portion" will always be the entire offset, so the *Abort* of a *Leap* always cleanly undoes what the *Leap* was scheduled to do (unless, of course, the *Leap* was already completed, in which case there is nothing to *Abort*). For a *Slew* adjustment an *Abort* causes the "Period" register to be returned to its original, pre-*Slew* value, with the not-yet-done portion of the *Slew* offset (or information sufficient to compute that value) being returned. An aborted *Sloop* may return either the entire offset or a fraction of it depending on whether the scheduled slew was aborted before or after it was initiated.

Note that a programming interface providing these operations to an application is required to also provide a way to return to the application the value of the "Time" register when an operation is initiated. Furthermore, while real implementations of the clock specified by the model may have constraints which prevent them from precisely performing the operation they are asked to (real registers are not arbitrarily precise, for example), it is required that any application interface return to the application an indication of the operation which was actually performed as precisely as possible. That is, we allow real implementations to be imprecise in how they implement an operation, but require them to precisely describe what they ended up doing.

Finally, while the model is intended to describe behavior without dictating implementation, it might be worth while to return to Figure 1 and imagine how the elements shown there might map to a real implementation in an actual computer. There are two elements to consider. The circle labeled "Oscillator" can only represent an actual piece of hardware soldered on a board somewhere, and will have all the characteristics of the analog oscillator providing the function. It will make errors, in that its output frequency will differ from the number written on its package, will vary unpredictably during operation and will generally be precisely unknowable at any particular time. In contrast, the box labeled "Clock" is an entirely mathematical construct whose operation is exactly defined by the arithmetic it performs, and as a consequence is perfectly predictable. The sole source of errors in that picture is the oscillator. The clock incorporates those errors by use of the oscillator's output as an input but (ignoring software bugs that real implementations might incorporate, which we'll assume will eventually be fixed) is itself error-free in operation and can make the adjustments it is told to with mathematical precision. The next section discusses useful properties a "perfect" clock (with an imperfect oscillator) can have.

3. Relationships Between Clocks

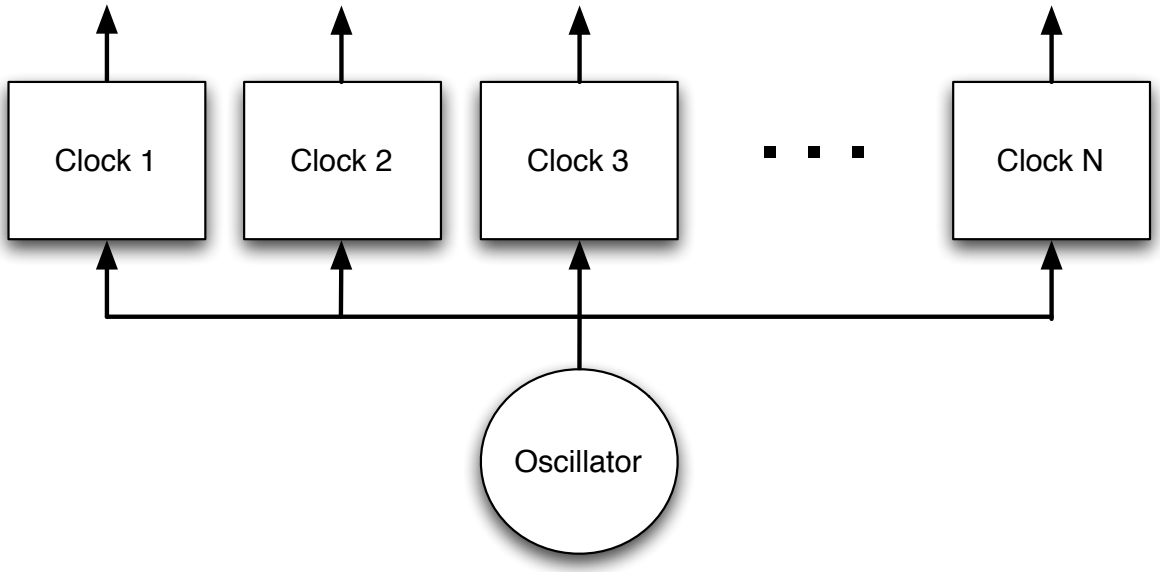

Figure 2: One Oscillator, Many Clocks

The mathematical predictability of the effect of adjustments on a clock, and the fact that the oscillator is the sole source of unknowable errors, has implications for arrangements with several clocks being driven by a single oscillator, as in Figure 2. Such clocks are *synchronous*, and are not independent; whatever errors are made by the oscillator effect all clocks in a precisely related way. While they may have different settings and different time outputs, the time value being output by any of the clocks at a particular moment can be computed precisely from a knowledge of the time output of any other clock at the same moment. That is, if you know the time being output by one of the clocks you can determine from that the time being concurrently output by all the others. To see why this is so, consider the pair of clocks attached to a common oscillator shown in Figure 3.

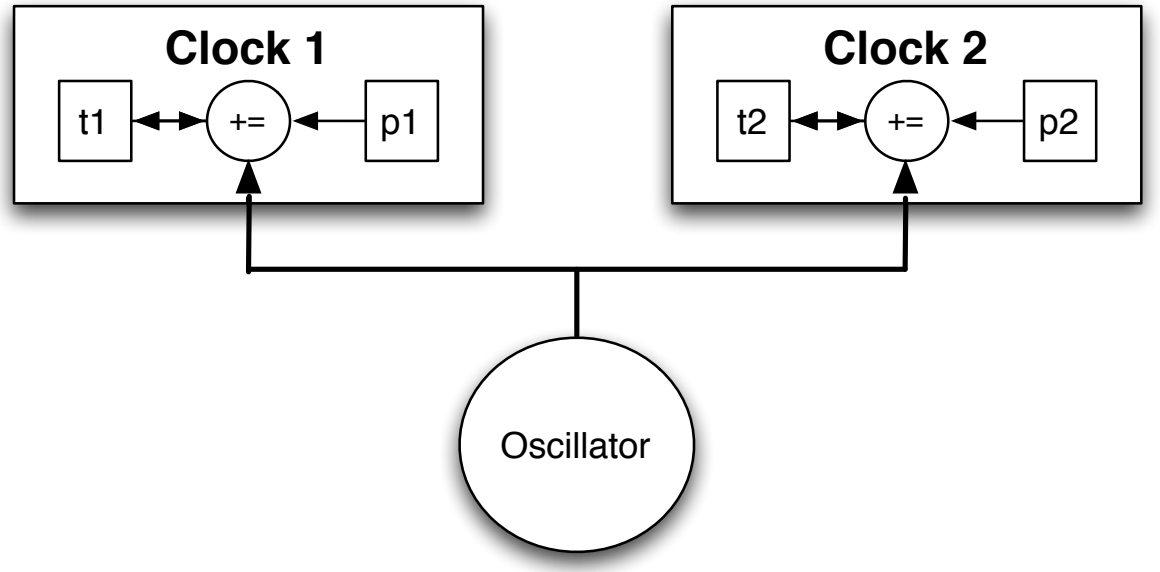

Figure 3: A Pair of Clocks

Given t_1, we wish to determine t_2. What we know, in addition to p_1 and p_2, is that at an arbitrary time, past or future, when the output of Clock 1 was T_1, the output of Clock 2 was T_2 at the same moment. If n is the number of oscillator cycles which have elapsed since that moment and the present (n may be negative) then the output from each clock can be generally written as

$$t_1 = p_1 \cdot n + T_1$$
$$t_2 = p_2 \cdot n + T_2$$

Eliminating n, we can write

$$t_2 = (p_2/p_1) \cdot t_1 + (T_2 - (p_2/p_1) \cdot T_1)$$

or, equivalently

$$t_2 = r_{21} \cdot t_1 + C_{21}$$

where

$$r_{21} = p_2/p_1$$
$$C_{21} = T_2 - r_{21} \cdot T_1$$

That is, the output of any clock may be computed precisely from the output of any other clock driven by the same oscillator by multiplying by a rate constant (which is the ratio of the two periods) and adding a time offset. If we know the output of one clock attached to the oscillator we can precisely compute the output of any, or all, other clocks that could be driven by the same oscillator by selecting the appropriate rate and time offset constants to use for the conversion. One way to look at this is that all possible clocks attached to the oscillator always exist and when an adjustment to the system clock is made what is really being done is to select a different one of those clocks for time consumers to sample, one whose output is currently a closer match for civil time. An alternative visualization might be that in reality there is only one clock outputting one time, that all settings of the clock output the same time but expressed in different units, and that the task of clock adjustment is to just find the output units which provide the current best representation of civil time. Since time adjustments are just unit changes, we can always continue to compute time in the pre-adjustment units, or any other, by a unit conversion implemented by a multiply and add.

Considering the relationship between the time before and after a *Step* and a *Relative Rate* adjustment (all other adjustments are variants of these) may make it clear why the clock adjustment interface is required to accurately return the time at which the adjustment takes effect to the application making the adjustment. If t_1 is the pre-adjustment time scale and t_2 is the post-adjustment time scale then, for a *Step* offset of O, we have

$$r_{21} = 1$$
$$C_{21} = T_2 - T_1 = O$$

giving us

$$t_2 = t_1 + O$$

independent of what the times on each time scale were at the moment the change was implemented (i.e. T_1 and T_2). For a *Relative Rate* adjustment of r implemented at time T (since a rate adjustment is phase continuous the time of the adjustment, T, will be the same in both time scales) we have

$$r_{21} = r$$
$$C_{21} = T_2 - r \cdot T_1 = (1 - r) \cdot T$$

or

$$t_2 = r \cdot t_1 + (1 - r) \cdot T$$

That is, for clock adjustments involving rate changes, knowing the time at which the rate was changed is necessary to determine the relationship between the old and new time scales. Note, too, that while we've computed the post-adjustment time scale as a function of the pre-adjustment time scale above, we could equally well have produced equations to determine time in the pre-adjustment time scale given the post-adjustment time. This means we can precisely answer two questions that can often useful to know an answer to when making measurements of the local clock against a reference to determine the adjustments to make, those being "What time would it be now if I hadn't made the adjustment?", and "What would my (pre-adjustment) measurements have been if I'd made this adjustment before making those measurements?". Since the behavior of a clock relative to others driven by the same oscillator is defined solely by mathematics the relationships above are indeed exact. A real implementation will of course be computing these relationships with finite precision arithmetic, which by definition won't be exact, but the goal of a programming interface is to implement adjustments and provide the application with data concerning what was done with all precision as is practical to compute.

There is a class of special-case clocks which is worthy of note as well. These are *Counter Clocks* and an example is shown in Figure 4.

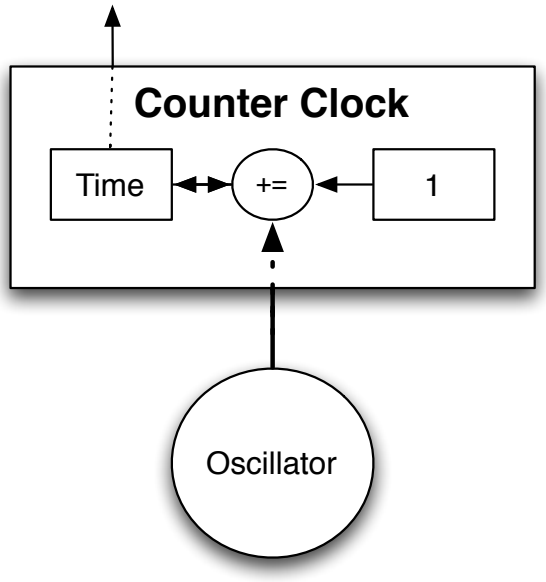

Figure 4: A Counter Clock

That is, counter clocks increment their “Time” register by 1 for every oscillator cycle, essentially counting oscillator cycles as they pass. Counter clocks are interesting because the arithmetic they need to do per cycle is minimally simple (adding 1 to “Time” at every cycle is less work than adding of an arbitrary precision “Period” at every cycle, as the generic clock model demands) but, as has been discussed in this section, the output of any other clock which could be (virtually) driven by the same oscillator can be determined from the output of the counter clock by a multiply and an add after the counter clock’s output is sampled. Assuming we sample the clock less frequently than the frequency of the oscillator, being able to do a simple increment per oscillator cycle in exchange for the cost of doing a multiply when the clock is sampled may be an attractive tradeoff, and this is certainly likely to be true on modern CPUs where the cost of multiplies is exceedingly cheap compared to, say, memory accesses. Moreover, we can maintain an adjustable time-of-day clock based on a nonadjustable counter clock with no loss of generality; adjustments are instead implemented by manipulating the multiplicative (rate) and additive (offset) variables to achieve the desired result.

What is more interesting about counter clocks, however, is that many modern processors have one or more of these implemented in hardware, perhaps counting processor cycles or the cycles of some I/O-related oscillator. As such, a counter clock is often the form in which the highest resolution time available to a processor is known, so assuming one is available and building an adjustment infrastructure around rate and offset constants used for the conversion to the desired real-time units is likely to yield a high quality implementation which is also quite portable.

4. Clock Synchronization Applications and their Cost

The point of defining a standard computer clock adjustment mechanism is to provide support for applications whose purpose is to maintain the accuracy of that clock, so a summary of how the mechanism described here might be used to provide this function can provide useful insight into both the advantages of the mechanism and the issues which an implementation should take care to address. Of course a summary of this topic would be grossly incomplete without at least some consideration of how NTP might make use of the mechanism. The standard Network Time Protocol is by far the most widely deployed method for maintaining a computer clock’s time and, as such, NTP is likely to be the most common application to use any clock adjustment interface for some time to come. NTP is, however, constrained by its own clock adjustment model which dictates the use of a feedback control approach to the problem, with the attendant performance constraints and stability concerns of these algorithms. While this is certainly a valid implementation choice the problem of computer clock synchronization is not inherently a feedback control problem, and the clock model documented here allows alternative approaches which treat the problem as a more straight forward measurement problem.

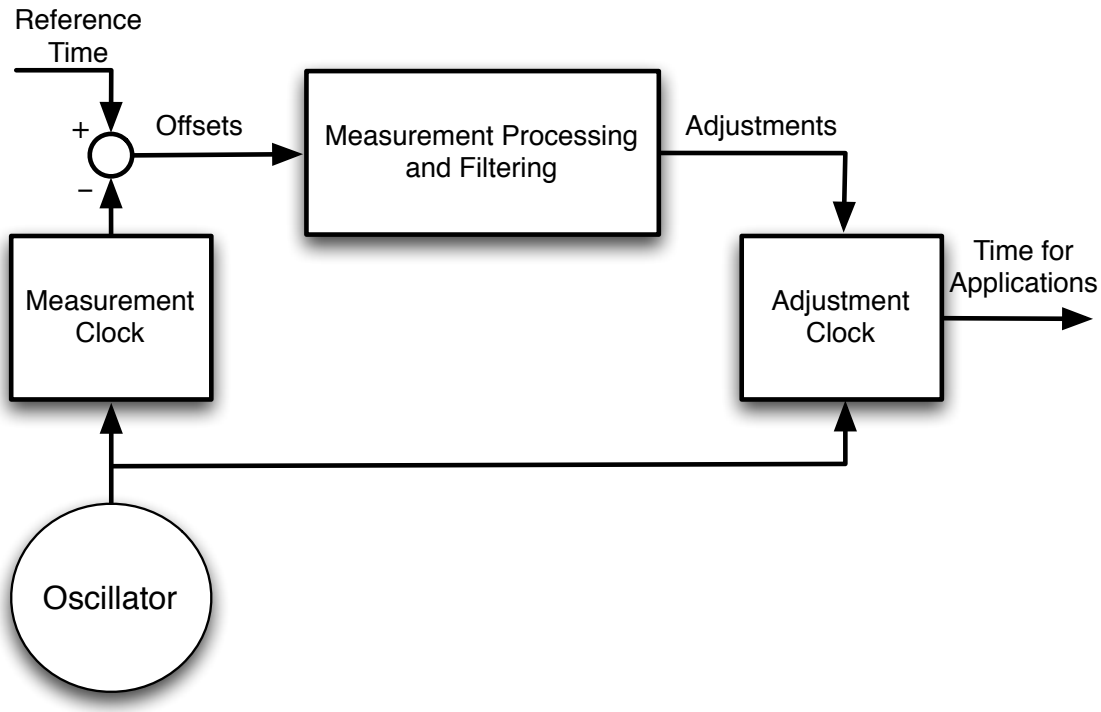

Figure 5: A Clock Synchronization Application

Consider, for example, the clock adjustment application schema shown in Figure 5. The application compares time from a reference clock to the time from a local "measurement clock". From this comparison it can determine the difference between the reference and measurement clocks' rate of time advance (i.e. the frequency error of the measurement clock's setting) and estimate the time offset between the reference and measurement clocks at a particular moment (since the measurement clock's period will not match the oscillator's actual period, the time offset will slowly vary with time). The application then uses this information to adjust a second, "adjustment clock" to compensate for the errors being measured, with the output of the "adjustment clock" being made available to consumer applications. Since both clocks are driven by the same oscillator the relationship between the two clocks can be known exactly and errors measured with respect to the measurement clock can be precisely transferred to the adjustment clock as adjustments to compensate for the errors measured (note that the two-clock arrangement is not my invention; see, for example, RADclock[7] for a slightly different use of the arrangement). Note well, however, that there is no feedback path in this application; all measurements are made with a clock which is never adjusted, and all adjustments are made to a clock whose output is never measured. It is an open loop controller, whose output will depend on the accuracy of the controller's model of the system it is adjusting, but in this case that system is a mathematical construct that performs in a manner which can be computed by the controller with great precision.

Of course it might occur to one to argue that the above application model is of no practical interest. Since there are no computer operating systems which maintain more than one clock there is only one possible source of time stamps and the clock being adjusted is necessarily going to be the same clock used for measurement timestamps. While this is certainly true, we know from the previous section that the timestamps from one clock can

[7] http://www.cubinlab.ee.unimelb.edu.au/radclock/

be used to compute the timestamps from any other clock connected to the same oscillator with great precision, so the fact that the operating system's kernel doesn't have a second, unadjusted clock does not prevent us from computing, with arbitrary precision, the timestamps we would have gotten from the unadjusted clock ourselves as a function of adjusted clock's timestamps. The implication is that even if you apply the adjustments to the clock you are using to take measurement timestamps, if you can precisely compute an answer to the question, "What timestamps would I be getting if I hadn't made that adjustment?" then the problem you are solving is not one which inherently requires a feedback control solution.

It may be informative to examine just why it is that NTP treats the time synchronization problem requiring a feedback control solution like the hybrid PLL/FLL algorithm it uses (and sometimes inserts in the kernel) to compute adjustments. The fundamental issue that leads to this is NTP's model of the computer's clock and how it is adjusted; if you look at a block diagram of NTP's processing loop you will no doubt find the construction shown in Figure 6. NTP models the adjustment of a computer clock as a change to the frequency of its oscillator.

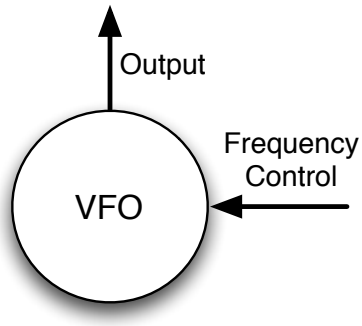

Figure 6: NTP's Variable Frequency Oscillator Adjustment Model

This might not seem like a big deal (clearly there is no actual VFO to adjust in most [any?] computers, so the above is a fictional model which is implemented by other means) but modeling the clock this way has profound implications for the assumptions the NTP application can make about the adjustment process. Recall that above we pointed out that, for the model described in this paper, the oscillator was the source of all errors; its output frequency was unlikely to match its nominal value, was likely to vary with time and was not precisely knowable or predictable. The oscillator's actual output frequency could only be determined by measurement, and only known to the precision those measurements could provide. In contrast the adjustment process, which is implemented in a digital clock whose operation was fully defined by the arithmetic it performs, is perfectly precise and predictable. The effect of NTP's clock model on the problem is to move the implementation of adjustments from the mathematically precise and predictable place where the current model places it to a place where it shares the error characteristics of the analog hardware source of all errors in the current model; that is, NTP assumes that not only is the oscillator inaccurate and in need of measurement, but also that the adjustments made in response to those errors are similarly inaccurate and also require measurement to determine what adjustment was done. Or, to put it another way, the clock model presented here allows the application the knowledge that whatever the frequency error of the oscillator is at the time of an adjustment, it will be unaffected by the adjustment itself. For the NTP model of the clock, however, an adjustment explicitly changes that error to something else entirely, and that the "something else" is not perfectly predictable, so the only way to determine the post-adjustment error is to measure it again. Since clock

adjustments effect the errors being measured, and the errors being measured in turn determine future clock adjustments, NTP's clock adjustment model implicitly defines the problem as a feedback control problem. The problem certainly would be a feedback control problem if the clock's oscillator really were a VFO, and if clock adjustments were implemented by an analog adjustment of its frequency, but there is likely no computer for which this is the case.

Treatment of the clock adjustment problem as a feedback control problem introduces the additional constraint of stability considerations, that is the problem of ensuring that future adjustments are predominantly a response to future errors being made by the oscillator rather than to the errors in previous adjustments made by the application. This is a particularly difficult problem in the NTP environment where the sample rate is relatively low and measurements are always delayed (i.e. they tell you what the errors were some time in the past, rather than what they are now). To approximate a stable response (since in theory delays in the feedback loop are unconditionally destabilizing) NTP must make its response slow enough that its sample delays can be assumed to be approximately zero in comparison, behavior which is apparent in the many hours NTP can take to correct a large frequency error. NTP's choice of a clock model causes NTP to respond to large frequency errors very slowly.

To be fair there are some considerable strengths to the NTP approach to the problem which may offset the drawbacks in many circumstances. For a computer clock the equivalent to the (nonexistent) VFO adjustment which NTP assumes for its model is a digital adjustment interface for the clock which doesn't, or can't, make the adjustment it is told to and in real life these abound (it is this which the current paper seeks to correct). NTP can be astoundingly robust in the face of quite poor adjustment interfaces since its operation includes measurement and correction (when it can) of adjustment errors that are made along with the more fundamental oscillator errors. Moreover, when dealing with the awful realities which can occur in over-a-network timekeeping, just going slow is not necessarily a defect. While we might debate the potential superiority of alternate approaches to the time synchronization enabled by the adjustment interface proposed here we must also admit that, lacking quantitative data for comparison, such debates are unlikely to be persuasive (even to me). What is worth looking at, however, is the necessary cost to a system of running a time synchronization application, whether NTP or something else, in terms of the frequency with which adjustments need to be made to the system clock (the cost) and the accuracy of the resulting time (the benefit), and how the current work might be used to bring NTP's behavior more in line with a more optimal cost/benefit tradeoff. Why we've chosen the frequency with which the adjustment interface is used as the major cost of time synchronization (it is self-evident that accurate time is the benefit) should become clear when a possible implementation of the clock adjustment interface is described.

We'll approach this by estimating the frequency with which a non-feedback, measurement-based clock controller might need to make clock adjustments to keep the time optimally accurate. We will assume the following behavior for the controller: The controller will act to measure the frequency error of the oscillator driving the clock (and the associated phase error accumulated during frequency excursions). When it makes an adjustment it will set the adjustment clock frequency and time to match its best estimate of the correct values based on its measurements (it should always be able to fully compensate for the errors it has measured no matter what the size of those errors are since the size of its adjustments are unconstrained by stability considerations). Once it has made an adjustment, however, it will refrain from making another adjustment until it can determine with confidence that its

new measurements are significantly different from the current settings of the clock in a statistical sense. That is, we will demand that the process compute its estimates in a way which includes error bars, or confidence limits, and allow it to make a change to the clock only when the clock's current settings are outside of the measurement's confidence limits so that changes are only made when we have statistical confidence that the measured values are indeed more accurate than the current clock's setting.

While we might lack a particular implementation of this approach, the general problem of the measurement of frequency errors is well understood, and is well characterized (if some assumptions about the errors the oscillator is making are true) by a quantity known as the *Allan deviation*[8]. To describe what this tells us about the current problem, consider that we have two conflicting problems in accurately determining the frequency error of the clock's oscillator. The initial problem is that the measurements against the reference which we make are noisy, with a random error superimposed on the signal. To achieve greater accuracy with noisy data we need to average them, with averages over longer and longer times providing more and more accurate low order bits in our estimate. The difficulty is that averages only unconditionally help if the thing we are measuring is a constant, and the frequency error is not that. Instead the oscillator's frequency is a slowly but continuously changing target, which means that after some length of time it will make no sense to average longer to determine more low order bits since the frequency will have moved far enough for the low order bits we'd determine by a longer average to have already changed because of the random frequency walk. In effect, for a given oscillator and measurement quality there is an optimum averaging time over which the frequency error can be determined with all the accuracy with which it is possible to know it. Dr. Mills's book[9] calls this the x-coordinate of the *Allan intercept*, and asserts that the time for a typical computer clock synchronized by a typical NTP time source is on the order of a few hundred to a few thousand seconds.

What this implies is that normally a measurement period of a few hundred to a few thousand seconds should be necessary and sufficient to determine our frequency error with as much accuracy as it is possible to know it (independent of the magnitude of the error), and that once this adjustment is applied (which it can be in full since we are free of stability-related constraints on the size of the adjustment) it is likely to take a few hundred to a few thousand seconds to determine a new error estimate which is significantly different from the last one. That is, under normal circumstances we might expect the clock to be kept as accurate as it is possible to do with NTP with between 1 and 10 clock adjustments per hour; no more than this is required, nor would it improve the timekeeping. Between 1 and 10 clock adjustments per hour should maximize the signal-to-noise ratio of the adjustments, given the documented Allan deviation behavior for computer clocks, so more frequent adjustments serve solely to introduce more noise (but not more signal) into the system's clock.

In contrast, the present NTP implementation usually ends up computing a new clock adjustment at intervals no less than once every few seconds (when steering the clock by calls to *adjtime(2)*, or some equivalent provided by an operating system) and perhaps as frequently as every kernel clock interrupt (when using its native system call interface) even when the rate at which it receives new data from the peer it is synchronizing to only a few

---

[8] See, e.g., http://www.allanstime.com/AllanVariance/index.html

[9] Mills, David L. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space*. CRC Press, 2011, pp. 210-214.

times per minute. The reason it does this is in part a response to the historical lack of a system clock rate adjustment mechanism (which the present work aims to correct), making it necessary to implement rate adjustments by a continuous series of offset adjustments, and in part because the NTP clock model views its adjustment task as a continuous-time process which it approximates in a piecewise-linear fashion. In either case the present NTP implementation ends up adjusting the clock far more frequently than can be justified by consideration of what is minimally necessary to keep the system's time to the accuracy that NTP can determine.

An important goal of the clock adjustment interface, then, is to enable clock synchronization applications to keep the system time adjusted with the full accuracy justified by the available synchronization data source, but at the minimum cost to the system in terms of the rate at which adjustments are applied. Clearly a clock synchronization application written with this goal in mind might approach the problem very differently than the present NTP implementation does, but it is desired to allow even legacy applications to approach this goal with as little effort as possible. The latter is enabled by providing clock adjustment primitives whose effect on the system clock is completely and accurately predictable. That is, it is quite possible for the current NTP implementation, even on current systems, to compute frequency and offset corrections for the system's clock without adjusting that clock at all. This would be accomplished by maintaining the timescale NTP would prefer to see internal to the NTP application as a function of the system clock's time, with NTP's own time is computed as a function of system timestamps, so that the timescale NTP operates on and makes visible to its peers is its own internal version of what time it is rather than the system's time. Given this, the predictability of the adjustment interface would also allow NTP to adjust the system clock to make it more accurate without demanding that the system clock be maintained in precisely the fashion that NTP demands of its internal clock; since the system call interface always allows the question, "What time would I be getting if I hadn't made this adjustment?", to be answered with great precision the ability of the NTP implementation to operate with full precision (but uselessly) without adjusting the system clock means that it will also be able to do its work even when the system clock is being adjusted using an orthogonal policy. NTP's feedback process can be implemented without feedback through the system clock when the system clock can be reliably maintained in an open-loop fashion.

In summary, we expect that a mathematically precise and predictable clock adjustment interface will enable precise system time synchronization to be maintained with a minimum rate of adjustment under all circumstances when the synchronization application has been written to make proper use of the facility, independent of the application's own timescale maintenance needs. Local-to-the-application timescales may be maintained as needed, and as accurately as if the application were instead applying those adjustments to the kernel itself.

5. Times, Timescales and Time Consumers

Operating system time is maintained for the benefit of time-consuming applications, but different applications have different requirements and consideration of these can inform our choice(s) about the time we should endeavor to deliver. If experience can be relied upon, most applications can be assigned to one of the following categories:

1. Many applications don't care what time it is at all, but rather simply want to reliably be able to measure time intervals. That is, they want to be able to perform an operation every 2 seconds, or time out an operation which fails to complete in 3 minutes, or measure the time it took to complete some other operation. This (in my experience, which may well provide only a biased sample) probably describes most applications with time-consuming needs, as well as most code in a Unix kernel which consumes time. While one could argue about the precision of time interval measurement such applications might really require it seems like they should be provided with time intervals no less accurate than those a cheap, undisciplined crystal (say, 10^{-5}) might produce, but would benefit if the rate of advance of their timescale matched the rate of the S.I. second as closely as possible.

2. Some applications want to know time which is aligned to civil time (i.e. UTC) with some precision. Many financial applications need to be able to determine this with significant accuracy, log files are best kept timestamped with accurate time (say, so events can be correlated between different machines) and *cron(1)*'s configuration is written to specify events in local civil time.

3. Some applications want time which is synchronized to some accuracy with networked neighbors. Some security applications may depend on this, and timestamps written into distributed file systems are often compared to the time on machines other than the one which wrote the file.

4. A few applications want time which moves continuously forward, both while a machine is up and across reboots. The most important example of this is likely *make(1)*, which uses comparisons of file system timestamps to determine whether generated files are up-to-date with respect to (i.e. newer than) the files their generation depends on. These applications don't necessarily have particularly compelling requirements for accurate time beyond this (unless, of course, the file system is distributed across several machines in which case the application may be called upon to compare timestamps generated on different machines).

It is tempting to assume that all of these applications could be well served by keeping time in UTC, on the local machine and on all machines it is required to maintain time synchronization with, but there are difficulties with this. Of primary concern is the fact that observing UTC requires access to resources outside the machine and the availability and reliability of those resources may be difficult to guarantee under all circumstances. This, in turn, requires action to be taken if it is found that that the machine's clock is not well aligned to UTC. This may have caused applications in categories (2) and (3) to make mistakes and, while it may be too late to do anything about past mistakes, the most expedient thing to do would seem to be to fix the clock quickly once the fault is known so that they don't continue to make mistakes in future. The time discontinuity required to bring the time into alignment with UTC may, on the other hand, violate the assumptions about the time made by applications in categories (1) and (4), causing them to fail. This is particularly unfortunate for applications in category (1) since the cost of their failure (or the cost of writing code to avoid the failure) is paying for a benefit they care not at all about, the alignment of their time to UTC.

There is also a similar, though perhaps smaller, problem concerning UTC leap seconds. The POSIX "seconds since the epoch" timescale ignores leap seconds and represents time under the assumption that all days have exactly 86,400 seconds, so some adjustment needs to happen when a leap second occurs to maintain alignment with UTC. The

currently documented behavior of standard NTP when a leap second is added (which they always are) is to hold the time output by the kernel nearly constant for that second, which can cause problems for applications in category (1) and may be problematic for applications in category (2) as well. Other approaches to leap seconds include all machines slewing the time in synchrony over a long period, which is good for applications in categories (3) and (4), but causes an observable inaccuracy in time intervals for category (1) applications and leaves category (2) applications with a time which is in error with respect to UTC for the duration of the slew (though they might correct this themselves, I guess, if the behavior was well understood, though this becomes a bit complex), or to just make a step adjustment at some appropriate time and rely on the error recovery code a reliable application must have (since not all step adjustments to UTC are predictable leap seconds; they may be corrections for errors we've just discovered), which causes problems for category (1) and (4) applications. There seems to be no perfect solution with a single time scale.

Consideration of these issues leads me to the following conclusions: Any reliable application which relies on UTC time from the operating system, and particularly those in category (2), are going to need to have error recovery code to do whatever is necessary when it is discovered that the system's time was wrong and required an arbitrarily large change to bring into alignment. What the system should do for such applications is to make it very easy for them to determine when an event like this has occurred (it should be exceedingly rare), so that they know when to run the error recovery code, and to learn the magnitude and direction of the change (a logging application, for example, might write this into the log file so that earlier entries could be manually corrected if need be). The system needs to be free to correct its estimate of UTC when it is in need of correction, so consumers of this time need to be prepared to handle this and need a way to determine when the event has happened so that they can recover as best they can

The other conclusion is that applications in category (1), which don't care about UTC and just need a timescale which is phase-continuous and reasonably frequency-accurate, should not need to be burdened with code to recover from UTC corrections since those are irrelevant to their operation. They should instead be provided with an alternate timescale whose rate of advance may be synchronized to a reference time source but which is guaranteed to always be phase continuous while the system is operating.

This leads one to the dual-timescale arrangement which is already common in many kernels. We define the timescale *time* to have an epoch defined by a relationship to UTC, for example the 1 January 1970 UTC epoch defined by POSIX. We also define a second timescale, *uptime*, to have an epoch close to the time when the system was booted but with no fixed relationship to UTC. We define a third time, *boottime*, by the relationship between *time* and *uptime*, in particular so that

$$time = boottime + uptime$$

is always true. That is, *boottime* is the epoch for *uptime* expressed in terms of *time*'s epoch (which makes it an approximation of the *time* when the system was booted). Finally, we define the rate of advance of *time* and *uptime* to always be identical, but maintain *uptime* so that it is phase continuous. Expressed in terms of the clock model, *time* and *uptime* are two clocks attached to the same oscillator which are maintained so that their "Period" registers are always identical, but with *Step* and *Leap* adjustments only being applied to the *time* clock's "Time" register.

Maintaining this relationship implies the following effects of clock adjustments on *time*, *uptime* and *boottime*. *Rate* and *slew* adjustments are applied to both the *time* and *uptime* timescales, leaving *boottime* unchanged. The definition of *uptime* as a frequency-accurate timescale implies that, strictly speaking, *rate* and *slew* adjustments should be made to correct oscillator frequency errors, with the *slew* being performed to correct offsets accumulated during past frequency excursions while the *rate* change is made to make the clock keep better time in the future. *Step* and *leap* adjustments, on the other hand, are applied only to the *time* timescale for the purpose of bringing it into alignment with UTC (in principle this should need to happen no more than once, except for leap seconds, since once the clock has been *Step*-adjusted to UTC all future adjustments will be corrections for frequency) and hence result in a corresponding change to *boottime* of the size of the adjustment. This means that *boottime* changes are always caused by discontinuous adjustments to *time* so applications can track *boottime* to determine when those adjustments have been made, and hence where there may be a need for recovery actions, along with the magnitude and direction of the adjustment.

Note that in this arrangement large *slew* adjustments (say to correct a large offset error, rather than to adjust for past offsets accumulated during a system clock frequency excursion), and *sloop* adjustments to implement leap seconds, violate the frequency-accurate maintenance policy for *uptime*, penalizing category (1) applications and increasing the time that category (2) applications suffer from less-than-accurate UTC synchronization, which is why I prefer large time offsets to be corrected by *step* adjustments and leap seconds to be implemented with a *leap* adjustment instead. There is no perfect solution, however, so all these options are available to those with other priorities. I would point out that a more complex option might allow slewing adjustments to *time* independent of *uptime*, but this has implementation and complexity costs which don't seem justified by the benefits.

Apart from this, the particular dual timescale approach we do suggest has some advantages. Since *uptime* is guaranteed to be phase continuous it is ideal for use by category (1) applications which are interested in measuring time intervals but not necessarily interested in UTC, but with the additional advantage that *uptime* timestamps may be converted to (the current estimate of) UTC, for display or logging, if *boottime* is made available to those applications as well. While *boottime* may (rarely) change the UTC timestamps computed from it will represent the system's best estimate of UTC at the time *boottime* is sampled while the *uptime* timestamps will remain valid no matter what adjustments need to be made to *time* to bring it into alignment with UTC. More than this, the fact that *uptime* is phase continuous makes it uniquely suitable for time stamping events, like packet arrivals and similar time-related occurrences, since the phase continuity makes the time when the event happened unambiguous and avoids the ambiguity of the *time* timescale should it need to be adjusted backwards to align with UTC or to implement leap seconds. Those applications which do need UTC timestamps can compute them unambiguously from *uptime* timestamps if they have knowledge of when discontinuities in *time* happened (with the latter conveniently measured in *uptime* timestamps as well), which suggests a facility that might be usefully implemented as part of a time adjustment system call interface.

This, of course, leaves those applications that do care about UTC, or about time synchronization with networked neighbors, to consume *time* instead. There can be no guarantee that there will be no discontinuous changes to *time* (at the operating system level; one could perhaps design sufficiently robust hardware from which to obtain UTC time to ensure that the system's internal clock was always kept close) as this is an

inherent cost of offering a timescale which promises alignment with an external standard: if *time* is wrong something will need to fix it. As a consequence applications which need to deal with *time* will need to be equipped to handle the issues raised by this, or to recover from errors already made as best they can (which may mean living with them), informed by indications of the size of the change measured in *boottime* and/or a record of when past discontinuities have occurred, should the system maintain this. This leaves category (4) applications with some remaining difficulties, but it is difficult to see how these can be fully satisfied short of implementing a third timescale. It is hoped the rareness of *step* adjustment events will be sufficient to allow them to live with the remaining warts.

In summary, we maintain two timescales, *time* and *uptime*, which always advance at the same rate. The *uptime* timescale has an arbitrary epoch: it counts from zero when the system is booted. The *time* timescale is maintained with respect to a UTC epoch. The *boottime* variable is maintained such that

$$time = boottime + uptime$$

at all times. That is, *boottime* represents *uptime*'s epoch in terms of *time*. We now turn our attention to the programming interface used to maintain these.

6. Clock Adjustment Programming Interface

The programming interface proposed here attempts to meet the following goals:

- It should implement the clock model of Section 2, and be mathematically precise in the fashion discussed in Section 3. That is, the effect of adjustments should be entirely predictable and should provide enough information about the operation being performed to allow other clocks to be maintained as a function of the system time.
- It should provide two timescales to time consumers, a phase-continuous *uptime* and a UTC-aligned *time*, as explained in section 5. The difference between the two, *boottime,* should also be available to time consumers to allow conversions from *uptime* to *time* and vice versa.
- It should limit itself to the use of 64-bit quantities for all values. While longer variables might provide useful functions (e.g. to ensure that time roll-over's are not even theoretically possible) they make computation more difficult. All modern compilers provide support for 64-bit integer math, so we'll limit ourselves to this as best we can and defer using wider variables where they might otherwise make sense to some future time when compiler support for 128 bit integer math is common. Note that, to maintain 64 bit precision at the programming interface, an implementation might require internal library support for certain intermediate operations in wider precision.
- It should provide the maximum precision possible given the 64-bit constraint. This means that when faced with a tradeoff between existing standard formats which make less than full use of the 64 bits and a non-standard format which makes full use of the precision, we choose the latter *for the adjustment programming interface*. The standard system calls which provide time in standard formats will of course continue to be maintained.
- It should provide the adjusting application with full control of the clock. In particular, the programming interface should provide mechanism but avoid implementing, or enforcing,

any particular policy. Note that the downside of this is that the interface itself does not constrain applications to behave economically with respect to the costs they might impose on the system (e.g. frequent adjustments are likely to be functionally costly in many implementations), nor does it constrain operations based on some measure of "reasonableness" unless the lack of "reasonableness" also places some undue burden on the implementation. The adjustment application is assumed to know what it is doing.

Several primitive types are defined for use by the interface, probably with the following definitions:

```
typedef uint64_t systime_t;
typedef int64_t  sysrate_t;
typedef uint64_t sysfreq_t;
```

The `systime_t` type is used to express times and time offsets. Its format is essentially the same fixed-point format used for NTP timestamps; that is, the high order 32 bits represent integer seconds while the low order 32 bits are maintained as a binary fraction of a second. The `systime_t` type is an unsigned type. This means that times represented as a `systime_t` are always positive (i.e. in the future) with respect to their epochs, while only the magnitude of a time offset can be represented in a `systime_t` with the sign of the offset being carried elsewhere. While this may be inconvenient in some circumstances it does provide a full 136 years from the timescale epoch to a rollover, with a time precision of 233 picoseconds at the low order bit. Since the time epoch of *uptime* is the time the system was booted, the system is unlikely to stay up long enough in practice to roll *uptime*. The *time* timescale is a more difficult issue, but here we do not require the epoch for *time* as represented at the adjustment interface to be the POSIX epoch. The *time* epoch (in terms of the POSIX epoch) is instead made available separately, and is assumed to be set to ensure that at least 60 years of system uptime will not roll over the *time* representation. This is imperfect, but seems to be the best that can be done while observing the 64 bit constraint.

The `sysrate_t` type is used to represent clock rate changes. It is a signed 64 bit fractional value in the range [-0.5, 0.5). What a `sysrate_t` value represents is a ratio of two clock rates, less 1. That is, a `sysrate_t` value of *r* changes the clock rate by a factor of *(1 + r)*. Positive values hence make the clock advance faster while negative values make the clock advance more slowly. The precision of a `sysrate_t` is about 5×10^{-20}, or about 3 orders of magnitude more precise than it is currently possible to measure a frequency. Multiplying a `sysrate_t` value by 1,000,000 will produce a rate change in units of parts per million, while multiplying by 1,000,000,000 will produce parts per billion units.

Two interpretations of `sysrate_t` rate ratios are used. The first, a *relative rate*, expresses a rate change as a fraction of the clock's currently programmed rate of advance. This form can be used to specify *rate* adjustments via the programming interface, and is always used to specify a *slew* rate. The second, an *absolute rate*, specifies the clock rate with respect to the system's nominal rate of advance for this clock, i.e. the rate which the system thinks is correct for this clock (which is probably related to the frequency written on the case of the clock's oscillator), so that an *absolute rate* of 0 sets the clock to the nominal system value. A *rate* adjustment may be specified as an absolute rate, and *rate* values returned by the programming interface (other than for *slew* adjustments) are always expressed as *absolute rate* changes to avoid accumulating

round-off errors. An *absolute rate* is also an appropriate value to store across a reboot to return the clock to its corrected rate after the reboot.

The `sysfreq_t` is used by the system to report a clock's nominal "frequency", and is an unsigned 64 bit value expressed in units of Hertz. What this value represents is system- and clock-dependent, but is likely to be the update frequency of a counter clock or some other hardware device underlying the system clock. It is also the value the system offers to define the nominal clock rate which *absolute rates* are expressed in relation to. The reason this is offered is partly one of art; it allows the expression of rate changes in units of Hz, rather than as a dimensionless ratio, if there is a reason to want that. What is more important, however, is that it allows systems which do not know the nominal hardware frequency of a clock, and which hence estimate it by measuring it against some other clock at boot time, to present the possibly-variable results of this measurement to the adjustment application. This in turn allows an adjustment application which has stored a pre-reboot *absolute rate* and the corresponding frequency of the same clock to make adjustments to the rate to account for a different nominal frequency determined for the clock after the reboot before reestablishing the previous rate in the kernel. That is, systems which often measure a different clock frequency after a reboot can have their previous rate programming restored by adjusting it to reflect the changed nominal rate.

There is one other primitive data type which is frequently used, that being the *sysclockid_t* type. Its definition is probably a plain integer type of the form:

```
typedef int sysclockid_t;
```

It is not uncommon for systems to have multiple hardware time sources that can be used as the bases for independently adjustable clocks. A reason for providing multiple clock support to clock adjustment applications may include providing access to clocks unsuitable for use as the "system clock", that is the common clock whose time is made available to time-consuming applications at the POSIX interface, but which may be used for the timestamping of events in hardware (frequently these are implemented in Ethernet MAC chips to provide IEEE 1588 support). Another reason may be that the system has a variety of hardware which may potentially be used as the time source for the system clock, but particular revisions of the hardware cause particular clocks to have impairments, like a non-constant rate or other misfeature, which makes those clocks unusable for timing in some situations; making all such clocks available at the interface can allow an inappropriate choice of system clock made by a generic kernel unaware of these peculiarities to be corrected after the fact. The programming interface provides methods to read and perform adjustment operations on all such clocks, to compare pairs of clocks and to select a different clock to use as the system clock source.

Each clock in the system is uniquely identified by a small, non-zero positive integer of type `sysclockid_t`. It is hoped (but not necessarily required) that a system will identify each available clock with the same value across reboots. In any case the integers are expected to be assigned to clocks consecutively, starting with 1, so that an application may determine all clocks which are present by polling starting at a `sysclockid_t` of 1 and incrementing until it finds a clock not present. A `sysclockid_t` of 0 is a distinguished value used to identify the clock which is currently in use as the system clock. Each adjustment function includes a `sysclockid_t` as an argument to specify which clock the operation should be performed on; most uses will typically specify this value as 0 to operate on the system clock. Note that the particular operation of changing the clock to be

used as the system clock is a mostly manual function done by a user space application, and generally involves polling the two clocks to determine an adjustment, then adjusting the new clock to match the current system clock as closely as possible, and then switching the system to use the new clock. As this is inherently imperfect, and may have some undesirable side effects, it is expected that any switch will be done after a reboot and before a significant number of daemons have started. Not implementing this functionality at all if possible is a recommended alternative.

Beyond this, several structures are defined to carry information in and out of the system. The first of these is the `struct sysclock_info` structure, which is used to pass out information about a particular clock. Its fields include the following:

```
sysclockid_t   sci_id;             /* ID of this clock */
int16_t        sci_prio;           /* OS-defined quality */
int16_t        sci_flags;          /* Misc. flags */
sysfreq_t      sci_hz_nominal;     /* "nominal" frequency */
systime_t      sci_precison;       /* resolvable time diff */
sysrate_t      sci_initrate;       /* init rate (usually 0) */
sysrate_t      sci_maxrate;        /* maximum abs rate */
sysrate_t      sci_minrate;        /* minimum abs rate */
sysrate_t      sci_rateprec;       /* minimum rate change */
time_t         sci_epoch;          /* epoch for time */
char           sci_name[SCI_MAXNAME];   /* clock name */
```

The structure is filled in by a call to:

```
int sysclock_info(sysclockid_t id, struct sysclock_info *sci);
```

The `id` argument indicates the clock whose information is to be copied into the structure pointed to by `sci`. A value of 0 returns the information for the clock being used as the system clock. The information returned for a particular clock is constant, and will not change while the system is in operation.

The `sci_id` field holds the (non-zero) ID of the clock. If the `id` argument was non-zero `sci_id` will match it, otherwise `sci_id` will indicate the ID of the system clock.

The `sci_prio` field is filled in with a system-determined value which is used to pick the default system clock. The value is arbitrary, but the user should note that bigger values are better and negative values indicate that the system has a reason to believe that this clock should not be used as the system clock.

The `sci_flags` field holds a number of flag bits providing other information about the clock. Only one flag is imagined currently, that being

```
SYSCI_F_MEMMAPPED
```

which indicates that this clock supports the mapping of conversion information to process space to support a system call free time-of-day API (this clock would generally be required to be based on a hardware counter accessible to user space processes).

The sci_hz_nominal field holds the nominal clock update "frequency", in Hertz, from which the clock's nominal rate (referred to by absolute rate adjustments) is derived. Note that changes to this value for a particular hardware clock across a reboot probably indicates that the system is estimating this value; the ratio of the before-reboot and after-reboot values may be used to adjust absolute rates recorded before the reboot to values appropriate for the new estimate.

The sci_precision value indicates the minimum time that can be resolved by this clock or, equivalently, the amount of time the clock's time is advanced by at each increment at the nominal rate. In general this value will be equal to the reciprocal of sci_hz_nominal, though it is not constrained to be this if the clock hardware is somehow unconventional. The value is rounded up to the nearest representable time which is larger than the actual value, so the value in the sci_precision field will be at least 1 (i.e. 233 ps).

The sci_initrate indicates the default, initial absolute rate the clock is set to at boot time. This value will generally be zero. The exception to this might be if the nominal clock update frequency as expressed in sci_hz_nominal is small, perhaps because the underlying clock is being maintained by counting clock interrupts, and the actual nominal frequency of the interrupt clock is known to be a non-integer value when measured in Hertz. In this case the value of sci_hz_nominal would be set to the nearest integer Hertz value less than the actual frequency of the clock (rounding to lower is always slightly better because the negative range of a sysrate_t is larger than the positive range) while sci_initrate would be set to a (negative) value representing the adjustment to account for the fractional Hz value. Note that even when sci_initrate is non-zero, absolute rate adjustments are still expressed as relative to zero, so that absolute rate values will always be valid across a reboot as long as sci_hz_nominal for the clock is the same, even if sci_initrate is in fact being estimated and varies across reboots.

The sci_minrate and the sci_maxrate fields indicate the lowest and highest absolute rates which are supported by the clock, both for persistent *rate* adjustments and temporarily during *slew* adjustments. The value in sci_minrate will be smaller than the value of sci_initrate (i.e. normally less than zero) while the value in sci_maxrate will be larger. The API may return errors for operations which specify rates outside these bounds. Note that these values are not intended to artificially constrain the *rate* and *slew* adjustments which can be programmed (that would be a matter of policy, which this interface is intended to leave to the adjusting application to the extent possible) but rather to reflect real constraints imposed by the underlying clock hardware and its adjustment maintenance code. For example, it is quite likely that sci_maxrate in particular may limit the range of positive rate adjustments to less than the range of a sysrate_t since increasing the clock rate will generally require increasing the magnitude of some field internal to the clock maintenance procedures, which may in turn require the limit to avoid an overflow of that field. In any case, an implementation should endeavor to provide at least 5000 ppm of adjustment range in both directions unless there is absolutely no way to avoid a narrower restriction. If the implementation has no particular reason to implement restrictions these fields should be set to their extreme values.

The sci_rateprec field indicates the smallest change of rate which the implementation for this clock is likely to be able to perform. Note that an implementation is not required to perform any adjustment precisely as specified, it is only required to return a precise

indication of the adjustment it did perform in response to a request for an operation, so while operations may be requested with a full 64 bits of precision an implementation may carry fewer bits than this in its internal procedures. Request parameters may hence be rounded to lower precision values, with the implementation being required only to describe what it did do with full precision. The `sci_rateprec` field (which will be positive) provides an estimate of the magnitude of the roundoff which may be used by applications to determine which rate adjustments are too small to be worth asking the clock code to make (since the result of round off will probably be to leave the clock at its current rate).

The `sci_epoch` field represents the UTC epoch for *time* in terms of the standard POSIX epoch. That is, to determine the seconds value when converting *time* values to standard POSIX time values with a January 1, 1970 epoch requires adding the high order 32 bits of *time* to the value in `sci_epoch`. If the `sci_epoch` field is non-zero (if it is zero then *time* values are computed with respect to the POSIX epoch directly) the system will have set it to increase the lifetime of the *time* timescale before rollover. Since *time* is only intended to represent UTC-aligned times “now”, `sci_epoch` may be set by the system based on the time (in UTC) when the kernel was compiled, or some other time which is guaranteed to be earlier than “now” to maximize the range of future times which can be represented by *time* without rollover (at the expense of limiting the possibility of representing past times). The system guarantees that `sci_epoch` will be the same for every clock in the system (it is strictly a system parameter rather than a per-clock parameter) and that it will represent a UTC time in the past with respect to when the system was booted. This is clearly a hack to minimize the constraints caused by the interface’s 64 bit *time* representation on systems which represent the time with a larger number of bits.

Finally, the `sci_name` field holds a name for this clock which is suitable for pretty-printing. The name is no more than SCI_MAXNAME (probably 32 or so) characters long, is zero-padded if it is shorter than SCI_MAXNAME, and may contain any printable ASCII character other than “” (i.e. a double quote character, hex 0x22), the latter to make life easy for applications which wish to print it in quote-delineated form.

The call to `sysclock_info()` will return 0 on success, and may return one of two errors if the operation fails. It will return `ENOENT` if the clock indicated by `id` does not exist (it will never return `ENOENT` if the value of `id` is zero), or `EINVAL` if there is something wrong with the memory address passed in the `sci` argument.

The next function is used to request clock adjustments:

```
int sysclock_adjust(sysclockid_t id, int op,
                    const struct sysclock_adjust *adj,
                    struct sysclock_adjust *ret_adj);
```

The `id` argument specifies the clock to be adjusted; as always, a value of 0 requests that the adjustment be performed on the current system clock. The `adj` argument points at a structure describing the adjustment which is requested while the `ret_adj` argument points at a structure which, on a successful return, is filled in with a description of the adjustment which was performed (or was initiated, at least) as a result of the request. The `struct sysclock_adjust` structure contains at least the following fields:

```
systime_t sca_offset;    /* time offset for step/slew op */
sysrate_t sca_rate;      /* rate for rate/slew op */
```

```
systime_t sca_uptime;    /* uptime when change implemented */
```

It should be clear that `sca_offset` will generally hold the magnitude of a time offset, `sca_rate` will hold a rate and `sca_uptime` will hold the value of *uptime* when some event occurred or is scheduled to occur. The particular contents of these values is dependent on the value of `op`, which may be one of the following:

```
SYSCLOCK_OP_QUERY
SYSCLOCK_OP_STEP
SYSCLOCK_OP_UPSTEP
SYSCLOCK_OP_RATE
SYSCLOCK_OP_ABSRATE
SYSCLOCK_OP_SLEW
SYSCLOCK_OP_LEAP
SYSCLOCK_OP_SLOOP
SYSCLOCK_OP_ABORT
```

Note that for certain of these operations, in particular `SYSCLOCK_OP_SLEW`, `SYSCLOCK_OP_LEAP` and `SYSCLOCK_OP_SLOOP`, a request only initiates an adjustment operation which may require a considerable amount of time to complete. Once one of these operations has been successfully requested, subsequent requests for adjustments will fail (returning `EBUSY`) until the previous operation has fully completed, or unless it is terminated early by requesting a `SYSCLOCK_OP_ABORT` operation. This failure condition may occur for any of the operations described subsequently unless other behavior is explicitly documented. Also note that, unless a contrary requirement is explicitly mentioned, `ret_adj` may always be specified as NULL if the returned values are uninteresting to the application.

The `SYSCLOCK_OP_QUERY` operation is a query operation about the current state of the clock. The value of the `adj` pointer is ignored (and may be NULL), while the structure pointed to by `ret_adj` is filled in by a successful call with values representing the state of the clock (note that this is the only case where a NULL `ret_adj` will return an error). This operation may be requested even when a previous operation is still in progress; in this case the values returned will reflect the state of the clock once the operation completes. The values returned are as follows:

- The returned `sca_offset` field will be set to an estimate of the magnitude of the offset yet to be implemented by an incomplete `SYSCLOCK_OP_SLEW`, `SYSCLOCK_OP_LEAP` or `SYSCLOCK_OP_SLOOP` in progress at the time of the query. Note that the returned value is a magnitude only, the direction may be determined only from knowledge of the operation previously initiated. A returned value of 0 in this field indicates that there was no incomplete operation in progress at the time of the call.
- The returned `sca_rate` will be the absolute rate the clock is programmed to. Immediately after the system boots this value will be equal to the value of the `sci_initrate` field in the `sysclock_info` structure for this clock, otherwise it will be the last value programmed via the API. If a slew operation is in progress the value will indicate the rate that will be returned to after the slew is complete. Knowing this value may be useful to determine the size of a relative rate which can be specified for a *slew* without exceeding the `sci_minrate` or `sci_maxrate` limits, for example.

- The returned `sca_uptime` field will be set to the *uptime* when the most recent adjustment completed. If the most recent adjustment is still in progress the value will be in the future and will indicate the *uptime* when the adjustment will finish if it is allowed to continue to completion. This value might be useful to determine the time at which an in-progress adjustment will complete.

A `SYSCLOCK_OP_STEP` operation requests a step offset adjustment to *time* alone (hence causing a change of the same magnitude to *boottime*), leaving *uptime* uneffected. The request structure pointed to by `adj` needs to be valid, with the `sca_offset` field set to the magnitude of the offset while the sign of `sca_rate` is used to specify the direction (the offset is added if `sca_rate` is positive, subtracted otherwise). In setting `sca_rate` for the request the constants `SYSCLOCK_RATE_MAX` (the extreme positive `sysrate_t` value) and `SYSCLOCK_RATE_MIN` (the extreme negative value) might be useful. The value of `sca_uptime` in the `adj` structure is ignored. On successful return from the request the contents of `ret_adj` (if it is non-NULL) will be set as follows: The `sca_offset` field will be set to the magnitude of the offset actually performed, and ideally will be identical to the request value; an implementation should endeavor to make offset adjustments precisely as requested, though a prudent application might not assume that the returned value is exactly what was requested in case some implementation arrangement makes this impossible. The `sca_rate` field will be set to `SYSCLOCK_RATE_MIN` or `SYSCLOCK_RATE_MAX` to indicate the direction of the offset. The `sca_uptime` field will be filled in with the value of *uptime* when the offset was implemented. This time may in fact be slightly in the future should an implementation require this; the implementation is not required to complete the operation by the time of the return to the caller, it is only required to guarantee that an immediately subsequent call to make another adjustment will not fail because of the delay in scheduling the previous adjustment. It is the duty of all implementations, even those which cannot implement a requested step with full precision, to return to the caller a full precision indication of what in fact was done (in `sca_offset`) and when it was (or will be) done (in `sca_uptime`). The contract with the application is not that adjustments will be done to the requested precision, but rather that the adjustment actually implemented will be precisely reported to the caller.

A `SYSCLOCK_OP_UPSTEP` requests a step offset adjustment to both *time* and *uptime*, hence leaving *boottime* unchanged. It operates pretty much identically to a `SYSCLOCK_OP_STEP`, though the discontinuous change to *uptime* requires some elaboration on the interpretation of the value returned in `sca_uptime` in the `ret_adj` structure: The value returned is the time of the change measured in terms of the post-change *uptime*. The corresponding pre-change *uptime* for the change can be computed by adding or subtracting, as appropriate (and indicated by the sign of `sca_rate`), the returned value in `sca_offset`.

It is clear that `SYSCLOCK_OP_UPSTEP` causes a violation of the phase-continuous advance of *uptime* which is this timescale's sole reason for existence, and as such an adjustment application likely should never perform this operation on the clock in use as the system clock (though the adjustment interface might not enforce this constraint, in keeping with the principle that the adjustment interface should provide mechanism but leave policy to the user). The purpose of this operation is to allow the *uptime* timescale of a clock other than the system clock to be brought into alignment with the system *uptime*, perhaps in preparation for a change of the system clock. It also might be useful (perhaps internal to the kernel) to advance both *time* and *uptime* of the system clock after the system awakes

from a sleep, to reflect the amount of time which passed without the counter the system clock is derived from advancing.

The SYSCLOCK_OP_RATE operation requests a relative rate change to the rate of advance of both *time* and *uptime*, that is a rate change expressed as a ratio of the new rate to the currently programmed rate. The request structure pointed to be adj needs to be valid. The sca_rate field indicates the magnitude and direction of the relative rate change (positive increases the clock's rate of advance, negative decreases it), the values of the sca_offset and sca_uptime fields are ignored. If the request is successful the structure pointed to by ret_adj will be filled in as follows: the sca_offset field will be zero, sca_rate will be filled in by the absolute (note the absolute!) rate that the clock is now programmed to as a result of the change, and sca_uptime will be set to the value of *uptime* when this change took, or will take, effect. Note that, depending on the implementation, the *uptime* returned for the rate change may be a bit into the future; even if this is the case, however, it is guaranteed that immediately subsequent adjustment operations will not fail because of this (though their returned *uptime* values may be similarly shifted into the future).

Note well that the absolute rate resulting from a SYSCLOCK_OP_RATE operation may not be exactly the rate one would have expected given knowledge of the previous absolute rate; it may be rounded off to a lower precision representation if that is what the implementation supports. What an implementation must endeavor to guarantee, however, is that the actual absolute rate the system ends up at, which is returned in the sca_rate field of the return structure, will be represented to the full precision of that field, and that sca_uptime will be returned with a full precision indication of the *uptime* when the rate change occurred, as is necessary to ensure that pre- and post-adjustment timestamps are precisely convertible (see Section 3).

The SYSCLOCK_OP_ABSRATE operation is identical to the SYSCLOCK_OP_RATE operation except that the sca_rate field of the adj structure is interpreted as an absolute rate rather than a relative rate. The returned values have identical semantics for both operations. The pair of operations exists as solely as a convenience to the programmer, who is free to choose which to use for any particular adjustment. If there is a difference between the two it is only that clock rate errors, which are normally measured by comparing the rate of advance of the system clock to some more stable time source, can be determined and applied to the system clock as a relative rate adjustment independent of the current *rate* setting of the clock, while an absolute rate adjustment can only be computed from a measured rate error with knowledge of the clock's current absolute rate (though the fact that absolute rates are returned in ret_adj in either case suggests that keeping track of the clock's current absolute rate might not be a hardship).

The SYSCLOCK_OP_SLEW operation requests an offset adjustment to the values of both *time* and *uptime* which is accumulated at the specified relative rate. The request structure pointed to by adj needs to be valid. The sca_offset is filled in with the magnitude of the time offset to be accumulated, while the sca_rate field is filled in with the relative rate of the slew with the sign of sca_rate indicating whether the offset is added (i.e. the rate of advance of the clock is increased until a positive offset of magnitude sca_offset accumulates) or subtracted (i.e. the rate of advance of the clock is retarded until a negative offset of magnitude sca_offset accumulates). The value of the sca_uptime field is ignored. Note that the operation is performed by making a temporary adjustment to

the clock's rate of relative magnitude and direction indicated by `sca_rate`, for precisely the length of time required to accumulate a time adjustment of magnitude `sca_offset`, after which the clock's *rate* is returned to the previous value. No subsequent adjustment may be performed until this one has completed (which may take a considerable amount of time if the magnitude of the relative rate specified is small or the offset is large). An implementation is allowed to decline to perform the operation if the requested parameters describe a slew which would take more than a day or so to complete; to execute a longer slew the application should divide it into multiple, shorter requests.

The values returned in the structure pointed to by `ret_adj` are as follows: The value of `sca_offset` indicates the magnitude of the offset adjustment actually performed and ideally should be identical to the requested offset (an implementation should endeavor to make this so if at all possible), though an adjustment application would be wise not to count on this and to accept whatever is returned. The value of `sca_rate` is returned as the actual relative (note the relative!) rate used for the slew; this should be as close to the requested relative rate as an implementation can manage, though if it must differ the implementation should choose a rate which is larger in magnitude than the requested value so that the slew takes less (not more) time to complete than the requestor might have planned. Note, however, that this rate may not be accurately reported if the slew rate is extreme and the rounded value implemented (which will be larger in magnitude than the request) exceeds the representable range of a `sysrate_t`. Should this occur the implementation may still perform the slew but will return a rate which is precisely `SYSCLOCK_RATE_MIN` or `SYSCLOCK_RATE_MAX` to indicate what has happened; in this case the actual rate may be deduced by doing a subsequent `SYSCLOCK_OP_QUERY` to determine the termination time. The value of `sca_uptime` returned is the time, measured as *uptime*, when the rate change which initiated the slew was made. The returned values should describe the operation actually performed (or, at least, initiated) with full precision even if the operation performed is not precisely what was requested. Note that, except when `sca_rate` is returned as an extreme value, the time interval over which the slew is performed should be precisely computable by dividing the returned value of `sca_offset` by the magnitude of the returned value of `sca_rate` (and the value of *uptime* when the slew will complete may then be obtained by adding the time interval computed to `sca_uptime`), though this will require an extended precision divide operation. This time may instead be determined by asking the system via a `SYSCLOCK_OP_QUERY` after the operation is initiated, which will return the value of *uptime* when the slew will complete as `sca_uptime`, with the rate being computed from the slew offset divided by the time interval. Apart from extreme rate values, if the two methods do not produce exactly the same time (within maybe a bit of roundoff error) then someone has screwed up.

The `SYSCLOCK_OP_LEAP` operation requests a step offset to be done at some time in the future. This request is identical to a `SYSCLOCK_OP_STEP`, except that the value of `sca_uptime` in the request's `adj` structure is taken to be the the value of *uptime* when the step requested is to occur. Note that if `sca_uptime` has a value of *uptime* which is already past the step operation will be performed immediately (or as "immediately" as a `SYSCLOCK_OP_STEP` would do it, the operation might still be deferred to a bit into the future). The value of `sca_uptime` returned in the `ret_adj` structure will be the *uptime* that the step was actually scheduled for (unless the requested *uptime* was already past, in which case you'll get more or less the same result that a `SYSCLOCK_OP_STEP` would have returned), and should be the same as the requested *uptime* to within the precision of the clock (as indicated by the `sysclock_info` structure's `sci_precision` field). Clearly, if

this operation is used to implement a leap second, the requested sca_uptime would likely be set close to midnight UTC on the day of the leap (perhaps at midnight, or one second after, depending on which "standard" you prefer), and would be computed by taking the *time* representation of the UTC time of the leap and subtracting *boottime* from it. An implementation is allowed to balk if a leap operation is scheduled too far into the future, say more than a day.

The SYSCLOCK_OP_SLOOP operation requests that a slew offset be initiated at some specified time in the future. The request is identical to a SYSCLOCK_OP_SLEW, except that the value of sca_uptime in the request's adj structure is taken to be the value of *uptime* at which the slew is requested to be started. Note that (and this should all be familiar from the explanation of SYSCLOCK_OP_LEAP) if the start *uptime* requested via sca_uptime is already past the slew requested will be started immediately, just as it would be for a SYSCLOCK_OP_SLEW operation; otherwise the start of the slew operation will be deferred to the requested time. The values returned in the ret_adj structure will be the same as for SYSCLOCK_OP_SLEW except that (assuming the requested *uptime* was not already past) the value of sca_uptime returned should be very close to (i.e. within the clock's precision of) the requested time, and will indicate the time the slew will begin. The time of the end of the slew may be computed using one of the same methods suggested when describing SYSCLOCK_OP_SLEW. An implementation may return an error if either the wait to the start of the slew or the duration of the slew itself would exceed a day.

The SYSCLOCK_OP_ABORT operation can be used to terminate an in-progress, but incomplete, SYSCLOCK_OP_SLEW, SYSCLOCK_OP_LEAP or SYSCLOCK_OP_SLOOP operation before its normal completion. An abort might be done if the previous operation was errant, or if something has changed to make the application change its mind about what should be done; if the abort executes successfully it is guaranteed that a subsequent adjustment will not fail with an EBUSY error. The normal result of an abort will be to leave some portion of the offset adjustment the previous operation was initiated to perform undone; the magnitude and direction of the undone portion of the offset will be returned. The adj argument is ignored for this operation and may be NULL. The structure pointed to by ret_adj, if present, will be filled in with information precisely describing the undone portion of the offset at the time of the abort. In particular, the sca_offset field will hold the magnitude of the undone offset, while the sign of sca_rate will indicate the offset direction (the value of sca_rate will be the relative slew rate for an aborted slew or sloop, or one of SYSCLOCK_RATE_MIN or SYSCLOCK_RATE_MAX for an aborted leap). The value of sca_uptime will be set to the *uptime* when the operation was aborted. Note that an abort operation will succeed even if no incomplete operation is currently in progress. In this case, sca_offset will be returned with a value of 0 while the remaining fields have the values that a SYSCLOCK_OP_QUERY adjustment would return. Clearly, aborting a leap operation will always return either the full requested offset or 0 in sca_offset since either the leap was completed or it wasn't. Aborting a slew should always return some fraction of originally requested offset which is less than the entire request; the returned value of sca_uptime should be consistent with this fraction. An aborted sloop may return the full offset or a fraction depending on whether the abort was requested prior or subsequent to the start of the slew.

A call to sysclock_adjust() will return 0 if successful, or one of a handful of errors if not. EINVAL will be returned in response to a negative id, an unknown value of op, when adj or ret_adj is NULL when the structure is required by the operation, or when there is

something wrong with the memory locations referenced by those pointers. ENOENT will be returned if id refers to a clock which does not exist. EBUSY will be returned to a call requesting an adjustment if the previous adjustment has not yet completed. E2BIG may be returned if an operation would take too long, for some implementation-defined value of that (applications should endeavor to schedule operations which complete in less than one day). ERANGE may be returned if the magnitude of the rate specified for a rate or slew adjustment is too large to be implemented by the system (see sci_minrate and sci_maxrate in the sysclock_info structure). EAGAIN may be returned if the implementation needs to avoid dealing with adjustments for a period; this error return should be extremely rare (and an implementation should endeavor to make that so), but exists in case it is necessary for the implementation to hold off adjustments while dealing with an internal hardware counter overflow or some other rarely occurring implementation-dependent exception. If an error is returned the structure pointed to by ret_adj will be unchanged by the function.

The next two entry points are used to read a clock's time. Neither of these is intended to replace the normal POSIX system interface functions which return the time, but rather are provided for adjustment applications to use to query times in the format of the adjustment interface. Both functions return the time in the form of a struct systimes with the following members:

```
systime_t sct_uptime;
systime_t sct_boottime;
```

The value returned in sct_uptime is the clock's *uptime* at the time of the call, while the value of sct_boottime is the clock's current value of *boottime*. Clearly, *time* may be obtained by adding the two values.

The two functions which return this structure are:

```
int sysclock_gettime(sysclockid_t id, struct systimes *sct);
int sysclock_time(struct systimes *sct);
```

The obvious difference between the two functions is that sysclock_gettime() takes a clock ID as an argument and can hence be used to read the time of any clock in the system, while sysclock_time() exclusively returns the time of clock in use as the system clock (i.e. the clock whose time would be returned by a call to sysclock_gettime() with an id of 0). The more subtle difference between the two may be present in systems which allow the mapping of clock conversion information to user space to support system-call-free time of day calls. In this case sysclock_time() may return a time using this method, avoiding a system call, but sysclock_gettime() is implemented to always make a system call even if the lower-overhead method exists. This difference may be useful for debugging.

Note that both functions return 0 on success, or an error if there is a failure. EINVAL is returned by either function if there is a problem with the address in sct, and by sysclock_gettime() if the id value is negative. The latter may also return ENOENT if id refers to a clock which does not exist.

When an implementation provides more than one clock the next function may be used to measure the time of one clock against another:

```
int sysclock_poll(sysclockid_t id0, sysclockid_t id1,
                  struct sysclock_poll *scp);
```

The members of the `sysclock_poll` structure are:

```
systime_t scp_uptime0_early;
systime_t scp_uptime1_early;
systime_t scp_uptime1_late;
systime_t scp_uptime0_late;
```

The system executes this request by first obtaining a timestamp from the clock identified by `id0`, followed by a timestamp from `id1`, followed by a second timestamp from `id0`. The earlier timestamp taken from `id0` is stored in `scp_uptime0_early` while the later timestamp is stored in `scp_uptime0_late`. The timestamp taken from `id1` is stored in `scp_uptime1_early` and likely copied into `scp_uptime1_late` as well (though I am acquainted with clock hardware which supports returning both an "early" and "late" timestamp from a single poll, so an application might be better written not assume that `scp_uptime1_early` and `scp_uptime1_late` are always identical). The timestamps returned are *uptime* timestamps from each clock. Assuming the delay between obtaining $early_0$ and $early_1$, and between $late_1$ and $late_0$, are equal the uptime offset of clock `id1` from clock `id0` may be computed as

$$offset_{10} = (early_1 + late_1)/2 - (early_0 + late_0)/2$$

with the `id0` clock *uptime* when the uptime offset was $offset_{10}$ (clearly the offset will be continuously changing with time if the clock rates are different) being

$$uptime_0 = (early_0 + late_0)/2$$

and with the magnitude of the possible error in the offset made by assuming equal delays between the timestamps being

$$error_{10} = \pm((late_0 - early_0) - (late_1 - early_1))/2$$

Note that the implementation should endeavor to obtain these timestamps as close together in time as possible while observing the sequencing since doing so will minimize the magnitude of $error_{10}$. Despite this, however, an application should treat the values from a single poll as noisy data, to be improved by repetition and filtering, since both the limited precision of one or both clocks and events like interrupts (should the implementation allow them) may make any individual sample suspect. The purpose of this function is to provide data to allow the time of one clock to be adjusted into synchronization with the time of another clock, perhaps because one clock is a measurement clock (not the system clock) which has been adjusted into alignment with UTC by being used to measure IEEE 1588 protocol packets, and we would like the system clock to track the now-accurate measurement clock, or perhaps to bring a clock's time into alignment with the system clock in preparation for a change of the system clock.

The function returns 0 when successful, or an error code for failure; if the call fails the structure pointed to by `sct` will be unmodified. EINVAL is returned if one or both of the clock ID values is negative or if there is a problem with the memory pointed to by `sct`. ENOENT is returned if one of the clock IDs refers to a non-existent clock. ENXIO is returned if this function is unimplemented (it might not be if the system only has one clock).

The last function is relevant only for implementations which provide support for changing the system clock at run time. Note that support for changing the system clock is not considered a feature of this API, in fact it is better viewed as compensation for hardware defects. Implementations which are required to run on a wide variety of hardware may have a number of potential hardware sources of time available, but various misfeatures of particular versions of the hardware may make time sources which are eminently suitable for use as a system clock on some hardware platforms distinctly unsuitable on others (this might be true, for example, if the rate of a hardware counter was guaranteed to be constant on some platforms but varies on others), so to allow production of generic kernels which may not be able to predict these hardware misfeatures the interface allows a kernel to support multiple clocks and let the user choose between them instead. Implementations which can better predict the characteristics of the hardware they are written to run on are encouraged to instead pick just one hardware source (the “best” one, measured by reliability first and precision second) to use as the system clock and to provide only this, plus useful measurement clocks, to the adjustment application.

The function to change the system clock source, should this be implemented, is:

```
int sysclock_changeto(sysclockid_t id);
```

The value of `id` should be non-zero and indicate the clock to use for the system clock. It is up to the caller to have ensured that the current system clock and the clock referred to by `id` are closely enough synchronized (both *time* and *uptime*) not to impair the operation of any concurrently active processes. An implementation is also not required to constrain itself in a way which guarantees this operation will always be safe for time-consuming applications. For example, an implementation is permitted to timestamp received packets in the device driver with a raw timestamp which is only converted to *uptime* when the packet is received at a socket which wants it. A change of the system clock may result in a gross error in a timestamp by applying the wrong clock’s conversion to the raw timestamp, depending on the implementation; the application must accept this error since an implementation is not required to correct it. Furthermore, it is possible for an implementation to prohibit clock changes once applications are running, in particular when the current system clock supports memory mapping of conversion data and one or more applications have already mapped the segment. Essentially, `sysclock_changeto()` is only guaranteed to execute safely, or at all, if called very soon after the system has booted and before any applications which might be intolerant of errors have been started. The capability of changing the system clock is not intended to be a general feature, it is rather a workaround for systems which have no way to determine the general health and quality of the selection of clock hardware the platform might support during the bootstrap process and hence need to rely on an administrative selection made shortly after booting.

The function returns 0 on success, or an error code on failure. EINVAL is returned when `id` is zero or negative, and ENOENT is returned when `id` refers to a non-existent clock. EPERM may be returned when the system can no longer make a change to the system clock because applications have accessed the current system clock in a way which makes

a change impossible. ENXIO may be returned if the function is unsupported by an implementation (which it might not be if the system itself can reliably select the best reliable clock source).

These 6 function calls represent the full adjustment interface. I believe they can support a wide variety of underlying implementations using clock hardware with a wide variety of attributes. The only property that adjustment applications rely on is that the adjustments which are made in response to requests are accurately reported to the application even if the adjustments implemented are somewhat at variance with the requests. The mathematical accuracy of these reports provides isolation between the measurement process of an adjustment application and the process of system clock adjustment; they can be maintained independently of each other to the precision that unadjusted clock timestamps can be determined from the timestamps of the adjusted clock, and an implementation must strive to ensure that this precision is high. What remains, then, is to make a convincing argument that the adjustment API is indeed implementable.

7. Hints at a Possible Implementation

This section attempts to sketch out an example implementation of clock maintenance code that is consistent with our clock model and that makes use of the adjustment interface above. The intent is not to present a complete, or even a realistic, implementation of the interface, but rather to describe enough of an approach to be able to demonstrate both that the adjustment interface is indeed implementable with the precision specified and that an implementation of the interface can also provide a very high quality, highly functional system clock.

The general goals of the implementation are as follows:

- To ensure that the arithmetic related to accurate reporting of clock adjustments can be done correctly and precisely. This is the fundamental requirement of an implementation since it is this property which allows clock synchronization software to perform adjustments in an open-loop fashion.
- To provide a very “clean” clock whose rate of advance is as jitter-free as the underlying clock hardware. This mostly means avoiding making adjustments to the clock at clock interrupt time the way clock maintenance is traditionally implemented, thereby incorporating clock interrupt latency jitter into the clock (note that this does not imply that we won’t support system clocks implemented by counting clock interrupts, only that if the clock hardware can be used in a way which avoids interrupt processing we will avoid it).
- To allow raw hardware “tickstamps” to be obtained and recorded very cheaply, deferring the work of converting these to real timestamps until sometime later. The intent of this is to make it feasible and inexpensive to, say, “tickstamp” every arriving packet, or generic interrupt event, in the interrupt routine of the hardware driver but to defer as much of the cost of converting this to a time to the point where the packet is found to be destined to a socket or other process interested in receiving timestamps (perhaps donating unused “tickstamps” to the random number entropy pool so nothing is wasted).
- To maintain conversion data (to convert “tickstamps” to timestamps) in a fashion such that readers who are using the conversion data will receive consistent data without locking even when an update process is concurrently active. This is maximally efficient

for SMP systems (since conversions will be far more frequent than adjustments avoiding a reader lock has large benefits) and is also a prerequisite for providing the data directly to user application space to allow times to be determined without system calls.

To perform the arithmetic to the precision required we will need several extended precision unsigned integer math functions. While most of the sample code which uses them is relegated to Appendix A these are documented here to provide an indication of how precise computations on the data types described in the previous section are possible. These use two structure data types. The first, a `u128int_t`, is used to represent an unsigned 128 bit integer and has the following members:

```
uint64_t lo;
uint64_t hi;
```

`lo` holds the low order 64 bits of the 128 bit integer, while `hi` holds the high order 64 bits. The second data type, a `u128div_t`, holds the results of unsigned long integer divisions and has as members:

```
uint64_t quot;
uint64_t rem;
```

`quot` holds the quotient resulting from a divide, while `rem` is the remainder.

The functions we use are described next. These particular functions are chosen because they can often be executed using hardware instructions available on 64 bit instruction set processors, while being quite straight forward to provide library support for on 32 bit machines. The first of these is:

```
int uclz64(uint64_t v);
```

This function returns the index of the highest order bit set (i.e. 1-valued) in `v`. 0 is returned if the highest order bit is set, while 63 is returned if the value of `v` is 1. This function is often available as a single instruction on 64 bit machines since it is useful for divide normalization.

```
u128div_t udiv128(u128int_t num, uint64_t denom);
```

This function returns the 64 bit quotient and remainder of the unsigned division of the 128 bit integer `num` by the 64 bit integer `denom`. It is the responsibility of the caller to ensure that `num` and `denom` are selected such that the division does not overflow the 64 bit value holding the resulting quotient. This function can be implemented as a single (slow!) hardware instruction on the x86-64 architecture, though implementation as a library function in standard C is straight forward as well. The function is only used for intermediate computations to implement adjustments.

```
u128int_t umul128(uint64_t a, uint64_t b);
```

This function returns the 128 bit product of the unsigned 64 bit integers `a` and `b`. Most 64 bit instruction sets have hardware instructions to compute this directly, though for other architectures a library implementation is straight forward and only modestly more costly than what is required to compute a 64x64=64 bit multiply on the same architecture.

```
uint64_t umul128hi(uint64_t a, uint64_t b);
```

This function returns the high order 64 bits of the 128 bit product of `a` and `b`. Clearly this returns the same value that `umul128(a,b).hi` does, but it exists as a separate function because this particular operation is very frequently executed and implementing this function separately may provide optimization advantages.

We now turn to the structure of the clock code itself. The fundamental hardware clock data type we compute times from is the `timecount_t`, which is likely defined as:

```
typedef uint64_t timecount_t;
```

A `timecount_t` is the value of 64 bit counter which is best initialized to approximately zero when the system is booted and is continuously incremented by 1 at the frequency of some hardware oscillator thereafter. This is the form that a hardware clock is represented in when it is presented to the clock-independent conversion code; it represents the output of a "Counter Clock", as defined in Section 3. Note that the `timecount_t` value will in general be a "cooked" value. That is, while a `timecount_t` might ideally be derived directly from the sampled output of some 64 bit wide hardware counter, it is not constrained to this. It might instead, for example, be a variable which is incremented in the interrupt service routine for a hardware clock interrupt, or its low order bits might come from a narrower-than-64-bit hardware counter with its high order bits being supplied by software associated with the clock, or it might even be computed from the value of a count-down counter used for the clock interrupt with clock-dependent software supplying the remaining bits and the conversion code. The goal of the clock-dependent code is to produce a 64 bit counter clock value to be used by the clock independent conversion. Note that the constraint that the counter start at zero at boot time is a soft constraint related to avoiding the need to run code to deal with a counter rollover. While rollover can be dealt with it is a bit awkward, and initializing the counter to zero ensures that a rollover will take at least 60 years of uptime at a counter rate of less than 8 GHz or so, which should safely avoid ever having to run the code to deal with a rollover.

Given a `timecount_t` value of *tc*, we perform the following computation to determine the time (reviewing Section 3 may make this clear):

$$t = r * (tc << s) + c$$

Here, *t* is computed to be the time in `systime_t` format. The value of *s* is selected so that a second's worth of advance of *tc* will cause a change to bit 32 in the shifted result at least once during the second, and is hence a constant for the clock related to the clock's increment frequency, while *r* and *c* are maintained by the adjustment software to produce a correct result. Note that the multiply returns the high order 64 bits of the 128 product, so the computation might be performed in code as:

```
t = umul128hi(r, tc << s) + c;
```

It might also be noted that the value of *s*, as defined, is a direct function of the clock's frequency (maybe not the nominal frequency, but rather the lowest frequency we might expect the clock to run at) and could be computed from a `sysfreq_t` value as follows:

```
s = uclz64(f) - 31;
if (s < 0) s = 0;
```

As we are directly computing a `systime_t` result, the offset *c* will also be a `systime_t` value. We maintain two different values of *c*, *c_time* and *c_uptime* where *boottime* (as a `systime_t`) is *(c_time - c_uptime)*, and select the appropriate constant to produce a result in the desired timescale. The task for the adjustment code, then, is to compute the changes to *r*, *c_time* and *c_uptime* required to implement a requested adjustment, and to publish these so time consumers may make use of them to convert hardware times to standard *time* or *uptime* values. It is probably clear now what the adjustment implementation will entail. A step adjustment will be implemented by modifying *c_time* appropriately, a rate adjustment by modifying *r* as requested with corresponding changes to *c_time* and *c_uptime* to maintain phase continuity at the time of the change, and a slew adjustment will temporarily change *r* (and *c_time* and *c_uptime*) for the duration of the slew before changing *r* back to its original value with *c_time* and *c_uptime* offset appropriately. Note that if *s* is selected to be the minimum value necessary for the computation to work then *r* will generally have at least 63 significant bits in its range (i.e. the high order bit will be 1) and will hence be able to implement rate adjustments to within less than a bit of the full precision of their `sysrate_t` specification.

Before moving on to the organization of the adjustment code, however, it might be useful to look at where `timecount_t` values come from. As was noted above we expect that the full 64 bit counter representation will generally not be (conveniently) available directly from the hardware, but instead may require some computation and additional software state management to derive from whatever the hardware time source provides. As such, we do not expect that a “tickstamp” returned by a call to sample the hardware clock will return a fully processed `timecount_t`, but instead will return a more primitive, clock-dependent value in the form of a `tickcount_t`, which will likely be defined by:

```
typedef uint64_t tickcount_t;
```

What a `tickcount_t` value holds is entirely opaque outside the context of the clock dependent support code. It is instead intended to be the most convenient and cheapest-to-obtain representation of the hardware clock’s value that can be converted to a `timecount_t` at some later point. For example, if the underlying hardware counter is a full 64 bits and it is convenient for the processor to sample it, then the `tickcount_t` value might be the full value of the counter and the conversion to a `timecount_t` might be trivial. On a 32 bit processor, however, sampling the full 64 bits of a hardware counter might be inconvenient (or, on any processor, the counter might not be 64 bits long) so there a better option might be to sample only the low order 32 bits of the counter and to fill in the high order, software-maintained 32 bits to form a `timecount_t` only later when the value is converted for use. And a really awkward time source, like the count register of a count-down timer used for clock interrupt generation, might just have the raw data from the counter dumped into the `tickcount_t`, leaving as much of the math to turn this into a full count as possible to the conversion time. The idea here is that obtaining a `tickcount_t` should be just as cheap as it can possibly be, deferring whatever additional work is required to form a `timecount_t` to the point where a time will be derived from it, so that the implementation of a system which “tickstamps” every packet which arrives but processes only the fraction which are delivered to a socket which cares about arrival timestamps into a time (with anything else being used for random number fodder) should be as inexpensive as possible.

Making obtaining a `tickcount_t` really cheap, and designing how the data required for the conversion to a `timecount_t` is maintained, is hence the major design work needed to support a clock source. It is also clear that a system which works this way may need to make judgement calls about the probability of errors introduced by this; for example a 32 bit counter which rolls over in a second may be converted in error unless the conversion is performed in less than a second after the `tickcount_t` is sampled. The general approach should be to try to make an error improbable in normal circumstances rather than eliminating the possibility entirely. If taking more than a second to get a packet from the interface driver to the socket might result in an error in converting the "tickstamp" taken in the driver this is probably acceptable since, if it is taking more than a second to get the packet from the interface driver to the socket the system probably has many much larger, more pressing problems to contend with than a timestamp in error.

In any case, the procedure for obtaining time from a clock is hence as follows:

- Obtain a `tickcount_t` from the clock-dependent code.
- Call the clock-dependent code to convert the `tickcount_t` into a `timecount_t` counter.
- Call the clock-independent code to convert the `timecount_t` to a time value in the desired timescale.

For the last bit we need a method for storing conversion constants (the *r* and the two *c* values). As we may be performing this conversion well after the original `tickcount_t` was obtained we would like to maintain a certain amount of history as well so that late conversions to time still end up at the same values they would have if done earlier, and since we wish to do conversions without locking we need a structure which can be updated concurrently with conversions. We might, then, store an individual set of conversion constants in a `struct sysclock_set` having the following members:

```
timecount_t     ss_tc;
uint64_t        ss_r;
systime_t       ss_c_time;
systime_t       ss_c_uptime;
```

Note that the `ss_tc` member of a set contains the `timecount_t` value at, or after, which this set of constants is valid. The `ss_r`, `ss_c_time` and `ss_c_uptime` values are used as described above. To allow past `timecount_t` values to be accurately converted, and to permit scheduling of future adjustments, we will maintain an array of these in a larger structure, a `struct sysclock_conv`, which has the following members:

```
int             sc_s;     /* `s' shift constant */
int             sc_end;   /* index of end array entry */
volatile int    sc_latest;     /* most recent array entry */
volatile struct sysclock_set  sc_sets[1 /* [0..sc_end] */];
```

Here the `sc_s` member records the shift for the `timecount_t` value prior to multiplication, `sc_last` provides the size of the array and `sc_latest` is the most recent entry added.

Given a `timecount_t` value `tc` and a set of conversion constants pointed to by `sc` a procedure for finding the appropriate conversion constants and applying them might be as follows:

```
     volatile struct sysclock_set *ss, *ss_end;
     timecount_t tc_set;

     ss_end = &sc->sc_sets[sc->sc_end];
try_again:
     ss = &sc->sc_sets[sc->sc_latest];
     while ((tc_set = ss->ss_tc) > tc) {
          if (ss == ss_end) {
               ss = sc->sc_sets;
          } else {
               ss++;
          }
     }

     ret_time = umul128hi(ss->ss_r, tc << sc->sc_s) +
                ss->ss_c_time;

     /* ret_uptime = umul128hi(ss->ss_r, tc << sc->sc_s) +
                ss->ss_c_uptime; */
     /* ret_boottime = ss->ss_c_time - ss->ss_c_uptime; */
     membar_consumer();
     if (tc_set != ss->ss_tc) {
          goto try_again;     /* changed while we were looking? */
     }
```

That is, we search forward from the latest entry to find the first with an `ss_tc` less than or equal to our target `tc` and use those constants for our conversion. This search always succeeds since the structure maintenance code always sets the `ss_tc` member of the oldest valid entry in the list to zero. Generally the value of `ss_tc` decreases as we advance through the array. The structure maintenance code also endeavors to guarantee that a change to the `ss_tc` value of an array entry always precedes (and is made globally visible before) a change to the other data in an entry, so the final check should detect a possible concurrent change to the data being read and responds by redoing the search.

Note that while this looks like a potentially expensive operation, in fact at an appropriate rate of adjustments the typical case will be that the first entry checked will be the one which is used, with searches beyond the first two or three entries being exceedingly rare, essentially only being required when a very long time has elapsed between the taking of the timecount and its conversion. In most cases the conversion would be expected to be done very close to the time when a timecount was obtained, though keeping a fairly sizable array (perhaps a significant fraction of the machine's page size if it is to be mapped to applications) might not be a hardship and will enable accurate conversion of quite old timecounts. And conversion constants will always be found, though extremely old timecounts might be converted using constants newer than the timecount.

It is probably also worth pointing out that the value computed by

```
umul128hi(ss->ss_r, ss->ss_tc << sc->sc_s) + ss->ss_c_uptime;
```

is the value of *uptime* at which a change is implemented. This is the value of the `sca_uptime` member of the adjustment structure returned after an adjustment. It should also begin to be clear how an adjustment would be implemented, that is by filling in new entries at the start of the array (i.e. just before the `sc_latest` entry), making sure these are globally visible and then updating the `sc_latest` index to atomically add them to a consumer's search. An implementation might keep a fraction of the array unused at any particular time (the segment of the array which is "in use" is that between the `sc_latest` entry and the first entry with a zero `ss_tc` value) to make it very unlikely that late references remain when an entry is reused, though late references should be detectable by `ss_tc` changes to the entry in any case. Note that to implement an immediate adjustment, when the adjustment process adds an array entry it will probably use an `ss_tc` value which is a bit into the future (we'll call that time "now") to enable it to fill in the new entry (or entries) and put them into service before that that timecount is reached by the hardware clock. To do this perfectly might require blocking interrupts so that the process doing the update has the exclusive use of the processor it is running on between taking the timecount timestamp and putting the adjustment into service, though an occasional error which is caused by not doing this might not have serious enough consequences to worry about.

While Appendix A has sample (toy) code implementing clock adjustments using this representation, it is probably worth summarizing how each type of adjustment might be performed. Any of the *step* adjustments (`SYSCLOCK_OP_STEP`, `SYSCLOCK_OP_UPSTEP` or `SYSCLOCK_OP_LEAP`) would be performed by adding one entry to the start of the array with `ss_tc` set to "now" (or, for a leap, sometime in the future) with the `ss_r` and `ss_c_uptime` constants copied from the previous entry but with `ss_c_time` adjusted by the requested offset (for a `SYSCLOCK_OP_UPSTEP` the `ss_c_uptime` constant would be similarly modified). Either of the *rate* adjustments would be performed by adding one entry to the start of the array with `ss_r` modified appropriately and with the `ss_c_time` and `ss_c_uptime` constants adjusted (by identical offsets) to preserve phase continuity with the previous entry at the time (i.e. `ss_tc`) of the change. Either of the *slew* adjustments, on the other hand, would be implemented by adding two new entries to the start of the array. The earlier one would be set with an `ss_tc` of "now" for a `SYSCLOCK_OP_SLEW`, or perhaps some time into the future for a `SYSCLOCK_OP_SLOOP`, with the `ss_r` constant reflecting the slew rate adjustment with the other constants adjusted to preserve phase continuity at the start of the slew. The second, later (after-the-slew) entry would have its `ss_r` copied from the current head of the list and the other constants adjusted by the requested offset, with the value of `ss_tc` being chosen so that phase continuity is preserved at the end of the slew. Note that in all these cases the detection of an incomplete adjustment (requiring an `EBUSY` error to be returned) is done in the same way: when the `timecount_t` value representing "now" is obtained it is compared to the `ss_tc` value at the current head of the array. If the latter is larger the previous adjustment is incomplete.

An *abort* is hence performed by adding an entry with a `ss_tc` of "now" to the front of the array, overriding (a portion of) the incomplete adjustment behind it. When the operation being aborted is a `SYSCLOCK_OP_LEAP` the constants in the abort array entry will be copied from the entry prior to the leap (i.e. without the leap's offset). Since the abort entry

will have an ss_tc which is smaller than that of the entry for the leap being aborted, the leap itself will never be found by a search for constants. The “undone” offset returned by the abort will be the offset of the leap (perhaps computed by comparing ss_c_time in the leap entry to ss_c_time in the abort entry). Similarly, aborting a SYSCLOCK_OP_SLEW will set ss_r to the pre-slew value, but with ss_c_time and ss_c_uptime offset by the amount of slew accumulated at the ss_tc abort time (which will be smaller than the full offset). The “undone” offset to return for the abort can again be computed by comparing ss_c_time in the abort array entry to the previous array entry (which represents the value it would have had had the slew been allowed to complete). Aborting a SYSCLOCK_OP_SLOOP is done the same way as one or the other of the above depending on whether the slew scheduled had started at the abort’s “now” ss_tc or not.

Supporting the abort operation has some implications for what metadata is maintained about the state of the array and, in particular, what kind of operation each entry resulted from. The abort will need to know whether it is aborting a slew or a leap to do its work correctly, so it needs to know what process added the most recent entries. Furthermore, the selection of an entry to terminate the array by setting its ss_tc value to zero should also be sensitive to what the entry represents (note that as entries are added to the front of the array for new adjustments, the process which does this should probably also be reclaiming old entries from the end of the array by advancing where the zero ss_tc entry is). In particular, the end of the array probably shouldn’t be set to an entry which was subsequently aborted, nor to the intermediate entry implementing a slew (its ss_r constant will be changed such that it isn’t useful for extrapolating times beyond the time interval when the slew was active, which is essentially what the end-of-array entry is used for), so having metadata describing what the oldest entries are will allow the reclaim process to advance the zeroed ss_tc to an appropriate entry.

It is probably worth noting another complexity here, that of dealing with timecounter_t value overflows. There are actually two ways an overflow can occur: the full timecounter_t value tc can wrap around to zero, or the shifted value (tc << s) can wrap to zero while retaining non-zero high order bits in the unshifted timecounter. Note that if the timecount_t value is initialized to zero when the system is booted then both of these events will take at least 60 years of uptime to occur at increment frequencies of less than 8.6 GHz (a frequency which, given the current state of the art, may take quite a few years for any hardware to reach), and even at 17.2 GHz the rollover is 30 years off. The trouble with assuming this is sufficient time to ensure this will never happen, however, is that it relies on counters which count from zero when the system is booted and it is possible that there might be hardware counters that do not, or are inconvenient to, reset this way, so prudence may demand that code to handle rollovers be written anyway, just in case.

The shifted overflow is easier to deal with since the unshifted timecount, which is stored in the conversion array, will still advance and it is only the conversion constants which must change. The adjustment code must be written to avoid scheduling an adjustment which crosses the rollover, most obviously slews and leaps but also any other adjustment as well (since even adjustments made “now” will be scheduled for a future time); the error code returned when this would happen is EAGAIN. Second, at some point close to, but safely before, the rollover, an entry needs to be added to the front of the the array with an ss_tc value set to the first rolled-over timecount (one bits in the *s* high order bits, but zero bits everywhere else) with ss_r from the previous entry but with the *c* constants adjusted for

the rollover (probably by adding the value of `ss_r` to them). Once the counter passes the rollover normal operations can continue on as usual.

The unshifted rollover is a bigger problem, since it requires a guess at how long a period `tickcounter_t` and/or `timecounter_t` timestamps might live in the system before they are converted (since there is no way to know this with certainty, a guess is all we have). Once you have the guess interval (a few seconds is probably realistic, a few minutes if you are a pessimist), begin by erring out (`EAGAIN`) adjustment requests at twice this interval before the rollover occurs to stabilize the conversion array contents. Then, at one interval before the rollover, rewrite the end of the array (the zero `ss_tc` entry) with constants appropriate for post-rollover timecounts (see above); as long as no timestamps older than the next-to-oldest array entry are extant in the system (which there shouldn't be if there have been no recent adjustments, since the next-oldest entry will still be old) this will be error free. Once the counter rolls over, processes with post-rollover timestamps will search right to the end of the array to convert them and hence both they and any straggling pre-rollover timestamps will be converted appropriately. Once enough time has past for pre-rollover timestamps to have been consumed, say one more guessed time interval past the rollover, a new zero `ss_tc` entry can be added to the front of the array, with the same constants as the entry at the tail, and normal operations can resume. While this is awkward and imperfect it may be good enough, and while rollovers like this should rarely (if not never) occur it is satisfying to know there is a way to handle them if they do.

This scheme is not the only way the adjustment interface could be implemented, in fact the adjustment interface itself imposes few implementation constraints beyond the precision of the values it returns describing the true effect of each adjustment. As a representative implementation, however, it is useful to look at some of its characteristics. The implementation allows a considerable separation between the time a `timecount_t` (or, more accurately, a `tickcount_t`) value is sampled and it is converted to a time value by retaining older conversion information for some time. This is useful for situations where `tickcount_t` values can be sampled considerably before the conversion, both in the kernel (sample `tickcount_t` in the device driver, convert it at the socket) and in user space where scheduler interrupts can keep a process from completing a conversion for an extended period of time. Since the array is consumed, and hence reclaimed, at a rate of one or two entries per adjustment then if the rate at which adjustments are programmed is suitably modest (ideally no more than is necessary to keep the clock optimally precise given the quality of the data used to synchronize it) even a modest amount of memory by today's standards can preserve a considerable length of history.

It is worth emphasizing that (with the possible exception of counter rollovers, which should be exceedingly rare) the only source of changes to the conversion data is adjustment requests. The data is never modified at clock interrupt time, as is typical of clock conversion management now, and is never modified at all unless some application chooses to make adjustments (with the exception, perhaps, of rollover handling, but this should rarely happen). This makes a modest rate of adjustment not only economical for maintaining sufficient adjustment data for late timestamp conversions, and for ensuring that timestamp conversions will generally never have to search past the earliest entry or two to find their constants, but is also efficient in terms of esoteric considerations such as multiprocessor cache performance, which can be an issue with frequently accessed data like this. Cache invalidations will occur only when the data is modified, and the data will only be modified when a new adjustment is made (as opposed to every clock interrupt, as often happens now) which means that at a modest rate of adjustments the data will use

CPU caches very effectively too. This probably implies that the implementation of this adjustment interface will provide not only a more accurate means of clock adjustment, and support additional functionality (e.g. user space conversions, or `tickcount_t` event timestamping), but will also likely be less expensive as measured by the processor resources consumed. Its primary cost is the multiply required for the conversion (or 3 lower precision multiplies on a 32 bit machine), but it has been a very long time since we've had machines where multiply performance was an issue at all in any sense, let alone relative to performance issues related to uncached memory references. To be clear, the interface by itself doesn't eliminate all clock interrupt processing related to system timekeeping since, for less-than-ideal hardware time sources, the maintenance of data to form `tickcount_t` values and/or the conversion from `tickcount_t` to `timecount_t` will often involve clock interrupt maintenance. For some subset of time sources (including those most likely to be useful for user space time conversion), however, the only action will be adjustments.

In summary, then, I think the proposed clock adjustment interface is not only useful for the precision it promises, but can provide this benefit with an implementation which is straight forward to write, economical to execute and which potentially provides functional benefits not available in current systems. The interface is not only attractive in specification, it is also attractive in implementation.

8. Editorial Summary and Conclusions

The adjustment API presented here was developed while experimenting with a PCI-X clock interface board. The board implemented a clock with a timestamp precision of 3.1 ns (i.e. a 320 MHz internal increment clock) which could be synchronized to the 10 MHz and 1 PPS output from a GPS receiver to that precision and which implemented a programming interface allowing that time to be transferred to the host processor clock with an ambiguity of less than +/-10 ns if done well. An implementation of the API presented here, using the processor cycle counter as a `tickcount_t` source, resulted in a system which would maintain the system clock in synchronization to the board's clock by sampling the difference between the processor and board clocks 4 times per second and, as a result, making a rate adjustment of about 10^{-9} and a slew offset of about 10 ns at a rate of once every 10 seconds, very roughly. The result was, arguably, a system whose clock was generally maintained to within about 20 ns of the GPS receiver's time. The time samples were received and processed by a user-space application which called the adjustment API to implement the required adjustments; while the kernel I started with had code to implement a similar function by processing sampled time data in the kernel I can think of absolutely no advantage, and many disadvantages (e.g. some computations are better done with floating point arithmetic) which result from that.

Doing this task successfully required a system clock with accuracy which was at least a match for the board and whose rate of advance could be accurately adjusted at the level of the frequency errors that could be measured. The clock in the system I started with could do neither. While that used the same counter as a timebase the rate steering was done during clock interrupt processing in a way which apparently incorporated both interrupt latency and some arithmetic approximations into the result, with the result that the system clock jittered at close to the microsecond level. The approach of deriving the system clock as a linear function of the underlying hardware counter, making interrupt-time adjustments unnecessary, was taken to avoid adding any jitter not present in the hardware

counter itself. The original code expressed clock rate corrections in units of nanoseconds per second, and made an additional math approximation which rounded the value to even lower precision, so the rate adjustments required in my situation disappeared into the roundoff error, which led to the implementation endeavoring to maintain as close as possible to 64 bits of precision in all calculations. The requirements for implementing time stamping as accurately as possible (and hence as cheaply as possible, since the time required to obtain a "tick stamp" from the clock directly contributed to the measurement uncertainty) when measuring against the clock board resulted in the systematic separation of the acquisition of raw `tickstamp_t` values from their conversion to system time values, a separation which proved useful for event time stamping in general, and seems equally useful for the implementation of a system-call-less time API for applications. And, finally, consideration of what was strictly required to avoid the complexities and constraints of implementing a feedback control algorithm to discipline the local clock, allowing one to concentrate entirely on the measurement and filtering problems, resulted in the clock model presented here and the guarantees of transparency with respect to the effect an adjustment has on the clock provided by the programming interface. The programming interface resulting from this seems both complete with respect to providing support for the needs of precision clock adjustment applications, and allows a kernel implementation which was both surprisingly straight forward to write and very compatible with an SMP environment.

It is probably worth summarizing several of the tradeoffs made in defining the clock model and its implementation, and their (debatable) justification. The error model for the clock, and hence for the oscillator driving it, is minimally simple; it allows a correction for phase errors and a correction for the rate of change of phase (i.e. *rate*, or frequency). Models of errors for precision clocks usually include a third parameter, a correction for the rate of change of frequency with time (*frequency drift*, or *aging*), but the variations of frequency of clock oscillators in most computers are so dominated by unpredictable and uncompensated environmental effects that there seemed to be no practical value in modeling drift errors that could justify the added cost and complexity for the implementation. If someone does arrange to clock their computer with a very stable oscillator they may dislike this tradeoff, though.

The second tradeoff is in the definition of the two timescales, *time* and *uptime*, which might be expected to appear at the POSIX system call interface as CLOCK_REALTIME and CLOCK_MONOTONIC, respectively. The UTC-aligned *time* and the arbitrary-epoch *uptime* are constrained to advance at the same rate with the difference between them, *boottime*, only changing when atomic *step* adjustments are made to correct *time* into alignment with UTC. The time interval applications which might be expected to be users of *uptime* might not care about UTC but are very likely to prefer, and be improved by, their time intervals being accurate with respect to the S.I. second, so maintaining a timescale which has an arbitrary epoch but which maintains a rate of advance which matches the rate of the S.I. second as best we can determine it seems like the ideal time source for these applications. Of course, a timescale which advances at the rate of the S.I. second as best we can determine must have a simple (if not quite unvarying) relationship to the system's UTC-aligned timescale, since the latter also advances at the (same) rate of the S.I. second as best we can determine it (this must also be true for the difference between POSIX CLOCK_REALTIME and CLOCK_MONOTONIC as I read the standard, unless they are also allowing the definition of "seconds and nanoseconds" to vary between clocks; the fact that they don't make the difference between the timescales available, a la *boottime*, makes the "monotonic" time distinctly less useful, however). Since determining the rate of the S.I. second as accurately as possible is the major work product of time

synchronization software, the connection between *time* and *uptime* such that they both receive the advantage of time synchronization seems like the best thing to do.

Despite this I must admit that the particular relationship between *time* and *uptime* chosen embeds a policy constraint (of the type I claimed to want to avoid) in the form of the choice to make (all) *slew* adjustments apply to both *time* and *uptime*. The problem with this choice, coupled with the aim that *uptime* be advanced at as close to the rate of the S.I. second as we can, implies that *slew* adjustments should only be used to adjust for phase errors accumulated due to prior frequency excursions. Use of a *slew* to implement a leap second adjustment, or to correct a small phase offset in *time* not related to a prior frequency excursion (which wouldn't necessarily eliminate *step* adjustments since there would remain the possibility of an error in *time* which is too large to *slew*) can only be done be degrading *uptime*'s frequency accuracy, so *uptime*'s frequency accuracy can only be maintained by the policy of making all *time* adjustments to correct phase errors by *step* adjustments. Worse, the form in which this arrangement causes *uptime* to track the rate of the S.I second, by phase locking, is itself a policy choice. A reasonable alternative, frequency locking (where *uptime*'s frequency is corrected, but phase errors from prior frequency excursions are not), can't be implemented. I would argue that this choice was made for non-policy reasons, both because it allows the simplest implementation and it results in changes to *boottime* being both discrete and rare, which makes that value much more useful. I believe this is a justifiable choice on its own merits, independent of my own policy preferences, but I must admit that my own preference that leap seconds and other phase adjustments to *time* be done as *step* adjustments may make me a biased judge.

The final tradeoff I would note is the use of a non-(POSIX-)standard time format for the system call interface, and a format which only carries a 32 bit seconds offset at that. The difficulty which led to this is that making use of the system call interface, while not difficult, requires one to do fairly careful arithmetic, both in the kernel and in the calling application. This code is significantly easier to write correctly if it is possible to write simple computations, like

```
time = boottime + uptime;
```

exactly like that. The time format must be one that allows arithmetic to be done in C without having to write function calls to perform even the simplest operations. The POSIX `struct timespec` is really unsuitable for this; it is a time format which seems designed to make printing times with `printf()` easy, but little else. The longest integer type you can count on C compiler support for is 64 bits, and if that is to fit a full timestamp you are limited at both the high and low order ends of the representation. I don't like the constraints, but I believe the choice works for the application and the warts it comes with are more than made up for by the ability to write clear code. I believe the ultimate "right" answer might be for a `systime_t` to become a 128 bit signed integer type or, if not that, then a system programming language which allows operations on a non-fundamental data type to be fully encapsulated, but the prerequisite for that would be C compiler support for a fundamental integer type of sufficient width or a different system programming language with object oriented features. Lacking either of these I think the choices made here allow an ease of writing software which justifies the warts and edge-case complexities of a slightly-too-frugal bit count for the application. I admit that reasonable people might differ, however.

These reservations aside I believe the API and timescale definitions presented here provide considerable utility. The work is based on a clock model which describes a computer clock in the unambiguous terms of the arithmetic it needs to perform, which in

turn allows a clear specification of the effects that adjustments to the clock should have and the considerations an implementation needs to address to provide a fully predictable and transparent adjustment interface. The two-timescale discipline provides a time source suitable for applications with requirements for measuring time intervals, while allowing timestamps to be converted to UTC for display or logging, and avoids the ambiguities that might otherwise occur if event timestamps were taken in a timescale which may require discontinuous phase adjustments to align it to UTC. The API provides mechanism without dictating a policy for its use, and is complete in the sense of fully exploiting the properties of the clock model.

Appendix A

[To Be Completed]