By Jeff Roberson,
Design team leader,
Eyring Corporation.

Accurate, High Resolution Absolute Timing on the PC Platform

After 16 years, The IBM® PC-AT compatible hardware platform is still alive and well. Today's PCs are plentiful, inexpensive and fast, with CPU clock rates now exceeding 1 GHz. However, the advances in real time operating system (RTOS) software timing capabilities have not necessarily kept pace with the hardware speed improvements. This article presents a design for an accurate, high resolution PC RTOS timing subsystem which takes full advantage of this extra CPU bandwidth to provide improved capabilities and services to real-time and embedded applications. By increasing the clock tick frequency and clock accumulator precision, the instantaneous and long term software clock errors can be reduced significantly. And the high speed clock allows tasks to be accurately scheduled in the absolute time domain with resolution that is improved by several orders of magnitude. EYRX® is a new PC RTOS from Eyring Corporation that incorporates these design concepts (see: http://www.eyrx.com). The timing performance improvements are impressive. For example; when running with a 200 kHz system clock tick frequency on a 500 MHz machine, this new timing subsystem allows Eyrx to 1.) provide a software clock with 5 microsecond resolution and accuracy that can be calibrated to within 32 microseconds per year, 2.) preemptively schedule 100,000 tasks per second with each task receiving a full two-tick timeslice, and 3.) awaken a sleeping task at a precise absolute time with a worst case delay of less than 15 microseconds. Data acquisition, process control, high speed networking, sequencers, robotic feedback loops and other time critical applications can all benefit from these improved operating system software timing capabilities.

PC CLOCK HARDWARE

On the PC platform, a 1.19318 MHz crystal provides the base reference frequency to the 8253/8254 Programmable Interval Timer (PIT) chip. Using channel 0 operating in mode 2 (Rate Generator), the PIT chip divides this base input frequency down by a programmable whole integer count (2 to 65536) and generates an output signal which pulses at this reduced frequency (18Hz to 596KHz). This periodic pulsing signal is fed into the IR0 input on the master 8259A Programmable Interrupt Controller (PIC) chip which generates an IRQ0 interrupt for each clock tick pulse received. The operating system IRQ0 Interrupt Service Routine (ISR) maintains clock time by adding a fixed amount of time to a master clock accumulator variable each time it services a tick interrupt. During system startup, this master clock variable is set using the time obtained from the MC146818 battery backed-up CMOS real time clock. The operating system's software clock time instantaneously jumps ahead by a quantum amount once each tick, then stays constant between ticks. Thus, while true time advances in a smooth continuous manner, the system software clock time jumps ahead in a periodic, stepwise, non-continuous manner.

TIMEKEEPING ACCURACY

This software clock time keeping methodology inherently introduces both instantaneous and long-term clock errors. These errors must be minimized by the RTOS to provide an accurate, high-resolution time base. For the sake of this discussion, we shall assume that the PC's time reference hardware crystal is perfectly stable and exhibits no frequency drift over time. We also assume that the software clock time exactly matches true time when it is initially set during the boot process.

INSTANTANEOUS CLOCK ERROR

Each time a running application calls the operating system to fetch the current time, the clock time value that is returned is always wrong by some finite amount. This is because at any given instant in time, the software clock time always lags true time by an amount which is limited to the clock tick time interval. This instantaneous clock error effect is illustrated in Figure 1, which plots software clock time (vertical axis) as a function of true time (horizontal axis). At true time Tt0 (11), a task calls the operating system, which returns the current software clock time Tc0 (8). The instantaneous clock error Terr0 is thus 11 - 8 or 3 time units. To illustrate the effect of frequency, at time 16 the system clock tick interrupt frequency is doubled and is doubled again at time 28. The actual clock tick interrupts are plotted along the bottom of the figure. Note

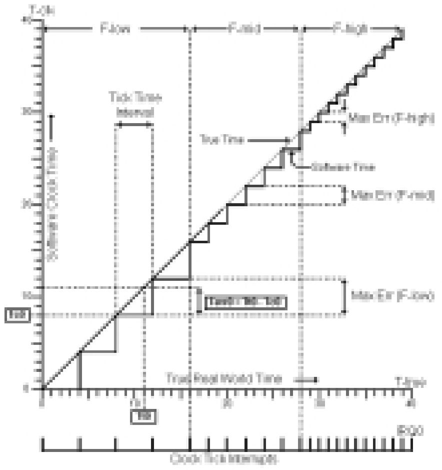

Figure 1. Long Term Clock Drift Error.

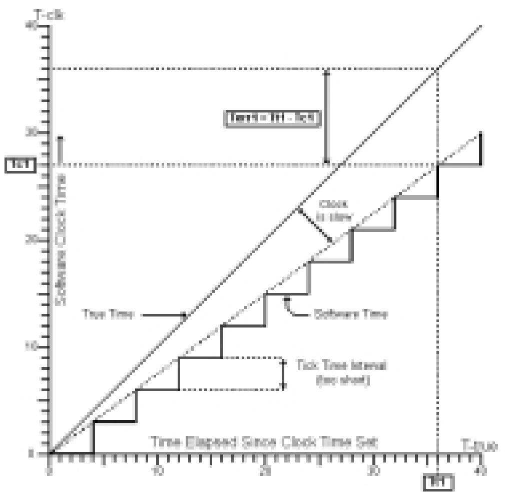

Figure 2. Long term clock drift error.

that as the tick interrupt frequency is increased, the worst case instantaneous clock error is proportionally decreased. This figure demonstrates that the worst-case instantaneous clock error is inversely proportional to the tick interrupt frequency.

If the amount of time added to the system clock each tick is not exactly correct, the software clock time will linearly drift away from true time. This long-term clock drift error effect is illustrated in Figure 2. In this (exaggerated) example, the amount of time that the tick ISR adds to the software clock is too small (is 3, should be 4). Thus, the clock is slow. At time zero, the system software clock is initially synchronized to the correct true time. Some time later, at true time Tt1 (36), a task calls the operating system, which returns the current software clock time Tc1 (27). The long-term clock drift error Terr1 is 36 - 27 or 9 time units. Note that in this example, the instantaneous error is negligible at Tt1, since the task fetched the time immediately after it was updated by a tick interrupt.

RTOS CLOCK DESIGN

Armed with an understanding of how clock errors occur, an RTOS timing subsystem can be designed to minimize these errors and thus achieve improved short and long term timekeeping accuracy. To minimize short-term instantaneous clock error, the tick interrupt frequency must be maximized. To minimize long-term clock drift error, the time increment that is added to the clock accumulator variable each tick must be adjustable to allow precise clock speed calibration. The following discussion describes the design of various time related RTOS components that combine to provide optimal system clock accuracy.

CLOCK VARIABLES

First, the resolution and size of the system clock variables must be chosen. Lets start with the clock accumulator variable, which is used to store the current time.

Unix clocks typically maintain the time elapsed since midnight, January 1, 1970 (a time otherwise known as the epoch). Since this is very common, we'll chose the same start time reference for our RTOS clock. For size, we could use a 32-bit integer and store the number of seconds elapsed since the epoch. This is obviously inadequate since a good RTOS requires timing resolution much finer than one second. Choosing a 64-bit integer with nanosecond resolution seems reasonable. This provides a nanosecond clock having a duration of 584 years that will expire in the year 2554 if the start time reference date is set to the epoch (i.e. same as Unix).

Next, the tick time interval variable must be defined. This variable contains the time quanta that is added to the clock accumulator once each tick. Since the tick ISR uses simple integer addition, the resolution of this interval variable must be nanoseconds to match the resolution of the accumulator variable. For size, a 32-bit integer is adequate to store the tick time interval in nanoseconds for all possible tick frequencies (i.e. 18 Hz to 596 kHz). Thus for our initial design, the tick ISR adds a 32-bit count of nanoseconds to a 64-bit master clock accumulator variable. However, this clock resolution shall later prove to be inadequate.

To reduce long term clock drift error, it was determined that the tick time interval must be adjustable to allow the clock speed to be calibrated. Since this time interval value is stored as an integer, the finest adjustment increment is one count (nanosecond). Note that this adjustment is applied once each clock tick, so the overall clock speed adjustability is affected by the clock frequency (slow clock frequencies provide finer overall adjustment resolution than higher frequencies). Lets check to see if our nanosecond clock design provides adequate speed adjustment resolution. With a relatively high clock tick frequency, say 100 kHz, the smallest possible tick interval adjustment (one nanosecond) has the following effect over a one-year period:

$$\text{TickInterval} = \frac{1}{\text{frequency}} = \frac{1}{100000\ \text{Hz}} = 0.000010\ \text{sec} = 10000\ \text{nsec}$$

$$\text{Adjustability} = \frac{1\ \text{nsec}}{\text{TickInterval}} = \frac{1\ \text{nsec}}{10000\ \text{nsec}} = \frac{8.64\ \text{sec}}{\text{day}} = \frac{52.596\ \text{min}}{\text{year}}$$

It can be seen that a one nanosecond per tick interval adjustment equates to 52 minutes per year when running at 100 kHz. This overall adjustment resolution is excessively coarse for our RTOS design. Thus, our initial nanosecond clock resolution proves to be inadequate.

By further extending the clock variables another 32-bits, the clock accumulator becomes 96-bits and the tick time interval becomes 64-bits. However, from a running application's point of view, the previous nanosecond clock design had more than enough resolution, so this extra 32-bits of clock resolution can be hidden from view and used internally by the operating system. The upper 64-bits of the 96-bit clock accumulator and the upper 32-bits of the 64-bit tick time interval variables shall retain their previous units (nanoseconds), and the extra 32-bits will be used to represent the fractional nanosecond portion. Thus, each clock accumulator count now represents exactly 2-32 nanoseconds or approximately 2.3283E-19 seconds.

Our new, extended precision clock now has the following overall adjustment resolution when running at 100 kHz:

$$\text{Adjustability} = \frac{2.3283\text{E-}19 \text{ sec}}{\text{TickInterval}} = \frac{2.3283\text{E-}19 \text{ sec}}{10000 \text{ nsec}} == \frac{0.735 \text{ microseconds}}{\text{year}}$$

This provides more than enough clock speed adjustment resolution (better than one microsecond per year). Thus, the size and resolution of our new clock's variables are now finalized. To summarize: At each clock tick, the ISR adds a 64-bit tick time interval to a 96-bit clock accumulator. Applications can only see the upper 64-bits of the system clock which represents the number of nanoseconds elapsed since the epoch.

PREEMPTIVE TIME SLICING

Our RTOS shall implement preemptive time slicing to provide timely response and to prevent any one task from hogging the CPU. Time is chopped up into timeslices which are distributed among runnable tasks. Once scheduled, each task gets to run for a maximum of one timeslice before the scheduler is re-invoked.

Note that a task can voluntarily relinquish the CPU at any time before the end of its timeslice, so the scheduler is not always invoked on a tick boundary. For optimum performance, each timeslice must therefore be comprised of two clock ticks. This is because each task's actual run time during the first tick of its timeslice will vary since the start time can randomly occur at any point between clock ticks. Thus, to guarantee that each task gets at least one whole tick in which to run, two ticks per timeslice are required. To implement this preemptive time slicing, each task shall have a ticks-to-go timeslice count variable in its task control block that is maintained by the clock tick ISR.

CLOCK TICK INTERRUPT SERVICE ROUTINE

Maximizing the system clock frequency implies minimizing the duration of the timer tick ISR. Since the tick ISR must be short, it can only do that which is absolutely necessary. The ISR must add the tick time interval to the system clock accumulator and decrement the current task's timeslice ticks-to-go count. When the task's count reaches zero, the ISR flags a task switch, reloads the timeslice count (for its next run) then invokes the scheduler.

SCHEDULER

Our new RTOS must allow a task to suspend on an absolute time event (i.e. until some arbitrary, specific, absolute time). And if the highest priority task in the system is suspended on an absolute time event, it must be awakened and scheduled in a quick, deterministic manner once its wake up time has been reached. These requirements influence our choice of scheduler. Two fundamental types of prioritized schedulers are commonly used. The main difference between the two is the location of the system overhead code that is responsible for waking up tasks.

A double-list scheduler type maintains two separate task lists. The first list contains all the tasks that are ready to run and is sorted by priority. The second list contains all the tasks that are suspended waiting for some event. This type of scheduler does not waste time scanning tasks that are suspended and is able to quickly find the next task to be run by simply picking the task that is currently at the top of the ready list. However, an RTOS that implements this double list scheduler type must also provide separate mechanisms to move tasks from the waiting list to the ready list (i.e. wake them up).

A single-list scheduler type maintains only one task list. All tasks, both running and suspended are placed in the single list, which is sorted by priority. As it scans looking for the highest priority runnable task, the scheduler is responsible for moving tasks from the suspended state to the running state. The highest priority runnable task in the list is always picked to be scheduled next.

If we chose the double list scheduler for our design, we must provide a way to move tasks from the waiting list to the ready list. Waking a task that is suspended on a non-time event does not present a problem since the sleeping task is quickly and deterministically awakened by an ISR or task that is responsible for handling the event. However, absolute time events do present a problem. With a double list scheduler, there are two ways to wake up tasks that are suspended on time events.

With the first method, the tick interrupt ISR scans all tasks that are suspended on time events, and wakes all those whose time has come. This method is unacceptable because the tick ISR duration must be minimized. The second method uses a separate, dedicated task which essentially does the same thing as the ISR, but from the task level rather than from the interrupt level. Both of these methods share a common problem, particularly when the system is running at a high frequency with lots of tasks. The entire task list must be scanned each invocation, which not only adds a lot of unnecessary overhead, it also requires a non-deterministic amount of time to complete. If 1000 tasks, including the highest priority task, are all suspended on time events, the highest priority task is delayed from running until all 1000 tasks have been scanned and possibly awakened. Thus, we conclude that the double list scheduler type is not appropriate for our high speed RTOS.

So we choose the single task list scheduler type, which works as follows: When a task suspends on an absolute time event, the 64-bit nanosecond wake up time is stored in its task control block. The tick ISR quickly and deterministically updates the system clock accumulator and handles preemptive timeslicing. When invoked, the scheduler first reads the clock time then proceeds to scan the task list starting with the highest priority. The wakeup conditions for each suspended task are tested in turn. If a clock tick occurs while scanning, the scheduler starts all over again from the top. Once a ready to run task is found, it is immediately scheduled. Thus, the time delay required to wake up and schedule any given task is proportional to the number of tasks which have higher priority (i.e.

high priority tasks are scheduled quicker than low priority tasks). In other words, the scheduler never wastes time waking up low priority tasks while high priority tasks are still ready to run.

TASK TIME SCHEDULING ACCURACY

So we now have an RTOS design that provides an accurate, high resolution clock and an ability to suspend tasks on absolute time events. So how well does it perform? When the time comes to wake up a task that is suspended on an absolute time event, the task is actually scheduled some time after the requested time has passed. Figure 3 illustrates the conditions required to produce the worst case for this task schedule time delay. For this example, the system clock tick frequency is 200 kHz with a corresponding time interval of 5 microseconds. With two ticks per timeslice, the scheduler is being invoked at a frequency of 100 kHz. TASK3 is the highest priority task in the system and is initially suspended on an absolute time event. Two other tasks, TASK1 and TASK2, are both actively running at a common (lower) priority and are being scheduled in a round-robin manner. The requested absolute wakeup time for TASK3 is Treq (21), which occurs just after a clock tick boundary (20). Thus, when time 20 comes along, the scheduler does not wake up TASK3. Instead, TASK1 is scheduled which runs for a full timeslice.

When TASK1 finishes its slice, TASK3 is finally scheduled at time Twake (34). The task schedule delay Tdelay is thus 34 - 21 or 13 microseconds. The figure graphically demonstrates that for fast clock frequencies, the worst case delay is limited to less than three clock ticks. (Note that for slow clock frequencies, the worst case delay is actually closer to two clock ticks.)

APPLICATIONS

With a software clock running at high frequencies, time is being chopped up into very fine slices. This provides several benefits: The system clock can provide accurate and high-resolution time keeping. Many tasks can be given a chance to run in a short period of time (when running at 200kHz, 100 active tasks can be scheduled in a single millisecond, with each task running for a full timeslice). And tasks can now be accurately scheduled in the absolute time domain with small worst case delays. Time sensitive applications that were previously scheduled with an accuracy on the order of tens of milliseconds, can now be scheduled to within tens of microseconds.

These new capabilities open up new application possibilities. A data acquisition system that receives data asynchronously could now provide accurate, precise time stamps for incoming data. For example, a packet-sniffer program that is responsible for monitoring high speed network activity, could time stamp packet arrival and departure times and compute more accurate delay times. Using absolute time events, a sequencer application could accurately generate external events with sub-millisecond precision. In the field of robotics, control feedback loops could be tightened up considerably and more tasks could run concurrently with no loss of response times, since many tasks can be scheduled in a very short time frame. Undoubtedly, other high speed process control applications could also benefit. This new RTOS design thus provides improved timekeeping accuracy and new timing capabilities to embedded/real-time/dedicated systems designers working with the PC platform.

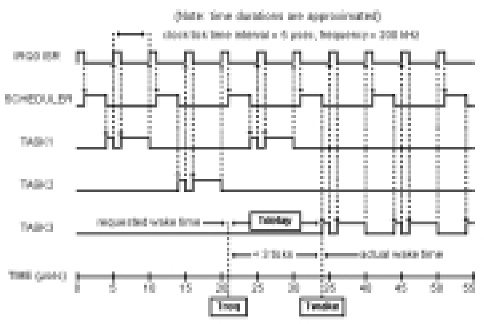

Figure 3. Worst case task schedule time delay.

EYRX®

Eyrx is a new RTOS for the PC platform from Eyring Corporation (http://www.eyring.com) which implements the design components just described. The clock speed is calibrated with a 32-bit dimensionless variable, which has units of picoseconds per second. This provides a clock calibration resolution of 31.6 microseconds per year. And since Eyrx has low overhead, it allows running the clock at very high frequencies indeed. For example: When running on a 1200 MHz processor, Eyrx allows setting the clock tick frequency to 400 kHz (with bandwidth to spare). Additionally, Eyrx allows changing the clock tick frequency "on-the-fly" at any time with no loss of timekeeping accuracy. This allows a system designer to select an appropriate clock frequency that is custom tailored to the current task at hand. For embedded applications, slow frequencies conserve power and reduce system overhead while high frequencies provide high resolution task time scheduling.

Eyrx was designed from the ground up to truly take advantage of today's high speed PCs. In addition to its advanced timing subsystem, it provides other features that should appeal to the dedicated systems market: A small, modular, scaleable footprint, virtually unlimited tasks and priorities, sharable dynamic link libraries, a rich API with ANSI C library (with many POSIX functions), standard PC device drivers, TCP/IP networking, familiar and inexpensive development tools, professional support and much more

---

In 1981, Jeff received his Bachelors degree in Civil and Environmental Engineering from Utah State University. He worked as an engineer for 11 years in the solid rocket motor aerospace industry designing, coding and maintaining real-time control and data acquisition software for robotic testing and inspection systems. More recently, he spent 4 years working for Eyring Corporation as the design team leader for the new Eyrx real time operating system.