Abstract

PINNIX, JUSTIN EVERETT. Operating System Kernel for All Real Time Systems. (Under the direction of Robert J. Fornaro and Vicki E. Jones.)

This document describes the requirements, design, and implementation of OSKAR, a hard real time operating system for Intel Pentium compatible personal computers. OSKAR provides rate monotonic scheduling, fixed and dynamic priority scheduling, semaphores, message passing, priority ceiling protocols, TCP/IP networking, and global time synchronization using the Global Positioning System (GPS). It is intended to provide researchers a test bed for real time projects that is inexpensive, simple to understand, and easy to extend.

The design of the system is described with special emphasis on design tradeoffs made to improve real time requirements compliance. The implementation is covered in detail at the source code level. Experiments to qualify functionality and obtain performance profiles are included and the results explained.

OPERATING SYSTEM KERNEL FOR ALL REAL TIME SYSTEMS

by

JUSTIN EVERETT PINNIX

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER SCIENCE
Raleigh
2001
APPROVED BY:

[illegible]

[illegible]
Co-chair of Advisory Committee

[illegible]
Co-chair of Advisory Committee

Biography

Justin Pinnix is currently employed as a Senior Software Architect for Alerts, Inc. in Raleigh, NC. Previously, he was employed as a Software Engineer for HAHT Commerce, Inc. He received a B.S. in Computer Science from North Carolina State University in 1996.

Acknowledgements

I would like to thank a number of people who helped me out with this project. Thanks to Dr. Robert Fornaro and Dr. Vicki Jones for co-chairing my committee and providing me lots of assistance throughout the process. Thanks to Dr. Munindar Singh for serving on my committee and contributing ideas. Thanks to Dr. Felix Wu for the loan of critical equipment necessary to complete the project.

Thanks to Marhn Fullmer, the King of Procurement, for making sure I had everything I needed to complete the project. Thanks to Barbara Collins for writing much of the OSKAR user’s guide and helping me support OSKAR. Thanks to the students of CSC714 Spring 2000 and 2001 for helping me find problems and make improvements to OSKAR. Thanks also to the Operating Systems Laboratory staff, especially Chris Gorski, for the hardware and systems support.

I’d also like to thank my employer, Alerts.com for allowing me time off to work on various aspects of my degree program.

Finally, I’d like to thank my family for their support while working on this project. Special thanks to my wife Heather, who had to put up with my absence for many meetings and lots of late nights in front of the computer, while I worked on this project.

Contents

List Of Tables

List Of Figures

1 Introduction

This paper outlines the needs for a low cost, hard real time research platform. We examine several possible existing alternatives and present OSKAR, a simple but capable real time operating system.

1.1 Requirements for a Research Platform

A research platform is the hardware and software environment in which a researcher's experimental software or hardware will run. In effect, the research platform is the "laboratory" in which computer science experiments are conducted.

1.1.1 Low Cost

Research institutions typically have limited financial resources. So, to have widespread usefulness, a research platform must be inexpensive. One major area of cost is the licensing fees for the operating system itself.

An equally important consideration is the cost of the hardware. Systems requiring obscure or unusual hardware can be costly to obtain. An ideal research platform would be old personal computers that have become obsolete for desktop tasks and have been replaced. Such machines can be obtained for little or no monetary expenditure.

Off the shelf hardware is cheaper to maintain as well, for two reasons. First, there is a large supply of compatible replacement parts. Since personal computers are modular in design, failed components from older machines can usually be replaced with current production components. Second, most hardware support personnel are familiar with the platform. Institutions do not have to hire high priced experts to repair these systems.

Even if the research institution can afford specialized hardware and an expensive OS, such financial requirements will inhibit the propagation of the work to other individuals or institutions.

1.1.2 Simplicity

A research platform should be simple. Researchers writing programs for it should not have to contend with huge APIs filled with features they do not need. Rather, the system should be simple and "to the point".

1.1.3 Open Source

Inevitably, systems need to be expanded. Often times, research projects require the integration of special hardware that the operating system does not directly support.

Open source systems allow the researcher, or in most cases a trusty assistant, to extend the system to meet their needs. Not only can more hardware be supported, but defects in the original implementation can be corrected without the participation of the original authors.

1.1.4 Code Legibility

If one is to take an open source system and extend it to meet one's needs, the system must be easily understood. Therefore, an open source research OS should be simple and coded cleanly. If this is not the case, a programmer trying to extend the system will quickly get lost and may not be able to perform the task.

1.2 Real Time Systems

A real time system is defined as a system which guarantees deadlines for the execution of tasks. If a task does not meet its deadline, the system is considered in error.

Real time systems are further divided into hard real time and soft real time systems. In a hard real time system, missing a deadline is considered a catastrophic system failure. Often, this is the case with control systems where missing a deadline can cause equipment damage, or in worst cases injury or loss of life. In a soft real time system, failure to meet a deadline is considered undesirable, but does not carry the severity it does in a hard real time system.

For a system to be considered hard real time, timing constraints must be placed on the entire system. The system does not necessarily have to be fast, but it must be predictable. If a priority scheduler is used, rate monotonic scheduling may be used to mathematically guarantee execution of periodic code within its deadlines. Such a scheduler must always execute the highest priority ready process. Often, designers of general-purpose operating systems will add mechanisms, such as process aging, to prevent higher priority processes from “starving” lower priority processes. Rate monotonic scheduling requires such amenities not be implemented, as they may cause a lower priority process to prevent a higher priority process from meeting its deadline [LIU73].

The theory of rate monotonic scheduling holds only for independent processes. If two processes interact using a semaphore, a priority inversion can occur when a low priority process uses a semaphore that is also used by a higher priority process. If the lower priority process obtains ownership of the semaphore and the higher priority process preempts it, the higher priority process will be forced to block until the lower priority process releases the semaphore. In the mean time, a medium priority process may be allowed to run, causing the higher priority process to not complete on time.

Priority inheritance protocols are a way of eliminating this problem. There are two such protocols, the Basic Inheritance Protocol (BIP), and the Priority Ceiling Protocol (PCP). In the BIP, the lower priority job will have its priority temporarily boosted to that of the highest priority job that is waiting on the semaphore owned by the lower priority job. When the lower priority job releases the semaphore, its priority will be returned to normal.

However, the BIP does not prevent deadlocks. If a higher and lower priority job lock semaphores A and B in a different order, the two can become deadlocked. Also, if a process accesses semaphores used by a number of other processes, a chain of blocking can occur. A higher priority process may have to wait for multiple lower priority processes to release locks on semaphores, increasing the amount of time the higher priority process is blocked. The PCP prevents both deadlocks and chains of blocking. It establishes a priority ceiling for each semaphore, which is the priority of the highest priority process that will access the semaphore. By using this ceiling value, the system will deny requests to lock semaphores that may cause a chain of blocking or a deadlock [SHA90].

Most general-purpose operating systems are optimized for fast average case performance. Stochastic techniques, such as virtual memory and caching, help average case execution times at the expense of poor worst-case execution times. Real time systems are generally deterministic, yielding consistent best and worst case execution times.

1.3 Real Time Research Platforms

A real time research platform is a system that meets the requirements for both a research platform and a hard real time system. It must be low cost, simple, open source, and have a deterministic kernel, a preemptive priority scheduler, and support priority inheritance protocols.

1.4 Existing Solutions

1.4.1 Commercial RTOS

There are a number of commercially available real time operating systems. These operating systems meet the requirements for a hard real time system. They are usually optimized for embedded work. Some popular commercial real time operating systems are pSOSystem, VxWorks, and QNX. All three of these systems have very similar feature sets.

All three systems feature a microkernel design. The kernel provides fast, deterministic context switching. The I/O drivers and most advanced system services are implemented as processes, rather than inside of the kernel. This allows them to be more modular and to follow the same preemption rules as user code.

These systems provide priority based scheduling. The scheduler can be configured into first in first out (FIFO) mode. In this mode, a process runs until a higher priority process preempts it, or it blocks. This type of scheduler allows rate monotonic scheduling. The presence of this type of scheduler, coupled with deterministic context switch times allows these systems to be declared hard real time.

These systems also feature an array of synchronization features. All three feature binary semaphores that use the BIP. pSOSystem also features the PCP for both semaphores and condition variables. Each of these systems also implements message passing and shared memory.

VxWorks, pSOSystem, and QNX offer complete TCP/IP implementations. The IP, UDP, TCP, ICMP, IGMP, and ARP protocols are supported. Each system supports Ethernet and PPP physical layers. The stacks are optimized to minimize the number of times a packet is copied while the packet is being queued for transmission or received.

Each system also supports a number of commonly used file systems. The systems can use MS-DOS, UNIX, and ISO 9660 file systems. NFS is also supported for accessing remote file systems. QNX and VxWorks support specialized file systems for reading and writing files from solid state flash memory.

POSIX compatibility is also an issue in commercial real time operating systems. POSIX 1000.4 describes a standard API for real time operating systems. Applications written to use the POSIX APIs are much easier to port to other POSIX compatible systems [GAL94].

pSOSystem and VxWorks have their own proprietary APIs and offer a POSIX compatibility layer to translate POSIX API calls into the native API. The QNX native API is itself POSIX compliant.

The major drawback to these systems is that they are quite expensive to license. Being commercial products they are also not open source. This limits the extensibility of the system and forces the researcher to rely on the provider for all support and defect repair of the product.

1.4.2 Open Source UNIX derivatives

In recent years, several open source operating systems, based heavily on UNIX have achieved great popularity. The most famous of these is Linux. Another popular system is FreeBSD.

Open source UNIX derivatives offer the researcher a number of advantages over commercial real time operating systems. Foremost, they require no license fees to be paid. Of equal importance is the fact that they run on Intel based personal computers which are ubiquitous.

Being open source, these systems are extensible. Additional features can be added and bugs can be fixed without the participation of the original authors.

Unfortunately, these systems are unsuitable for real time work, because none of them are true real time systems. To implement a real time operating system, real time requirements must be considered in almost every line of kernel code. Code paths within the kernel must be short and run in bounded time. None of the popular open source UNIX operating systems make this guarantee.

A secondary concern is the complexity of the operating system itself. The kernel for Linux 2.2.14 contains over 1.5 million lines of C code. This provides a large array of features for the end user, but can make modifying the operating system more difficult. Being open source, the code was written by hundreds of different people, and coding conventions vary from module to module.

1.4.3 RTLinux

RTLinux [YOD97] is an add-on package for Linux that makes it a hard real time system. This is an interesting problem, since Linux itself is not hard real time. The kernel is large and monolithic. Kernel switch times are not deterministic. The scheduler is not a preemptive priority scheduler; it implements process aging and round robin scheduling, which is typical of conventional operating system designs.

Rather than trying to modify the Linux kernel to meet hard real time requirements, RTLinux implements its own kernel. This kernel has a priority preemptive scheduler and deterministic context switch time. It meets the requirements for a hard real time system. This kernel has total control of the CPU and hardware. The original Linux kernel runs on top of the RTLinux kernel, as the bottom priority process.

This design allows any real time process to preempt the Linux process at any time. So, no matter how slow or unpredictable Linux may be, we can build a hard real time system around it, because the RTLinux processes are always given higher priority and can preempt Linux at any time.

Real time processes are considerably different than Linux processes. Rather than the traditional UNIX APIs, real time processes are managed using POSIX thread functions. For example, to create a process under Linux, one would use the `fork()` and `exec()` system calls. However, to create a real time process in RTLinux, one must use the `pthread_create()` call.

This system's greatest strength is its ability to fully leverage Linux. Linux has drivers for a huge array of hardware devices, a world class networking subsystem, a file system, remote access, a huge runtime library, and many other facilities. An array of resources of that size is simply not available with real time and embedded operating systems, including OSKAR.

A major drawback is that access to Linux-controlled resources (such as the console, network, disk drives, etc.) must be coordinated through a Linux process. Real time processes cannot directly access these devices. Since Linux controls these devices and Linux is not a hard real time system, no timing guarantees can be made with regards to accessing these devices.

RTLinux provides inter-process communication mechanisms (shared memory and a pseudo I/O device) that allow real time processes to communicate with Linux processes. However, the application programmer must coordinate work between the real time and Linux portions of the system.

RTLinux does have drivers that allow RTLinux processes to access certain pieces of hardware directly. RTLinux has drivers for the standard serial port, the Tulip PCI Ethernet driver, and a few data acquisition boards.

1.5 OSKAR

OSKAR (Operating System Kernel for All Real time) is a small real time operating system that fulfills all of the requirements for a hard real time research platform. It is a simple hard real-time system that runs on a single, Intel Pentium compatible CPU.

1.5.1 Commodity Hardware

OSKAR runs on Intel based PCs. This is the same family of PCs that run the popular Microsoft Windows™ operating systems and the venerable MS-DOS™. As an additional cost savings, OSKAR can be run on systems that are too old to run current versions of Windows.

OSKAR requires a Pentium™-compatible microprocessor. It requires at least four megabytes of RAM and an NE2000 compatible network interface card (NIC). OSKAR does not require a hard drive.

In contrast, Windows 2000 requires a Pentium 133 or later and 64 megabytes of RAM [MS00]. The difference in system requirements between Windows and OSKAR means that for the foreseeable future, there will be a huge supply of machines that are not able to keep up with Windows, but can run OSKAR easily. Such hardware can be obtained cheaply, or in many cases for free.

OSKAR does require a host system running Linux to host the editor, compiler, and remote booting. However, this system need only meet the minimum requirements for Linux, which are still considerably less than that of Windows. OSKAR was developed using a host machine, which contained a 486, 8MB of RAM, and ran Linux. Therefore, there is a large supply of suitable host machines as well.

1.5.2 Open Source

OSKAR is an open source system. All source code is freely available to the public to compile, study, or even modify to fit one's own needs. The kernel is written in C with a few modules in assembler.

In addition to OSKAR being open source, it is also built on open source tools. The source code can be compiled with the GNU C compiler (gcc) and GNU assembler (as). These tools are widely used in the open source community. This means that they are robust, well supported, and very familiar to most operating systems programmers. All of the tools necessary to build, deploy, and extend OSKAR come standard as part of Red Hat Linux 6.2.

1.5.3 Small Code Base

The OSKAR system has approximately 5000 lines of C code. It was written by a single author using a consistent coding standard. The code is well documented, both internally in comments, and in the included external documentation. It is quite easy for a programmer to extend the operating system and add more features.

1.6 OSKAR Features

1.6.1 Hard Real Time

OSKAR's most important feature is the fact that it meets the requirements for a hard real time operating system. All of the time critical kernel code paths are short and deterministic. The system has been analyzed and guarantees can be made about its execution time.

OSKAR implements a preemptive priority scheduler, capable of handling both fixed and dynamic priorities. This allows real time systems to be developed using rate monotonic scheduling. In addition, priority inheritance protocols are implemented to allow systems with dependent processes to meet real time requirements. Both the basic inheritance and priority ceiling protocols are implemented for both fixed and dynamic priority scheduling.

1.6.2 IP Networking

OSKAR supports a subset of the TCP/IP suite of protocols. TCP/IP is the dominant protocol suite for communication between computer systems. It is the foundation for the modern Internet. This allows OSKAR to interact with more full featured, general-purpose operating systems using an Ethernet connection.

OSKAR allows user programs to communicate to other systems using user datagram protocol (UDP). This protocol is a member of the TCP/IP family of protocols and is understood by virtually all modern operating systems, including Linux, FreeBSD, Solaris, and Windows.

An OSKAR program can use UDP to send data back to a data-logging program running on a host computer that runs a more conventional operating system for easier analysis. OSKAR programs can use UDP to communicate and coordinate activities with other OSKAR systems as well. If connected to a router, an OSKAR program could send data to and receive data from a remote host attached via the Internet.

OSKAR supports network booting using DHCP (dynamic host configuration protocol) and TFTP (trivial file transfer protocol). The network boot loader may be burned into an EPROM, allowing a completely diskless system.

1.6.3 Software Clock

OSKAR provides a software clock with mi crosecond resolution. It combines inputs from several hardware clocks with different characteristics to make a single virtual clock. This clock is used for all OSKAR timing operations. The clock can be synchronized to a globally distributed time source.

Many real time systems require the need for a globally distributed time source. Telecommunications systems often use synchronized clocks to perform time division multiplexing. Synchronizing clocks can be a difficult problem, especially if the two clocks are in different geographical areas.

The Global Positioning System (GPS) is one of the most accurate ways to receive time signals. Each GPS satellite has on board atomic clocks synchronized to the USNO master clock. A GPS receiver uses the relative phase of a number of these time signals to determine its position. Once it knows its position, it can adjust the time signals for propagation delay. This allows a GPS receiver to provide a very accurate time signal.

OSKAR directly supports the connection of an external GPS receiver to a PC running OSKAR. Not only can the current time be received from the GPS receiver, but also the GPS measurements are used to discipline the computer's internal clock to the correct frequency and phase. Therefore, even if the GPS is disconnected, OSKAR applications have a free running time base that is still more accurate than one which has never been disciplined. When the GPS signal is reconnected, the clock will once again be disciplined to the GPS signal. Any phase and frequency drifts that were encountered during the undisciplined period will be gradually corrected.

GPS synchronization enhances a system's hard real time properties. It does so by improving the system's ability to measure time. A hard real time system can only make timing guarantees within the error bounds of its time base. For example, an undisciplined system may guarantee a certain periodic process will run once per millisecond. However, if this system measures a real millisecond as only 0.8 milliseconds, then the periodic process will be running with a period greater than one real millisecond. Therefore, the system is in error.

1.6.4 Basic I/O Drivers

OSKAR provides I/O drivers for the system keyboard, video display, serial ports, and parallel ports. The keyboard driver allows OSKAR programs to read ASCII characters from the keyboard as they are typed by the user.

The video driver allows ASCII text to be displayed to the computer's monitor. A printf() function is included to allow easy text and numeric formatting, based on a C programming language standard. The video driver also includes a text windowing library. This allows user programs to divide up the available 80x25 character screen space into a number of independently scrolling windows. A common use for this feature is to allow the user to see messages from two running processes in two different areas of the screen, without their messages becoming interleaved. Text windowing is also useful for creating control panel type applications for monitoring multiple variables of a system.

OSKAR provides a serial port driver capable of blocking and buffering. Buffering allows an OSKAR process to read and write data from the driver in units greater than one byte. This cuts down on the number of context switches and improves performance of serial I/O. If an OSKAR process attempts to read the serial port and no characters are available, the process will become blocked, allowing lower priority ready processes to run. When the system receives a character, the receiving process is unblocked and the lower priority processes are preempted. Blocking also occurs if the transmit buffer is full.

The serial driver supports up to four serial ports. The ports are independent; they do not share any buffers or blocking data. Non-standard I/O addresses and interrupt request levels (IRQs) can be used, as OSKAR is not limited by the COM1/COM2 standard adopted by early PCs.

OSKAR's parallel port driver allows the system to communicate with a printer. The PC parallel port is really a 16 bit TTL I/O port. It has 4 input lines, 4 output lines, and 8 bi-directional lines. This makes is useful for connecting external hardware that OSKAR can monitor or send signals to. The OSKAR parallel port driver has calls for accessing hardware attached in this manner. In addition, there is one input that can generate an interrupt. OSKAR provides a system call to block a process until this interrupt is triggered.

1.6.5 Timing

The OSKAR kernel provides a high-resolution clock based on an aggregate of several hardware clocks and may optionally synchronize to an external time source. The OSKAR clock's resolution is one microsecond.

1.6.6 Inter Process Communication

OSKAR provides several features for inter process communication. Shared memory is provided. Binary semaphores are implemented to serialize access to shared memory and other shared resources. Also, a message passing library is implemented to allow serial communication and work queuing between processes.

2 Design of OSKAR

2.1 System Components

The operating system code is divided into two categories: kernel and non-kernel. Kernel code is not interruptible and not reentrant. In effect, it runs at maximum priority, above all non-kernel code. Primarily, the kernel consists of the scheduler, which determines what process should be running, the context switching code, which makes the chosen process run after the kernel exits, and the interrupt handlers, which service requests from hardware devices. Also, the code to initialize the system is contained in the kernel as well.

All non-kernel code must run in the context of an OSKAR process. These processes may be preempted by a higher priority process, or the kernel itself.

OSKAR applications make use of system services by calling into OSKAR APIs (application programming interfaces). Some APIs are implemented in the kernel, while others are implemented in non-kernel code. For example, the semaphore wait operation (`sem_wait()`) is implemented in kernel code. When a user program calls this API, it generates a software interrupt. This software interrupt causes a context switch into kernel code. A handler function inside of the kernel processes the semaphore request. During the course of its operation, it may change the state of the calling process and cause another process to be scheduled.

Other APIs, such as the TCP/IP stack are implemented completely in non-kernel code. A call to the `udp_send()` API simply calls into the networking library. A handler function in the networking library encapsulates the data into a properly formed network packet and places it into the network driver's send queue. At any point during the execution of this API (except for a small critical section in the message passing code), the process may be preempted by a higher priority process.

Some portions of OSKAR are complete processes themselves. They are implemented in non-kernel code. Containing these parts of the system in their own processes allows them to run independently of user-supplied code, but still follow the same scheduling rules as all other processes. Most of these system-supplied processes are device drivers.

2.2 Phases of Operation

The OSKAR system goes through three different phases. The first phase is the boot phase. At this phase, the system is executing boot loader code to load the OSKAR image (including the user program) into memory.

The second phase is the initialization phase. During the initialization phase, all of the OSKAR subsystems are set up. All interrupts and I/O are disabled. Interrupt handlers are put in place. Memory is allocated. Hardware devices are configured. The user's `init()` function is called to allow the user program to change the configuration of the system before it begins running. The `init()` function will cause at least one process to be created.

Once initialization is complete, the system enters the running phase. During this phase, OSKAR processes are executed. Interrupts and I/O are enabled. The kernel regulates access to system resources and schedules the appropriate processes to run according to the scheduling rules. The system stays in the running phase until the computer is turned off, rebooted, a `halt()` instruction is issued, or a fatal exception occurs.

2.3 Kernel Architecture

The OSKAR kernel is designed to be a simple "textbook" kernel. It coordinates execution of a number of processes. All processes share a single processor. The kernel performs context switches to change which process is currently executing.

All processes execute in the same address space. All processes have full access to every other process' memory. This allows for faster context switching, easier inter-process communication, and a simpler kernel. It does require the programmer be more careful, as the operating system cannot prevent memory overwrites from other processes.

Processes may only be created at boot time. Each process has a fixed priority, specified at creation time. If the system is configured for dynamic priority scheduling, this fixed value is ignored. The maximum number of processes defaults to 10. A user program may increase this limit to any reasonable value at boot time, but it must do so before any processes are created.

Access to the CPU is governed by a scheduler. OSKAR's scheduler allows for both fixed and dynamic priority scheduling. The choice of which scheduling algorithm to use can be specified by the user's program at boot time.

2.4 Memory Allocation

OSKAR implements a trivial memory allocation scheme. Blocks of memory are allocated using the `allocate()` call. This call simply increments a pointer. This allows memory to be allocated in constant time. However, memory allocated with this call may never be freed.

The `allocate()` call is used to allocate objects at initialization time (processes, semaphores, buffers, etc.). Such objects are not allocated after the initialization phase and have no need to be freed. User programs can also use the `allocate()` call for memory requests with the same constraints.

A general-purpose dynamic memory manager, capable of freeing and reusing memory, is not supplied, because such memory managers are generally not able to run in constant time. Constant time operations are easily bounded. Any algorithm used in a hard real time system must be bounded, otherwise it is impossible to guarantee its completion time.

Applications that require the ability to free and reuse memory must implement their own dynamic memory schemes. Often, it is possible to create a dynamic memory scheme that does run in constant or bounded time within the context of a specific application. If it is know that an application only requests certain sized blocks, or only requests a limited number of them, it is easy to create a specialized algorithm, or bound a generic one. The message-passing library (see Section 2.6) is an example of a specialized memory manager that runs in bounded time.

2.5 Semaphores

The kernel provides basic counting semaphores. These semaphores provide **wait** and **signal** operations. In addition, semaphores interact with the scheduler to provide priority ceiling protocols. Both the BIP and PCP protocols are supported. Like processes, semaphores must be created at boot time. The maximum number of semaphores defaults to 10, but like the number of processes, it may be increased before any semaphores are created.

2.6 Message Passing

OSKAR provides a message-passing library. The message-passing library allows a process to asynchronously send a block of data to another process. Since the message passing operations are asynchronous, the sending process need not wait for the receiver to handle the message before continuing execution.

The message-passing library implements several fundamental objects. A message is simply a pre-allocated block of memory of a set size. Each message has a “length” field to record the number of bytes actually used in the message. Messages are created by creating a message pool. The number and size of messages must be specified at creation time. Once the pool has been created, messages can be allocated and have appropriate data copied into them.

Messages are passed between processes using message queues. A queue is simply a collection of semaphores and a linked list of waiting messages. The semaphores are necessary to ensure serial access to the linked list, allowing the list to be safely accessed by multiple processes. A sending process sends a message by enqueuing it onto a queue. The receiving process dequeues it. There are both blocking and non blocking options for the dequeue operation. The non-blocking version returns immediately if there are no messages in the queue. The blocking version blocks the receiving process until a message is enqueued.

Once a message is no longer used, it can be freed, allowing it to be used again. Each message pool maintains queue of free messages. The message allocate operation is actually a dequeue of the pool’s free queue. Likewise, the free operation is an enqueue to the pool’s free queue.

Message pools are a limited form of dynamic memory. Since all memory objects are of equal size, both the `allocate` and `free` operations can run in constant time. This allows their runtime to be predictable, which is a requirement for a hard real time system.

2.7 I/O Support

In a “textbook” conventional OS, all device I/O is done within kernel space. Since the I/O routines run in kernel space, they need be neither reentrant nor interruptible. Confining all

I/O access to kernel space guarantees serial access to the hardware itself and allows the OS to prevent user code from having direct hardware access.

This type of I/O creates a priority inversion that is problematic for real time systems. Kernel code effectively executes at top priority, above the highest priority process. A low priority process can request a lengthy I/O operation that will prevent higher priority user processes from running. Even worse, an external device can cause an I/O handler to execute, creating a priority inversion that was not even triggered by user code. In such a system, rate monotonic scheduling is not possible.

These types of priority inversions are unacceptable even in conventional operating systems. These operating systems have ways of attempting to overcome this problem. Linux queues lengthy requests into a task queue to be handled by a piece of code called the “bottom half”. [RUB98] The bottom half is basically a second scheduler. It schedules pieces of work that are done on behalf of the interrupt handler, but run with interrupts enabled. Such a system gives acceptable performance for a conventional OS, but is very difficult to model for rate monotonic scheduling, since there are two schedulers instead of one.

OSKAR adapts a microkernel-like approach [SIL94]. I/O drivers still have interrupt handlers, but they are kept extremely simple and execute quickly. A driver process performs the bulk of the work. In most cases, the interrupt handler simply acknowledges the interrupt and unblocks the driver process.

The driver process is an OSKAR process. The process is automatically created by the driver when the driver’s initialize routine is called by the user’s application. It is usually necessary for the user’s application to specify a priority for the driver process as an argument to the initialize routine. This priority is used as a fixed priority to schedule the driver process when the system is configured for fixed priority scheduling. This means it is entirely possible (and many times desirable) for user code to preempt the device driver. This can happen whenever a user process has a higher priority than the priority assigned to the device driver.

OSKAR processes have full access to the hardware, so the driver process can do all of its communication with the device. This design allows I/O access to be prioritized and managed using the scheduler. The driver process delivers data to and from user processes using one of OSKAR's inter-process communication mechanisms.

2.8 Timing

OSKAR's time sources are the Pentium cycle counter, the 8254 timer chip, and the CMOS real time clock. The cycle counter provides a high-resolution, undisciplined time source. The 8254 is a lower resolution time source, but it is capable of producing interrupts. The CMOS clock is non-volatile, so it keeps running even when the machine is powered off. The OSKAR clock combines all of these time sources to make a virtual clock that has high resolution, is disciplined, can generate interrupts, and is non-volatile.

External synchronization is done via a pulse per second (PPS) signal from external hardware. OSKAR slews the frequency of its internal clock so that the frequency and phase differences between the PPS signal and OSKAR's internal clock approach zero.

2.9 Networking

OSKAR provides two distinct areas of network functionality. The first is network booting. This is accomplished using a third party open source tool called etherboot. Etherboot is a bootstrap loader that is loaded from a boot floppy or a ROM. It makes a DHCP request to obtain an IP address for itself. Then, it contacts a TFTP server and downloads the OSKAR system image into memory and executes it.

The rest of the network functionality is implemented in OSKAR's networking subsystem. This consists of an Ethernet driver and a protocol stack. The Ethernet driver currently supports only NE2000 compatible network cards. These are far from being the fastest cards ever manufactured, but they are widely available and inexpensive ($14 each). OSKAR's driver and driver interface are adapted from the etherboot source, so it is not difficult to add support for any card etherboot supports.

OSKAR's protocol stack supports a subset of the TCP/IP protocol suite. User programs may send and receive UDP datagrams. This allows an OSKAR system to communicate with another OSKAR system, or a conventional platform such as Windows or UNIX. The

Address Resolution Protocol (ARP) protocol is implemented. OSKAR also supports sending and receiving broadcast packets.

Traditional operating systems usually implement the IP stack inside of the kernel. This type of implementation is problematic in a real time system. The problem is best demonstrated using an example. Suppose that a real time system has two processes. Process A operates a dangerous pneumatic drill, while process B answers IP requests for current weather conditions. Using rate monotonic scheduling, we determine that process A needs to have a higher priority than B, since it needs to check the position of the drill bit 20 times a second, while the response time of weather requests need not be guaranteed. Since the receipt and transmission of IP packets is implemented in the kernel, which preempts all user processes, it is possible for those network operations to preempt process A, possibly causing it to miss a deadline.

OSKAR's IP stack is implemented completely in non-kernel code. Most of the work of preparing IP packets to be transmitted and unpacking received packets runs in the context of the calling process. The physical transmission and receipt of packets is handled in the Ethernet adapter's driver process. Shuttling of the packets to and from the Ethernet driver's process is handled using the message passing library.

Since the IP stack is implemented in non-kernel code, all network operations are subject to preemption by the scheduler. As long as process A had a higher priority than the Ethernet driver process (whose priority is set by the user's application), network requests would never preempt process A. Instead, those requests would be handled by process B and the Ethernet driver process when the scheduler allowed them to execute.

It is important to note that the IP protocol itself was not designed to meet hard real time requirements. It does not make any guarantees about the timeliness of message receipt or delivery.

One barrier to real time IP is the Ethernet medium itself. If two hosts attempt to transmit at the same time, each host will wait a random period of time and retry [STE94]. A random timeout is certainly not predictable. The timeout itself can be bounded, but there is no way to guarantee the retry will not produce another collision. So, we must settle for a “soft” real time requirement. A lightly loaded Ethernet segment will have few collisions.

Another real time culprit is the ARP. This is a protocol used to translate IP addresses into Ethernet addresses. When a packet is transmitted, the following steps are taken [STE94]:

1. Look up the IP address in an in-memory table (the ARP cache). If found, use the Ethernet address in the table.

2. If not found, broadcast an ARP request packet to all hosts on the local Ethernet segment. (Who has IP address aaa.bbb.ccc.ddd?)

3. Wait for an ARP reply packet from the owner of the IP address. If received, add the ARP entry to the cache and use its Ethernet address (I have IP address aaa.bbb.ccc.ddd my Ethernet address is xx:xx:xx:xx:xx:xx).

4. If no reply is received within a given period of time, drop the transmitted packet.

This algorithm has several real-time unfriendly aspects. First of all, any time a cache is involved, the algorithm becomes stochastic rather than deterministic. The first time a packet is sent to a host, it will take a long time. Each successive send will be quick, as long as the cache entry is kept current. In order to bound it, you must find the worst case time. But, this can not be computed either, because a foreign host (possibly running a non real-time operating system) must produce the reply. Also, there is the possibility of collisions on both the ARP request and ARP reply. So, it is difficult to come up with an upper bound less than the ARP timeout value.

OSKAR has a solution to this problem. The IP layer can be configured to not use ARP. In this mode, if the IP address is not found in the ARP table, the packet will be sent to the Ethernet broadcast address. Also, the user can force static entries into the ARP table, which will never expire. This makes the ARP lookup stage deterministic and fast.

2.10 What Makes OSKAR Measurable

Much of OSKAR's performance profile can be determined by executing test routines in user space against the OSKAR kernel. Such a measurement is performed in Section 4.3.1 to determine the context switch time. Some performance measurements are made by the system and can be read using a system call. An example of this type of measurement is performed in Section 4.3.2 to determine the interrupt latency.

Most operations can be timed using OSKAR's internal clock. OSKAR's clock provides microsecond accuracy. Since the OSKAR clock can be calibrated to a GPS, performance measurements made using this clock are in effect NIST traceable.

3 Implementation

3.1 Execution Environment

OSKAR executes on Intel based personal computers. It requires a Pentium compatible processor. The processor runs in Protected mode. It uses the Intel “flat” memory model. All physical memory is mapped into one 4GB address space, which is shared by all processes. In this regard, processes are more like threads, whereas conventional operating system processes have their own address spaces. The processor’s segment registers are locked to fixed values and not switched as part of a context switch.

For performance and simplicity, most of the processor’s security mechanisms are disabled. Any process may access any other process’ memory, or the system memory. This eliminates the need to do expensive context switches to access data common to the system and application processes.

The floating-point features of the processor are supported. A context switch between two processes will save the floating-point registers of the old process and load the values last saved with the new process.

Context switch functionality is spread over a number of functions. All context switches initiate with an interrupt. Interrupts are generated either by hardware when it needs the OS to perform an action, or by user code when it needs to make a system call. The processor handles the interrupt by pushing the EIP and EFLAGS registers to the stack, disabling further interrupts, and calling an interrupt handler. All OSKAR interrupt handlers share almost identical entry and exit code. This is assembly code is defined in `system.S`. The interrupt handler will push the remaining registers onto the stack in a set order using the `PUSHA` instruction. Then, the stack pointer is stored in the current running process’ PCB. Next, the stack register is set to point to a system stack. Since all interrupts are handled serially, one common system stack will suffice. Next, the entry code makes a call into a C function that does the “real work” of handling the interrupt. Most C interrupt handlers make a call to the `schedule()` function. This function determines which process should be running and sets the global variable `pid` to that process’ ID.

When the C interrupt handler returns, the exit code reads the stack pointer value from the current process' PCB. The current process is determined by reading the value of `pid`, which may have been changed by the handler. Once the stack pointer is reset, a POPA instruction is executed. This retrieves the correct register values for the process that should be running from that process' stack. Lastly, the IRET instruction is executed, which restores the EIP and EFLAGS registers. This causes the processor to resume execution at the point where the new current process was last interrupted. The processor's interrupt mask is stored in the EFLAGS register, so the IRET call will also re-enable interrupts.

3.2 System Calls

As described in Section 2.1, some OSKAR APIs are implemented in kernel code. To call these functions from a user process requires a special mechanism to switch the system into kernel mode.

The kernel call is implemented as a C function in the OSKAR kernel source. The function definition is preceded by a `K_CALL()` macro, which takes the name of the system call as a parameter. Next, the function is declared. The function implementing the system call must begin with the letter k, followed by an underscore, followed by the name of the system call. Prepending `k_` to the name is necessary because user code must not call this function directly.

When OSKAR is compiled, the script `gen_kcalls.pl` searches all source files and finds all instances of the `K_CALL()` macro. In turn, this script generates a stub function, whose name is the same as the system call name. The stub function switches the system to kernel mode and then calls the implementation (`k_`) function. After the implementation function returns, the stub function switches the system back to user mode.

Figure 3-1 is an example of the `sleep_until` system call. User code will call into the `sleep_until()` function, which is generated as a result of the `K_CALL(sleep_until)` macro. After switching the system to kernel mode, `sleep_until()` calls the implementation function `k_sleep_until`().

```
K_CALL(sleep_until)
void k_sleep_until(timestamp_t wakeup)
{
    pcb[pid].wakeup = wakeup;
    pcb[pid].state = state_sleep;
    schedule();
}
```

Figure 3-1 - Definition of a System Call

3.3 Scheduler

The scheduler code is listed in Figure 3-2. Every interrupt handler makes a call to the scheduler, normally near the end of the interrupt handler's execution. First, it records the elapsed time since the last time it was called and adds this value into the PCB's `cpu_time` field. Next, it iterates through all available processes. Each running or ready process is considered. If the system is configured for fixed priority scheduling, the process with the lowest value of `eff_pri` in its PCB is selected. If dynamic scheduling is used, the process with the soonest timestamp value for the `deadline` field in the PCB is selected. In the event of a tie, the process with the highest PID is chosen. This corresponds to the process created last.

All processes are visited in this loop. It never terminates before visiting all processes. This helps give it a more constant run time and makes it easier to predict.

Once the new running process is selected, the global variable `pid` is set to the id of the selected process. If the new process is not the same as the old process, the floating point registers are swapped into the PCB.

```
void schedule()
{
    static timestamp_t last_call = 0;
    timestamp_t now = get_time_exact();

    int new_pid = n_processes - 1;  // Which pid to activate next
    // Default to "idle" process
    int i;
    static int n = 0;
    n++;

    // Record statistics
    if (pid != pid_none)
    {
      pcb[pid].cpu_time += now - last_call;
    }
    last_call = now;

    // Choose a new process
    for (i = 0; i < n_processes - 1; i++)
    {
      switch (pcb[i].state)
      {
      case state_running:
      case state_ready:
          if (scheduler_type == scheduler_fixed)
          {
            if (pcb[i].eff_pri < pcb[new_pid].eff_pri)
            {
                new_pid = i;
            }
          }
          else
          {
            if (pcb[i].deadline < pcb[new_pid].deadline)
            {
                new_pid = i;
            }
          }
          break;
      default:
          /* Ignore */
      }
    }
```

Figure 3-2 – Scheduler Code

```
    // Switch new pid into context
    if (pid != new_pid)
    {
      assert(new_pid >= 0 && pid < n_processes);

      if (pid != pid_none)
      {
          // Switch out old process
          pcb[pid].sp = user_sp;
          if (pcb[pid].state == state_running)
            pcb[pid].state = state_ready;

          save_fpregs(&pcb[pid].fpregs);
      }

      // Switch in new process
      pid = new_pid;

      user_sp = pcb[pid].sp;
      pcb[pid].state = state_running;
      restore_fpregs(&pcb[pid].fpregs);
    }
    last_call = now;
}
```

Figure 3-2 - Scheduler Code (cont'd)

3.4 Clock

OSKAR keeps time using a number of time sources. The primary source of time is the Pentium processor's cycle counter. This is a 64-bit counter that accumulates the number of clock cycles since the system was booted. It is extremely useful for a number of reasons. First of all, it is very precise. On a 100 Mhz computer, the cycle counter will have a 10 ns resolution. Second, it never overflows. On a 100Mhz computer, it would take 5,845 years for the clock to overflow. Third, it can be read with a single CPU instruction. Hardware timers often require the processor to wait a number of cycles before the result can be read.

The cycle counter alone cannot be used as the time source. Its value cannot be set. At minimum, a timestamp of the system's boot time needs to be added to it for it to return "real" time. Its frequency cannot be directly adjusted. Its frequency varies from system to system, as different model PCs have different clock speeds. It cannot generate interrupts, which means it cannot be used for preemptive multitasking.

All PCs contain another hardware timer called the 8254 [IBM85]. This timer runs at a much lower frequency, roughly 1Mhz. This frequency is the same for all PCs (within a tolerance). This timer has a decrementing counter attached, which can be configured to generate an interrupt when it reaches 0. The OS can set its initial value, and it can be configured to automatically re-trigger itself when it reaches 0. So, this chip divides the frequency of the 1Mhz oscillator and generates processor interrupts at that frequency.

Neither of these timers are synchronized to an external time source. Both use extremely cheap oscillators, whose frequency tolerances can cause drift of several seconds per day. One way to solve this problem would be to connect an atomic clock to the PC and use it as an additional time source. However, this is an extremely expensive proposition.

GPS receivers, however, do provide a globally distributed time source based on a collection of atomic clocks. The output of a GPS receiver is within 50µs of UTC. OSKAR can use a GPS receiver as an additional time source, providing OSKAR applications with a clock of similar tolerances.

OSKAR combines these three time sources to make a software virtual clock that has the benefits of all three. OSKAR's software clock uses the convenient base unit of microseconds. Absolute times are reported as the number of microseconds since the epoch. OSKAR uses the UNIX epoch of January 1, 1970.

The 8254 interrupt frequency can be set at boot time by calling the system call `set_pd_clk()`.

OSKAR's clock works by periodically updating a global variable called `cur_time`. This update is performed as a result of the 8254 timer generating an interrupt. This interrupt is handled by the `timer_intr()` function in `timer.c` (see Figure 3-3). `cur_time` contains the number of microseconds since the epoch. Each time an 8254 interrupt occurs, a number of microseconds are added to `cur_time`. However, the number of microseconds is not constant, rather it is a value computed to take clock adjustments into account.

```
void timer_intr(int intno, struct regs *regs)
{
    timestamp_t cycle;
    int i;

    /* Record interrupt latency */
    latency_buffer[latency_buffer_pos] = pd_clk + get_cal_time() - 65536;
    latency_buffer_pos = (latency_buffer_pos + 1) % LATENCY_BUFFER_SIZE;

    /* Read the cycle counter */
    cycle = get_cycle_count();

    /* Wiggle interrupt controller bits */
    outb(0x20, 0x20);

    /* Update cur_time */
    remainder += nanos_elapsed(cycle - last_cycle);

    cur_time += remainder / 1000;
    remainder = remainder % 1000;

    /* Allow "new" phase_adjust value to take effect */
    last_freq = eff_freq;
    last_cycle = cycle;

    // Resume any processes waiting on their
    // deadlines
    for (i = 0; i < n_processes; i++)
    {
        if (pcb[i].state == state_wait_period && pcb[i].deadline <
cur_time)
        {
            pcb[i].deadline = pcb[i].next_deadline;
            pcb[i].state = state_ready;
        }
        else if (pcb[i].state == state_sleep && pcb[i].wakeup < cur_time)
        {
            pcb[i].state = state_ready;
        }
    }

    schedule();
}
```

Figure 3-3 - Clock Update Routine

Each time a timer interrupt is received, the Pentium cycle counter is read into the variable `cycle`. The value of the cycle counter from the previous run is stored in the variable `last_cycle`. `last_cycle` is subtracted from `cycle` to come up with the number of cycles elapsed since the last timer interrupt. This value is multiplied by one billion and divided by `last_freq`, to compute the number of nanoseconds elapsed since the last interrupt.

This value cannot be directly added to `cur_time`, because `cur_time` is specified in microseconds. The global variable `remainder` is used to extend the precision of `cur_time` to the nanosecond level. This helps prevent the accumulation of round off errors. The number of nanoseconds elapsed is added to `remainder`. `remainder` is divided by 1000. The quotient is added to `cur_time`, and the variable `remainder` becomes the remainder of that division.

At system boot time, the frequency of the cycle counter is measured against the 8254 clock. This value is stored in the variable `freq`, which is never modified. It is also used as the initial value for `last_freq`. However, if OSKAR is synchronizing to an external source, `last_freq` will be adjusted to make phase and frequency corrections in the clock.

The system call `get_time()` is used to report the current time (in µs past the epoch) to user programs. This call returns the value of `cur_time`. While cheap to execute, this version's precision is limited by the frequency of the 8254 timer. For more demanding applications, `get_time_exact()` must be called. It returns a more accurate version by reading the cycle counter and performing the same calculations as the interrupt handler to compute a new value of `cur_time`. It is important to note that `get_time_exact()` does not actually modify any of the variables, rather it returns what `cur_time`'s value would be if an interrupt occurred at that instant.

A potential problem emerges here. `last_freq` is being constantly changed. What if a process called `get_time_exact()` two times consecutively, returning t1 and t2? What if between the first and second calls, the system made a frequency adjustment, slowing down the clock frequency? Since the second call is using a lower divisor than the first call, t2 could have a lower (earlier) value than t1. This would make the clock non-monotonic, as time would appear to be flowing backwards.

This does not happen in OSKAR, because the clock adjustment code is not allowed to directly modify `last_freq`. Instead, it must modify the variable `eff_freq`. At the end of each timer interrupt, the value of `eff_freq` is copied to `last_freq`.

OSKAR's external clock synchronization algorithm is based loosely on the NTP (network time protocol) algorithm described by Mills [MIL94]. Like the NTP algorithm, it uses a modified phase locked loop (PLL) to adjust the phase and frequency of the internal clock to match the external source. Unlike NTP, the OSKAR algorithm only deals with pulse per second (PPS) signals. NTP is optimized to work accurately with sporadic updates whose periods range from 1 second to 2^{16} seconds. NTP also uses integer math for all calculations, while OSKAR uses floating point. This was done to allow the algorithm to be easier to program and more accurate. Since the synchronization code only runs once per second, a few floating point operations is an acceptable overhead.

The GPS receiver interfaces with the OSKAR clock in two places. Once a second two events occur. First, the GPS receiver sends a pulse through its PPS output, which should be connected to the computer's parallel port. Second, it outputs the current UTC time (as of the last pulse) to the serial port, which should be connected to a serial port on the computer.

When the system is booted, the system time must be stepped to a value that is approximately UTC, by calling the `set_time()` function. The function `gps_get_time()` will return the UTC time last transmitted by the GPS. Passing the value from `gps_get_time()` into `set_time()` will cause the clock to be set to a value that is at most one second behind UTC.

An OSKAR process must monitor the PPS signal using the `par_wait()` system call. When the GPS signals the computer's parallel port, the interrupt handler will immediately call `get_time_exact()` and store its value. This is the observed time of the PPS event. Next, it will cause the `par_wait()` call to unblock. The monitor process calls the `pps()` function (see Figure 3-4). The `pps()` function requires two parameters, the observed time of the PPS event, according to the computer, and the real time the PPS event occurred, according to the GPS.

```
void pps(timestamp_t observed_time, timestamp_t real_time)
{
    int64 new_freq = 0;

    // Compute periods of real and observed timestamps
    observed_period = observed_time - last_observed_time;
    real_period = real_time - last_real_time;

    // Make sure pulse falls within acceptable boundaries
    if (observed_period > 800000 && observed_period < 1200000 &&
      real_period > 800000 && real_period < 1200000)
    {
      last_pps_status = PPS_GOOD;

      // Compute offset and adjustment values
      offset = observed_time - real_time;
      phase_adj = ((double) offset) / ((double) real_period);
      phase_adj /= pps_params.phase_divisor;
      freq_adj =
          (((int) observed_period) -
           ((int) real_period)) / ((double) real_period);
      freq_adj /= pps_params.freq_divisor;

      // Compute new frequency using adjustments
      new_freq = eff_freq;
      new_freq += (freq_adj * (double) eff_freq);
      new_freq += (phase_adj * (double) eff_freq);

      // Apply clamping and set eff_freq
      if (new_freq < min_freq)
          eff_freq = min_freq;
      else if (new_freq > max_freq)
          eff_freq = max_freq;
      else
          eff_freq = new_freq;
    }
    else
    {
      // Last pulse was bad.  Ignore it
      last_pps_status = PPS_BAD;
    }

    // Remember timestamps for next iteration
    last_observed_time = observed_time;
    last_real_time = real_time;
}
```

Figure 3-4 - PPS Update Routine

The `pps()` function makes a number of calculations. First, it computes a real and observed period, by subtracting the current and previous values of real and observed time respectively. Next, “bad” pulses are thrown out. If the GPS has missed a pulse, or it is its first pulse, `real_period` will not equal 1 second. If the observed period is less than 800ms or greater than 1200ms, the pulse is thrown out as well. This type of problem can result from line noise or a faulty GPS receiver.

Next, the offset is computed. The offset is the difference between the observed and real times of the PPS event.

Two measurements of the "correctness" of the observed signal are made: a phase measurement (a_f) and a frequency measurement (a_f). These measurements are computed as follows:

$$a_f = \frac{t'_o - t'_r}{(t'_r - t_r)} \qquad a_f = \frac{(t'_o - t_o) - (t'_r - t_r)}{(t'_r - t_r)}$$

To demonstrate the necessity of two measurements, let us look at each measurement independently. If only the frequency differences of the two signals were used to adjust the frequency of the OSKAR clock, then we would have a frequency locked loop (FLL). This would guarantee the two clocks were ticking at the same rate, but their relative times would continue to be off. Thus, the clocks would never converge.

If only the phase difference were used (as was done in an earlier attempt at this algorithm), a subtle problem emerges. Consider the case where both clocks are running at exactly the same frequency, but the OSKAR's clock is one second ahead of the GPS. OSKAR would detect a phase difference of 1,000,000 ìs and begin slowing the clock down. Closing the gap between the two takes a number of seconds. With each PPS pulse, the phase difference is measured again. In each case, the OSKAR clock is still ahead, so OSKAR will continue to reduce the frequency of its internal clock. Finally, the two clocks will converge and their phase difference will be zero. However, at this instant, their frequencies are considerably different. One second later, OSKAR will be behind the GPS clock. It would begin speeding up and the phases would converge, only to overshoot again. With the right damping factors, this system would eventually converge, but it will take much too long. As the phase offset approaches zero, OSKAR needs to begin undoing the frequency adjustments it made to bring the two together.

OSKAR's solution to this problem is to use a linear combination of both measurements. Each measurement has a damping factor associated with it. At each iteration, frequency is adjusted using the following formula:

$$f' = f\left(1 + \frac{a_f}{d_f} + \frac{a_f}{d_f}\right)$$

d_f and d_f are damping factors for phase and frequency respectively. Since they are divisors, smaller values mean more of that component is used to "steer" the clock frequency. d_f should always be smaller than d_f. This allows OSKAR to begin correcting the frequency as the offset decreases. A low $\frac{d_f}{d_f}$ ratio allows more phase correction than a higher ratio. More phase correction means the clocks will converge quicker, but are more likely to overshoot a few times before settling down. A higher ratio makes the clocks converge slower, but will eliminate any overshoot. If the ratio is too low, the algorithm will overshoot every time and will never converge. If the ratio is too high, all phase adjustments will be overridden by frequency adjustments and the clocks will never converge. Since no one choice is right for every application, OSKAR users can change the damping factors using the `pps_set_params()` call. Default values are used if this call is not made.

In addition to the damping factors, frequency movement is further limited by the `max_slew` parameter. `max_slew` specifies a percentage beyond which the frequency may not be adjusted. If `max_slew` is 0.10 (the OSKAR default), then f' will not be allowed to fall below 90% or above 110% of the clock frequency measured at boot time. This prevents the PPS code from adjusting the frequency to something awfully wrong. In addition to wreaking havoc on running applications, allowing `eff_freq` to creep too far out of range would break OSKAR's noise rejection code, which requires both real and observed periods to be within 20% of 1 second.

The value of f' is limited by the `min_freq` and `max_freq` variables. These are computed by the `pps_set_params()` function by using the `max_slew` parameter and the clock frequency measured at boot time.

The values used in the above equations correspond to variables in the OSKAR code. The translation is provided in Table 3-1.

Table 3-1 - Variable Name Translations

Expression	Variable Name
t_o'	`observed_time`
t_o	`last_observed_time`
t_r'	`real_time`
t_r	`last_real_time`
f'	`new_freq`
f	`eff_freq`
a_f	`phase_adj`
a_f	`freq_adj`
d_f	`pps_params.phase_divisor`
d_f	`pps_params.freq_divisor`

The PPS code forms a phase locked loop. The bad pulse rejection is the low pass filter. The offset calculation is the phase detector. The feedback gain is computed by the damping factors, and the VCO is the value of freq.

It is important to note that variable delays between the return of `par_wait()` and the invocation of `pps()` are not a problem. The parallel port interrupt handler actually does the observation of time delays. So, `pps()` may be called anytime within the next second. Thus, the process calling `pps()` should have a low priority, since its frequency is 1 Hz.

Also, the PPS routines may be used with any time source that produces a PPS pulse, not just GPS receivers.

3.5 Semaphores

Semaphores are implemented as counting semaphores in the classical sense, except for the addition of priority inheritance protocols. The semaphore code supports the BIP, PCP, or can use no priority inheritance at all. The user sets the inheritance protocol by calling the `set_pip_type()` function at startup.

The `sem_create()` function creates a new semaphore. Each semaphore is represented by the structure `semaphore`, which has the following fields: `owner`, `waiter`, `prev_pri`, `count`, and `pri_ceil`. `owner` is the PID of the process that currently owns this semaphore. If the no process currently owns the semaphore, it will contain the special value `PID_NONE`.

The `owner` field must be only read or set from within a kernel call handler. Since kernel call handlers are interrupt handlers, no interrupts will be handled during their execution. This allows us to guarantee serial access to the semaphore's owner field.

The field `waiter` represents the head pointer to a linked list of processes waiting on the semaphore. Like owner, it also contains a process ID. If no processes are waiting on the semaphore, it will be set to `PID_NONE`. Processes are chained together using the `next_sem` field of the PCB. This scheme allows constant time access to the waiting process list without allocating additional memory. `count` stores the value of the semaphore. If the value is greater than or equal 0, a wait operation will exit without locking the semaphore.

```
K_CALL(sem_wait)
void k_sem_wait(sem_t s)
{
    if (pip == pip_ceiling)
      k_sem_wait_pcp(s);
    else
    {
      sem[s].count--;

      if (sem[s].count < 0)
      {
          pcb[pid].next_sem = sem[s].waiter;
          sem[s].waiter = pid;
          pcb[pid].state = state_wait_sem;

          if (pip == pip_basic)
          {
            if (pcb[sem[s].owner].eff_pri > pcb[pid].eff_pri)
            {
                // Owner of semaphore inheirits this job's priority
                pcb[sem[s].owner].eff_pri = pcb[pid].eff_pri;
            }
          }
          sem[s].owner = pid;
          sem[s].prev_pri = pcb[pid].eff_pri;
          schedule();
      }
    }
}
```

Figure 3-5 - Semaphore Wait Routine

`sem_wait()` is implemented in the kernel as the function `k_sem_wait()` (see Figure 3-5). This function first checks the value of the `owner` field. If no other process has the semaphore locked, `owner` will be `pid_none`. In this case, owner is set to the current process' PID and `k_sem_wait()` returns immediately.

If the semaphore has been locked by another process, `owner` will contain that process' PID. In this case, `k_sem_wait()` inserts the current process to the head of the `waiter` linked list. The process' state is set to `wait_sem`.

If the BIP is used, an additional check is made. If the process owning the semaphore has a lower priority than the current process, that process' `eff_pri` field (in the PCB) is set to the current process' PID. This allows the owner of the semaphore to inherit the current process' priority.

`schedule()` must be called at the end of `k_sem_wait()`, as the current process will become blocked, so a more eligible process must be selected to run.

```
K_CALL(sem_signal)
void k_sem_signal(sem_t s)
{
    if (pip == pip_ceiling)
      k_sem_signal_pcp(s);
    else
    {
      sem[s].count++;
      if (sem[s].count <= 0)
      {
          /* Restore original priority */
          if (pip == pip_basic)
            pcb[pid].eff_pri = sem[s].prev_pri;

          /* Transfer ownership of semaphore to next in line */
          sem[s].owner = sem[s].waiter;

          if (sem[s].waiter != pid_none)
          {
            pcb[sem[s].waiter].state = state_ready;

            if (pip == pip_basic)
                sem[s].prev_pri = pcb[sem[s].waiter].eff_pri;

            sem[s].waiter = pcb[sem[s].waiter].next_sem;
          }

          schedule();
      }
    }
}
```

Figure 3-6 - Semaphore Signal Code

`k_sem_signal()` releases a process' lock on a semaphore. If the BIP is used, the owner process' original priority is restored. Ownership of the semaphore is transferred to the next waiting process at the head of the `waiter` linked list. The new process' state is switched to ready and if the BIP is used, the process' current priority is stored. If no other processes are waiting on this semaphore, `waiter` will have the value `pid_none`. `schedule()` is called to ensure the highest priority ready process gets to run.

3.6 Networking

3.6.1 Initialization

Users must first call `eth_init()` to initialize the Ethernet interface. `eth_init()` expects the IO address of the Ethernet card, its IRQ, and priority for the driver process. Currently, OSKAR creates two driver processes. The transmitter process is given the caller-specified priority and the receiver process is given one priority level lower. `eth_init()` also creates a message pool and a transmit queue. These are created using OSKAR's message passing subsystem.

Next, the user may modify any of the TCP/IP subsystem parameters. To do this, the user should call `tcpip_get_params()` to obtain a copy of the parameter block, change the desired parameters, and call `tcpip_set_params()` to copy the parameter block back to the system.

Next, the user calls `tcpip_init()`. This initializes the TCP/IP subsystem with the parameters set above. If `TCPIP_USE_DHCP` is specified (it is by default), `tcpip_init()` calls `read_bootp()`. `read_bootp()` finds the DHCP request packet that was received by etherboot when the system was booted. It uses values stored in the packet to set the IP address.

Next, the ARP cache is initialized by `arp_init()`. `arp_init()` creates a process to clean expired entries out of the ARP cache. This process' priority may be set using the `arp_cleanup_pri` member of the `tcpip_params` structure. Also, the size of the ARP cache may be set using the `arp_cache_size` member.

3.6.2 Transmission

User processes call `udp_send()` to transmit a packet. First, it allocates a message from the Ethernet message pool. This packet is big enough to hold all of the header information needed for the various layers of the networking subsystem. The user-supplied data buffer is copied into this message. The data is not copied again until it actually reaches the Ethernet driver. A timeout of 0 is passed into the `msg_alloc()` function. This means that if there are no free message packets, the function will return null. In this case, `udp_send()` simply exits. Dropping packets is considered an acceptable way to handle errors in TCP/IP.

Next, `udp_send()` fills in the IP and UDP headers. The IP checksum is computed using the `ipchksum()` function. Then, the packet is passed to the `arp_send_ip()` function.

`arp_send_ip()` is responsible for determining the Ethernet destination address of the packet and causing the packet to be transmitted once it has done so. First, it sets the source address of the packet. Next, it locks a mutual exclusion semaphore for the ARP table. Then, it looks up the destination IP address in the ARP table by calling `arp_lookup()`. If a valid entry is found, the Ethernet address from the table is copied to the message buffer and the message is queued for transmission.

Otherwise, `arp_request()` is called. `arp_request()` creates a new entry in the ARP table for the address. It allocates another message packet. This new packet is filled in with the necessary structure to form an ARP request packet requesting the Ethernet address corresponding to the requested IP address. `arp_request()` queues this packet for transmission and sets the new ARP entry's status to `ARP_WAITING`.

Control is returned to `arp_send_ip()` where the original message (the one passed into `arp_send_ip()`) is added to a linked list of pending messages in the ARP cache. This list of pending messages is a "corral" where outgoing IP packets wait for their replies to come back. At the end of the function, `arp_send_ip()` releases the ARP cache semaphore.

It is important to note that all of the code described above runs in the context of the user's process. Once the packet is ready to be transmitted, the message packet is added to the Ethernet driver's transmit queue. This transmit queue is an OSKAR message queue and a caller uses the normal `msg_queue_add()` function to enqueue the message. This message queue is the mechanism that relays the message packet to the Ethernet transmit process (`eth_tx_proc()`). `eth_tx_proc()` runs in a loop executing `msg_queue_remove()` calls. The next available message is dequeued and passed into the Ethernet driver. The Ethernet driver copies the message's contents to the NIC, causing the packet to be transmitted. `eth_tx_proc()` blocks until the NIC signals the transmission is complete by generating an interrupt. Once transmitted, `eth_tx_proc()` frees the message packet and repeats the process for the next incoming packet.

3.6.3 Receiving

User processes must call `udp_socket()` to create a socket for receiving UDP packets. A UDP socket is simply a wrapper that contains a message queue. Next, the user must call `udp_listen()` to connect the socket to a port. The TCP/IP layer keeps a 65,536 entry table called `udp_listeners`, that maps port numbers to sockets. `udp_listen()` simply places the socket handle into this table at the entry corresponding to the subscribed port. `udp_listen()` may be called multiple times to bind one socket to multiple ports.

To receive packets, user processes call `udp_recv()`. `udp_recv()` performs a blocking `msg_queue_remove()` operation on the socket's message queue.

The actual receipt of a packet occurs in the Ethernet receiver process. `eth_rx_proc()` blocks, waiting for the NIC to signal that a packet has been received. An interrupt handler in the kernel unblocks the process when this happens. The `poll` method of the Ethernet driver is called to read the packet into `nic.packet`. Next, it calls `dispatch_packet()` to perform the appropriate action for the packet.

`dispatch_packet()` examines the Ethernet packet type. If it is not an ARP or IP packet, it is ignored. If it is an ARP or IP packet, a message packet is allocated and the contents are copied into it. If the received packet is an ARP packet, `dispatch_packet()` calls `arp_process_request()`.

If the packet is an IP packet, the protocol field of the IP header is examined. If it is a UDP packet, the destination port number is looked up in `udp_listeners`. If a socket is listening to this port, the message packet is queued onto that socket's message queue. If not, the packet is discarded.

Again, the message queue transfers the message packet from the Ethernet driver processes to the user process. `udp_recv()` dequeues the message. Information such as the sender's IP and port numbers are copied from the message buffer to the user supplied pointers. Then, the message payload is copied into the user supplied buffer. Last, the message packet is discarded and control is returned to the user code.

If `dispatch_packet()` receives an ARP packet, it calls `arp_process_request()`. This function examines the ARP packet. If it is a request from another host requesting our Ethernet address, the message packet is converted to a reply and the appropriate information filled in. This reply packet is requeued onto the transmit queue for transmission by the Ethernet driver.

Otherwise, the ARP semaphore is locked. The IP address referenced in the ARP packet is looked up in the ARP cache. If an entry is found, it is updated with the information supplied in the packet. The ARP entry's expiration time is extended to 20 minutes from now. If any messages are in the entry's waiting list, they are enqueued on the transmit queue. The list is cleared. Lastly, the semaphore is released and the message packet freed.

3.7 I/O Drivers

3.7.1 Video Driver

The video driver operates by directly accessing the video buffer. The video buffer is an area of memory, located at address B8000 that represents the information currently displayed on the monitor. Each screen location has a corresponding 16 bit value in the video buffer. The low byte is the ASCII code of the character that is displayed at that location. The high byte specifies the background and foreground color of that character.

The core of the video driver is the `print_str_new()` function (see Figure 3-7). This function takes as its arguments a string to print out, and a window structure defining the boundaries within which it may be printed. It simply walks through each character in the string and updates the screen buffer. The `screen_addr()` function translates the cursor x and y positions in the window structure to an address offset in the video buffer. The high byte of the screen location is the hex value 7, which indicates the formatting for the character cell should be white text on a black background. After printing each character, the x and y positions are updated to valid positions within the window.

```
void print_str_new(struct window *w, char *str)
{
    for (; *str; str++)
    {
        if (*str == '\n')
        {
            w->x = w->right - w->left;
        }
        else if (*str == '\b')
        {
            w->x -= 2;
        }
        else
            vidbuf[screen_addr(w)] = *str | 0x700;

        /* Advance cursor */
        w->x++;
        if (w->x < 0)
        {
            w->y--;
            w->x = w->right - w->left;
        }
        else if (w->x > w->right - w->left)
        {
            w->x = 0;
            w->y++;
        }

        if (w->y > w->bottom - w->top)
        {
            scroll_window(w);
            w->y = w->bottom - w->top;
        }
    }

    show_cursor(w);
}
```

Figure 3-7 - Video Driver String Printing Function

If the last line in the window is written, `scroll_window()` is called to move the contents of the window up one line to allow a new line to be written in the bottom position. See Figure 3-8 for the code that accomplishes this.

```
void scroll_window(struct window *w)
{
    int y;
    int linelen = (w->right - w->left + 1) * 2;

    /* Scroll window up 1 line */
    for (y = w->top; y <= w->bottom; y++)
    {
        memcpy(&vidbuf[y * 80 + w->left],
               &vidbuf[(y + 1) * 80 + w->left], linelen);
    }

    /* Clear bottom line */
    memset(&vidbuf[w->bottom * 80 + w->left], 0, linelen);
}
```

Figure 3-8 - Scrolling Function

3.7.2 Keyboard driver

The keyboard driver has two entry points: the keyboard interrupt handler, and the `getch()` system call. The interrupt handler `kbd_intr()` is called whenever the user presses a key. The keyboard controller generates a hardware interrupt to signal the presence of a keystroke. `kbd_intr()` reads a status byte from the controller. If it was a valid keystroke, it reads the scan code from the controller. The scan code indicates which key was pressed. The function `handle_scancode()` looks up the scan code in a table. Each entry specifies a handler to be called when the scan code is received. Each key can use one of six handlers. `ignore_handler()` does nothing, causing the event to be ignored. `ctrl_handler()` handles the press or release of a control key such as control, alt, or shift. It updates global variables reflecting the state of these control keys. `lock_handler()` updates the state of the caps lock key. `del_handler()` handles processing of the delete key. Its only function is to reboot the system if control and alt are depressed. `break_handler()` is called when the keyboard controller sends a prefix byte indicating the next scan code represents a key that has been released. It sets a flag noting that the next scan code is a release.

Normal key presses are handled by the `ascii_handler()` function (see Figure 3-9). This function converts the scan code into an ASCII character using values specified in the lookup table. Different values are returned depending on the state of the shift, control, and caps lock keys. The ASCII value is enqueued into a circular receive buffer using the `enqueue_key()` function (see Figure 3-10).

enqueue_key() checks to see if a process is blocked waiting on a character. If so, the process is unblocked and the character returned directly to it. If no processes are waiting, the character is enqueued into the receive buffer.

```
struct key_type
{
    void (*handler) (int);
    char normal;
    char shifted;
    char ctrl;
};

/* Key table is declared in keytbl.h */

static void ascii_handler(int code)
{
    char c = 0;
    if (isbreakcode(code))
      return;

    if (ctrlstate[ck_ctrl])
      c = keytbl[code].ctrl;
    else if (ctrlstate[ck_shift])
      c = keytbl[code].shifted;
    else
      c = keytbl[code].normal;

    if (c)
      enqueue_key(c);
}

static void ignore_handler(int code)
{
}

static void ctrl_handler(int code)
{
    ctrlstate[(int) (keytbl[code & 0xff].normal)] = !(code & 0xf00000);
}

static void lock_handler(int code)
{
    if (isbreakcode(code))
      return;
    ctrlstate[(int) (keytbl[code].normal)] ^= 1;
}

static void break_handler(int code)
{
    breakcode = 1;
}
```

Figure 3-9 - Keyboard Table and Handlers

```
void handle_scancode(int code)
{
    if (breakcode)
    {
      code |= 0xf00000;
      breakcode = 0;
    }

    if (keytbl[code & 0xff].handler == NULL)
    {
      printf("NULL handler code %x\n", code);
      halt();
    }
    keytbl[code & 0xff].handler(code);
}
```

Figure 3-9 - Keyboard Table and Handlers (cont'd)

```
void enqueue_key(char key)
{
    /* Is someone waiting on this? */
    if (q.owner != pid_none)
    {
      pcb[q.owner].state = state_ready;
      kcall_return_int32(q.owner, key);
      q.owner = pid_none;
    }
    else
    {
      q.buf[q.tail] = key;
      q.tail = (q.tail + 1) % KBD_BUFSIZE;
    }
}
```

Figure 3-10 - enqueue_key() Function

User processes call `getch()` (see Figure 3-11) to retrieve the next available character from the keyboard. `getch()` first checks the receive buffer. If a character is available, it is dequeued from the buffer and returned to the caller. Otherwise, the process is blocked.

```
K_CALL(getch)
void k_getch(int *bp)
{
    /* Is receive buffer empty? */
    if (q.head == q.tail)
    {
      /* Wait on it */
      pcb[pid].state = state_wait_io;
      q.owner = pid;
      schedule();
    }
    else
    {
      /* Return value immediately */
      kcall_return_int32(pid, (int) q.buf[q.head]);
      q.head = (q.head + 1) % KBD_BUFSIZE;
    }
}
```

Figure 3-11 - getch() System Call

3.7.3 Serial Driver

The serial driver is implemented as an OSKAR process. This process reads characters from the serial port and enqueues them into a FIFO character buffer, called the receive buffer. It also dequeues characters from another FIFO character buffer, called the transmit buffer, and writes them to the serial port. User programs call the `ser_read()` and `ser_write()` APIs to read and write characters from the transmit and receive buffers. These APIs do the appropriate blocking if characters are not available.

User programs call `ser_init()` to open the serial port from their `init()` routine. `ser_init()` configures the port with the correct communication parameters (baud rate, stop bits, etc.) and creates a driver process.

The driver process is implemented in the function `ser_proc()` (see Figure 3-12). `ser_proc()` is an infinite loop. First, it calls `ser_wait()`, a kernel call which blocks the calling process until a serial interrupt is received, or another process calls the `ser_release()` system call. Next, it reads all available characters from the serial port hardware. These characters are enqueued into the transmit buffer using the `buf_put()` function. Once all available characters have been read, the driver checks if the port is ready to receive characters to transmit. If so, characters are dequeued from the transmit buffer and written into the serial port. Afterward, the `sem_rx` and `sem_tx` semaphores are signaled to release any processes that may be blocked waiting on characters to be read or written to the serial port.

```
static void ser_proc(void *param)
{
    int port = (int) param;
    char c;
    int wrote;
    int read;
    int lsr;

    for (;;)
    {
      /* Lock the serial port mutex */
      sem_wait(serial[port].sem_mutex);

      /* Read any waiting characters and queue them into the rx buffer */
      read = 0;
      while (1)
      {
          lsr = inb_p(serial[port].io + LSR);

          if ((lsr & RBF) == 0)
            break;

          c = inb_p(serial[port].io + RBR);
          buf_put(&serial[port].rx_buf, c);
          read++;
      }

      /* Write any pending characters from the tx buffer */
      wrote = 0;
      while ((!buf_empty(&serial[port].tx_buf) &&
             (inb_p(serial[port].io + LSR) & THRE) != 0))
      {
          outb_p(buf_get(&serial[port].tx_buf), serial[port].io);
          wrote++;
      }

      /* Release the mutex lock */
      sem_signal(serial[port].sem_mutex);

      /* If we read anything, release any process waiting on the
         sem_rx semaphore */
      if (read > 0)
          sem_signal(serial[port].sem_rx);

      /* If we wrote anything, release any process waiting on the
         sem_tx semaphore */
      if (wrote > 0)
          sem_signal(serial[port].sem_tx);

      /* Wait for interrupts to occur, or for ser_write to unblock us
         using ser_release */
```

Figure 3-12 - Serial Port Driver Process

```
      /* Reenable serial port interrupts */
      inb_p(serial[port].io + IIR);
      /* TODO: What if interrupt happens here? */
      ser_wait(port);
    }
}
```

Figure 3-12 - Serial Port Driver Process (cont'd)

The `ser_read()` (see Figure 3-13) API is non-kernel code. It runs directly in the calling process' context. It attempts to read up to `len` waiting characters from the receive buffer. If the buffer is initially empty, it will wait on the `sem_rx` semaphore. This will cause the process to block until characters are put into the buffer. Next, it copies all available characters (up to `len`) into the user-supplied buffer. Last, it returns the number of characters read.

`ser_read()` locks the `sem_mutex` semaphore during its operation. This semaphore prevents user processes and the driver from simultaneously modifying the receive or transmit buffers. Before blocking itself, `ser_read()` must release `sem_mutex` so that the driver process can access the semaphore. After being unblocked, it must reacquire `sem_mutex` before continuing.

```
int ser_read(int port, char *buf, int len)
{
    int i = 0;

    sem_wait(serial[port].sem_mutex);
    while (buf_empty(&serial[port].rx_buf))
    {
      sem_signal(serial[port].sem_mutex);
      sem_wait(serial[port].sem_rx);
      sem_wait(serial[port].sem_mutex);
    }

    while (i < len && !buf_empty(&serial[port].rx_buf))
    {
      buf[i] = buf_get(&serial[port].rx_buf);
      i++;
    }

    sem_signal(serial[port].sem_mutex);
    return i;
}
```

Figure 3-13 - Serial Port Read API

Like its companion, `ser_write()` (see Figure 3-14) is a non-kernel function that runs in the user's process context and locks the `sem_mutex` semaphore. It iterates through the characters supplied in the user's buffer and writes each one into the buffer. If at any point the buffer is full, `ser_write()` will block itself until the `sem_tx` semaphore is signaled.

```
int ser_write(int port, char *buf, int len)
{
    int i = 0;
    sem_wait(serial[port].sem_mutex);
    for (i = 0; i < len; i++)
    {
      while (buf_full(&serial[port].tx_buf))
      {
          sem_signal(serial[port].sem_mutex);
          sem_wait(serial[port].sem_tx);
          sem_wait(serial[port].sem_mutex);
      }
      buf_put(&serial[port].tx_buf, buf[i]);
    }
    sem_signal(serial[port].sem_mutex);
    ser_release(port);
    return i;
}
```

Figure 3-14 - Serial Port Write API

3.7.4 Parallel Driver

The parallel port driver is the simplest driver in OSKAR. Only one of its APIs is implemented in the kernel. The others are non-kernel code that execute in the user's process. Their code is provided in Figure 3-15. `par_read()` simply looks up the parallel port's IO address and reads the two input port registers from it. `par_write()` accepts a 16-bit word and writes the upper and lower words to the two output port registers. `par_dir()` copies the `input` parameter to the parallel port's data direction register.

```
int par_read(int port)
{
    int iobase = parport[port].io;
    return inb(iobase) | (inb(iobase + 1) << 8);
}

void par_write(int port, int data)
{
    int iobase = parport[port].io;
    int cr;

    outb(iobase, data & 0xff);
    cr = inb(iobase + 2);
    cr &= 0xf;
    cr |= (data >> 8) & 0xf;
    outb(iobase + 2, cr);
}

void par_dir(int port, int input)
{
    int iobase = parport[port].io;
    int cr;

    cr = inb(iobase + 2);
    cr &= 0x20;
    cr |= (input ? 1 : 0);
    outb(iobase + 2, cr);
}
```

Figure 3-15 - Parallel Port Non-Kernel APIs

`k_par_wait()` (see Figure 3-16) is the only parallel port API implemented in the kernel. Its purpose is to block the calling process until a parallel port interrupt is generated. These interrupts are triggered by a level change on pin 10 of the parallel port. This allows almost any piece of hardware to be connected to the parallel port and be able to generate an interrupt.

It modifies the calling process' PCB to cause the process to block. The PID is written into the port structure, so that the interrupt handler knows which process to unblock. `par_wait()` calls `schedule()` to schedule another process to run.

```
K_CALL(par_wait)
void k_par_wait(int port,timestamp_t* time)
{
    /* Write my pid into the parport structure so I get woken up */
    parport[port].waiter = pid;
    parport[port].waiter_time = time;

    /* Block process and schedule next one */
    pcb[pid].state = state_wait_io;
    schedule();
}
```

Figure 3-16 - par_wait() System Call

When the interrupt is generated, `par_intr()` (see Figure 3-17) is called to handle it. Its first task is to record the time the interrupt occurs. Next, it signals the interrupt controller to allow future interrupts to be generated. Next, it uses the interrupt number to locate the corresponding port number. From this, the ID of the blocked process is read. If a process is waiting on this interrupt, its state is set to ready and the scheduler is called to select the appropriate running process. Last, the timestamp of the interrupt is copied to a user-supplied buffer, if a pointer other than NULL was supplied for this buffer.

```
static void par_intr(int intno, struct regs *regs)
{
    /* Figure out which port owns this interrupt */
    int i;

    time_last_intr = get_time_exact();

    /* Reenable interrupts */
    outb(0x20, 0x20);

    for (i = 0; i < nports; i++)
    {
      if (parport[i].irq == intno)
          break;
    }

    if (i < nports)
    {
      int pid;

      inb(parport[i].io);

      /* Release the waiting process */
      pid = parport[i].waiter;
      if (pid != pid_none && pcb[pid].state == state_wait_io)
      {
          pcb[pid].state = state_ready;
          schedule();
      }

      /* Report the timestamp of the interrupt (if requested) */
      if (parport[i].waiter_time != NULL)
            *(parport[i].waiter_time) = time_last_intr;

      parport[i].waiter = pid_none;
      parport[i].waiter_time = NULL;
    }

}
```

Figure 3-17 - Parallel Port Interrupt Handler

4 Experimental Results

4.1 Functional Qualification

OSKAR is first submitted to a battery of functional tests to make sure the scheduler and semaphore code are working properly. This includes tests that start multiple processes with differing priorities and ensure they execute in order by their priority.

These test programs are included in the OSKAR distribution in the `src/testcase` directory. The `schedXXX.c` programs test scheduling and priority. The `semXXX.c` programs test semaphore functionality, often making use of features proven in the scheduler tests.

4.2 Maximum Number of Processes

No system is useful without knowing its limits. In this test, we determine the maximum number of processes OSKAR can execute. There is no one number that can answer this question. The number of processes depends on a variety of factors. First, the deadline requirements of the code being executed must be considered. For this test, we use periodic tasks of varying periods (see Table 4-1 for all values tested). Second, the complexity of code must be considered. The code used is a simple array element increment executed in a periodic loop (see Figure 4-). Also, the speed of the CPU has a great effect on the number of processes. OSKAR cannot possibly execute more code in a given unit of time than the processor could execute in the absence of OSKAR.

```
#include "os.h"

/*************************************************************************
 * This program determines the maximum number of processes that can run
 * in an OSKAR system.  It does so by creating COUNT number of simple
 * processes, with varying priorities.  A top priority supervisor process
 * monitors the results of these processes and reports the number of
processes
 * that are doing work.  If less than COUNT processes are reported, then
 * some processes are not able to finish their work
 */

/* Number of processes to attempt to run */
int count = 2750*6;

/* Required periodicity */
timestamp_t period = 60000000;

/* Pointer to an array of counters, one per process */
int *counters;

/* Periodically increments this process' entry in the counters array
 */

void proc()
{
    int pid;
    timestamp_t when;

    pid = get_pid();
    when = get_time();

    for (;;)
    {
      counters[pid]++;
      when += period;
      sleep_until(when);
    }
}

/* Periodically scans the counters array and counts the number of
processes
 * that have been able to run at least once since the last scan.  Resets
 * the counters to detect further runs
 */
```

Figure 4-1 - Code Executed for Maximum Process Test.

```
void supervisor()
{
    int i;
    int expected = 0;
    int active;

    while (wait_period(period))
    {
      active = 0;
      for (i = 0; i < count; i++)
      {
          if (counters[i] >= expected - 1)
            active++;
      }
      printf("%d of %d processes running\n", active, count);
      expected++;
    }
}

void init()
{
    int i;

    /* Allocate enough counters and PCBs for all of these
     * processes */
    counters = (int *) allocate(sizeof(int) * count);
    memset(counters, 0, sizeof(int) * count);
    set_proc_limit(count + 5);

    /* Create the processes.  Doing these first allows us
       to use their PIDs as array indexes */
    for (i = 0; i < count; i++)
    {
      create_process(i + 5, 1024, proc);
    }

    /* Create the supervisor process at top priority */
    create_process(0, 8192, supervisor);
}
```

Figure 4-1 - Code Executed for Maximum Process Test (cont'd)

`maxproc.c` is unable to directly measure the maximum number of processes *m*. It simply attempts to execute a given number of processes *n* and reports how many of the processes met their deadlines. If this number is less than n, we know that $n > m$. The test is repeated using lower values for *n*, until the test succeeds. At this point, we know that *n* is an acceptable lower bound for *m*.

The worst case for a scheduling algorithm is to have all processes ready to run at the same time. Since the `sleep_until()` is executed at the end of the loop, this worst-case scenario happens the very first time the processes are executed. There must be enough processor bandwidth to complete all tasks before the supervisor process wakes up, otherwise the test is considered failed and a lower number is chosen.

The results of the tests are summarized in Table 4-1 and graphed in Figure 4-2. The maximum number of processes does not scale linearly with the periodicity. This is to be expected, since the scheduler is an $O(n)$ algorithm, dependent on the number of processes in the system. For *n* processes to complete, there must be at least *n* context switches. The context switch overhead is $O(n^2)$. This context switch overhead reduces the available processor bandwidth, so it makes sense that the number of processes grows in a less than linear fashion.

Table 4-1 - Maximum Number of Processes

Required Periodicity	Maximum Number
10ms	30
100ms	300
1s	800
5s	1700
10s	2750

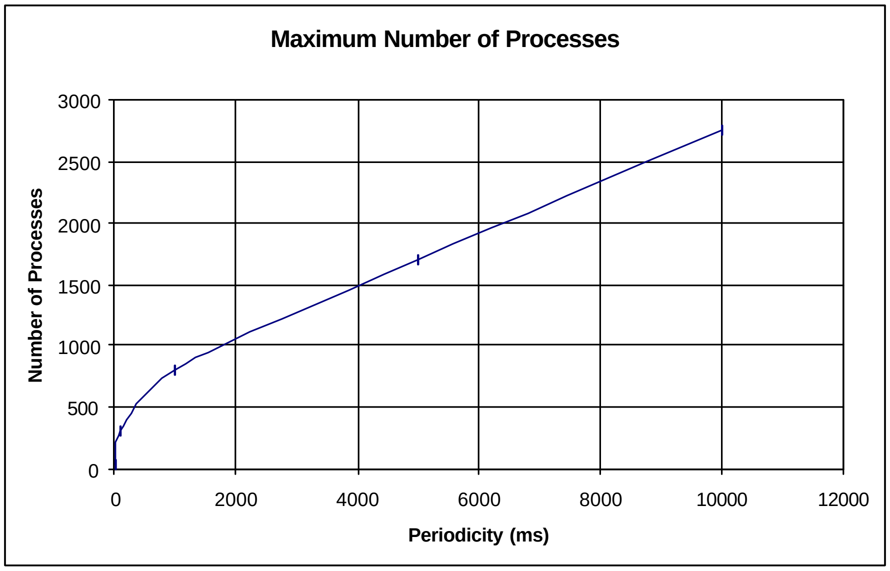

Figure 4-2 - Maximum Number of Processes

4.3 Performance Profile

As mentioned in Section 1.6.1, a hard real time OS should have bounded, predictable kernel path execution times. In this section, we measure these values experimentally. All tests in this section were conducted on an IBM PC350-100Mhz Pentium. All of the measurements in this section are extremely system dependent, so users of OSKAR should follow these procedures to make their own measurements to determine the profile for their hardware.

4.3.1 Context Switch Time

It is necessary to measure and accurately model the amount of time needed for a context switch. As discussed in 4.2, the context switch time directly affects the available processor bandwidth.

4.3.1.1 Test Conditions

The program `prof_ctx.c` was executed to measure the context switch time. This program, listed in Figure 4-3, consists of two processes. The processes run in infinite loops, periodically repeating the measurement. `proc1()` runs at top priority. First, it determines a wakeup time. This is the time that the next iteration of the test will occur. Next, it measures the current time. Then, it sleeps until the wakeup.

`proc1()`'s sleep forces `proc2()` to wake up. `proc2()` again reads the current time. It computes the difference of the two times, which is the actual context switch time. This time is printed to the console and `proc2()` sleeps until the same wakeup time as `proc1()`.

The context switch time reported by this program is slightly longer, because it also includes the execution time of two `get_time_exact()` calls. This provides a nice "safety margin" for using this number to compute processor bandwidth.

The variable `extra_procs` controls how many “extra” processes are created. These extra processes do no useful work and are of lower priority than `proc1()` and `proc2()`. However, their presence will affect the context switch time.

```
#include "os.h"

int extra_procs = 198;

timestamp_t before, after, wakeup;

/* High priority process who initiates the context switch */
void proc1()
{
    for (;;)
    {
      wakeup = get_time_exact() + 1000000;
      before = get_time_exact();
      sleep_until(wakeup);
    }
}

/* High priority process who runs after the context switch */
void proc2()
{
    for (;;)
    {
      after = get_time_exact();
      printf("Context switch took %d us\n", (int) (after - before));
      sleep_until(wakeup);
    }
}

/* Extra do-nothing process */
void proc3()
{
    for (;;)
    {
    }
}

void init()
{
    int i;

    set_proc_limit(extra_procs + 5);

    create_process(0, 8192, proc1);
    create_process(1, 8192, proc2);

    for (i = 0; i < extra_procs; i++)
      create_process(2, 8192, proc3);
}
```

Figure 4-3 - Code to Measure Context Switch

4.3.1.2 Expected Results

When `sleep_until()` is called, several things happen. Since kernel calls must run as interrupt handler, the user's code calls a generated stub function that triggers a software interrupt The system's kernel call handler receives the interrupt and calls the appropriate handler (`k_sleep_until()`). `k_sleep_until()` simply modifies the calling process' PCB and calls the scheduler. Neither the stub, the interrupt handler, nor `k_sleep_until()` contain any loops. Therefore, their runtime should be constant (excepting effects from the system cache). However, the scheduler loops through all of the process control blocks to select the next running process.

So, a mathematical formula for the context switch time would be $t = u + i + k + s_c + ns_p$ where u is the execution time of the stub, i is the time of the interrupt handler, k is the time required for k_sleep_until, s_c is the constant portion of the scheduler, s_p is the time of the portion of the scheduler executed for each process, and n is the number of processes in the system. Since the constants are not measurable or useful on their own, we combine them into a single constant k. So, the new equation is $t = k + ns_p$, which is a simple linear equation.

It is difficult to predict the values of k and s_p, since they are dependent on processor speed and instruction cycle counts. But, by taking two context switch time readings using different numbers of processes, they can be obtained using linear regression. Once values for these two constants have been derived, we should be able to use the equation $t = k + ns_p$ to predict context switch times for other values of t.

4.3.1.3 Actual Results

The first measurement was made with no extra processes. This means n=3 (`proc1()`, `proc2()`, and the "idle" process which the system creates). It returned a value of 16 µs. The second measurement was made with 9 extra processes, for a total of 12 processes. It returned a value of 25µs. Using these two data points, k=13µs and s_p=1µs.

A number of additional data points were taken and compared to predicted values. The results are summarized in Table 4-2. The predictions are within 10% of the actual values. The predicted times are longer than the actual times. Therefore, the mathematical model derived above is sufficient to produce reasonable upper bounds for the context switch time.

Table 4-2 - Context Switch Results Summary.

Number of Processes	Predicted Time (μs)	Actual Time (μs)
3	N/A	16
12	N/A	25
25	38	36
50	63	59
75	88	83
100	113	108
150	163	155
200	213	208

4.3.2 Interrupt Latency

Interrupt latency is defined as the period of time between when a piece of hardware signals an interrupt and the operating system handles it. OSKAR takes a sample of this measurement every time a timer interrupt occurs. The first step in the `timer_intr()` interrupt handler (see Figure 3-3) reads the 8254's current time. Since we know the 8254's value at the time the interrupt occurred was 0, the 8254's time will indicate the elapsed time since the interrupt, which is the interrupt latency.

The OSKAR `get_interrupt_latency()` system call analyzes the sample buffer and returns minimum, maximum, and average interrupt latencies.

4.3.2.1 Test Conditions

The program `prof_il.c` (Figure 4-4) measures interrupt latencies. It does so by calling the `get_interrupt_latency()` system call.

```
#include "os.h"

int extra_procs = 500;

/* Periodically reads the interrupt latency from the system */
void proc1()
{
    float il_max, il_min, il_avg;

    for (;;)
    {
      sleep(1000000);
      get_interrupt_latency(&il_min, &il_max, &il_avg);

      printf("min=%dns max=%dns avg=%dns\n", (int) (il_min * 1E9),
             (int) (il_max * 1E9), (int) (il_avg * 1E9));
    }
}

/* Do nothing process */
void proc3()
{
    for (;;)
    {
    }
}

void init()
{
    int i;

    calibrate_busy_wait();
    set_proc_limit(extra_procs + 5);

    create_process(0, 8192, proc1);

    for (i = 0; i < extra_procs; i++)
      create_process(2, 8192, proc3);
}
```

Figure 4-4 - Interrupt Latency Measurement Code

4.3.2.2 Expected Results

The measures interrupt latency should be lower than the context switch time, as the context switch time includes an interrupt latency. More importantly, interrupt latency should be constant. It should not vary with the number of processes in the system.

4.3.2.3 Actual Results

On the IBM PC350-100Mhz, the interrupt latency was between 1.676µs and 2.514µs. The average was 1.771 µs The program was executed with 3, 10, 25, and 50 processes. The latency did not vary.

4.4 Clock Synchronization

These tests focus on validating the functionality and performance of the OSKAR clock and PPS synchronization code. For these tests, the program `clock.c` is used. `clock.c` is provided as part of the OSKAR distribution, in the `sample` directory.

This program performs GPS synchronization and monitors its progress. At start up time, it waits for an initial time signal from the GPS. The clock is stepped to this time. The OFFSET macro can be set to force the initial time to be incorrect. This exercises the synchronization code.

Each time a PPS signal is received, `clock.c` prints out a status line containing:

1. The observed time of the last PPS event (µs since 1/1/1970)
2. The real time of the last PPS event (µs since 1/1/1970)
3. The observed period between the last two pulses (µs)
4. The offset between #1 and #2
5. The frequency adjustment factor
6. The phase adjustment factor

These lines are echoed to the serial port where they can be captured by a host machine for further processing.

4.4.1 Measuring Convergence Times

The first test is designed to measure how long it takes OSKAR to synchronize its clock to the external time source.

4.4.1.1 Test Conditions

The sample program `clock.c` was executed in OSKAR. At startup, `clock.c` also sets the phase and frequency damping factors. These factors are varied with the different test runs to empirically obtain near-optimal values.

Tests were performed on an IBM PC350-100Mhz computer. The time source was a Trimble Palisade GPS receiver attached to the serial and parallel ports of the PC350.

4.4.1.2 Expected Results

The clocks will be considered to have converged when the absolute value of the offset falls below 10 μs and stays within that range. The clocks should converge within a reasonable period of time (3 minutes).

Damping factors with lower ratios will converge quicker. Damping factors with higher ratios will converge more slowly, but not overshoot. Excessively low d_f values should converge considerably slower than more aggressive ones. Excessively high d_f values will cause excessive overshooting, which will increase the convergence time, possibly causing the clocks to never converge.

4.4.1.3 Actual Results

The program output for each run was captured from the serial port. It was read into a spreadsheet for further analysis. Two graphs were produced. The first was a plot of the offset (how far off the OSKAR clock was) vs elapsed time (in seconds). An example of such a graph is shown in Figure 4-5. This graph allows one to visualize the performance of the clock synchronization routine. The slope of the convergence code is visible and gross overshoots may be seen.

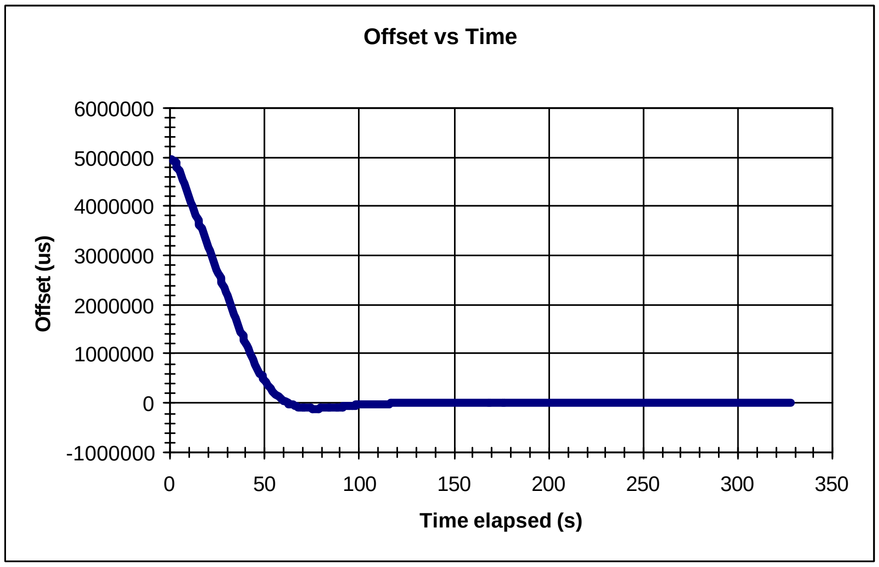

Figure 4-5 - Example of a Convergence Time Graph

However, the criterion for convergence of the two clocks is the point in time after which the offset does not exceed 10µs. To determine this point, we zoom in and plot a version of the same graph that only shows offsets between –20µs and +20µs (Figure 4-6). To determine the convergence time, we simply read the X axis value at the point where the line crosses +10 or –10µs for the last time.

The detailed graph also allows us to determine the number of overshoots. An overshoot is defined as an event where the OSKAR clock over corrects itself and changes from being behind the master clock to ahead of, or vice versa. Graphically, an overshoot is seen as a point where the line crosses the 0µs line. It is important to note the overshoots which occur after the convergence time are not counted. It is normal for the OSKAR clock to fluctuate a few µs above and below.

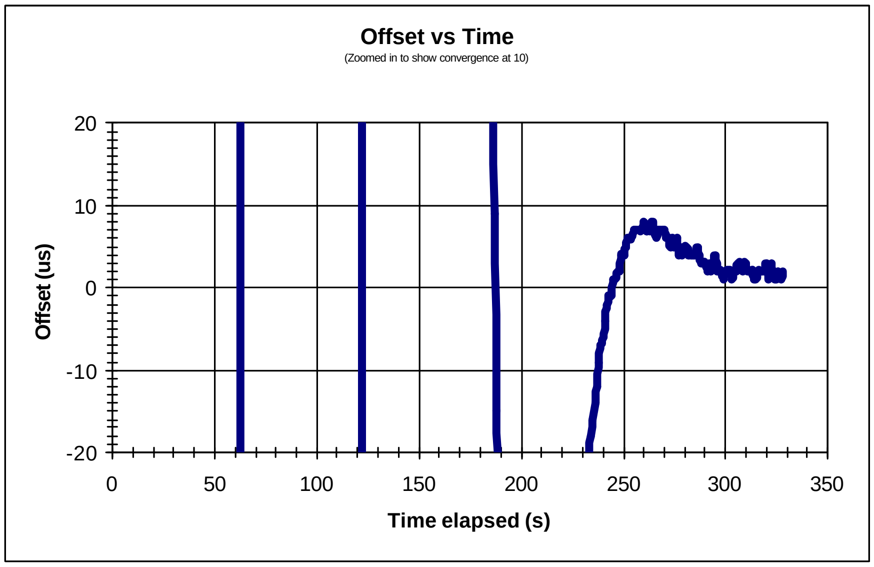

Figure 4-6 - Example of a Detailed Convergence Time Graph

A number of tests were performed. Three different ratios and four different d_f values were tested. The results are summarized in Table 4-3. Detailed results of each run are listed in Appendix A.

Table 4-3 - Summary of Clock Synchronization Tests

Test Number	**Initial Offset (s)**	d_f	d_f	$\frac{d_f}{d_f}$ **Ratio**	**Convergence Time (s)**	**Number of Overshoots**
1	+5	200	10	20	237	3
2	+5	100	5	20	212	1
3	+5	40	2	20	261	0
4	+5	100	10	10	238	3
5	+5	50	5	10	126	3
6	+5	20	2	10	139	1
7	+5	50	10	5	288	11
8	+5	25	5	5	151	7
9	+5	10	2	5	84	3
10	+5	20	10	2	485	29
11	+5	10	5	2	275	28
12	+5	4	2	2	∞	∞
13	-5	10	2	5	86	3
14	0	10	2	5	38	7

The "initial offset" column indicates the value of the `OFFSET` macro used to force the initial clock set to be incorrect. The initial time set always introduces an additional –56 ms of error into the time. So, even when `OFFSET` was 0, there were 56 ms of time correction needing to be done.

When plotting convergence time versus ratio (see Figure 4-7), each value of d_f produces its own curve. Each curve has a minimum point. This minimum is the ideal ratio for that value of d_f. The ideal ratio varies from value to value.

Lower d_f values have lower curves. This means they generally converge quicker. The quickest convergence was obtained with $d_f = 2$ and $d_f = 10$. This is a 5:1 ratio. Based on these results, these values are used as the OSKAR default.

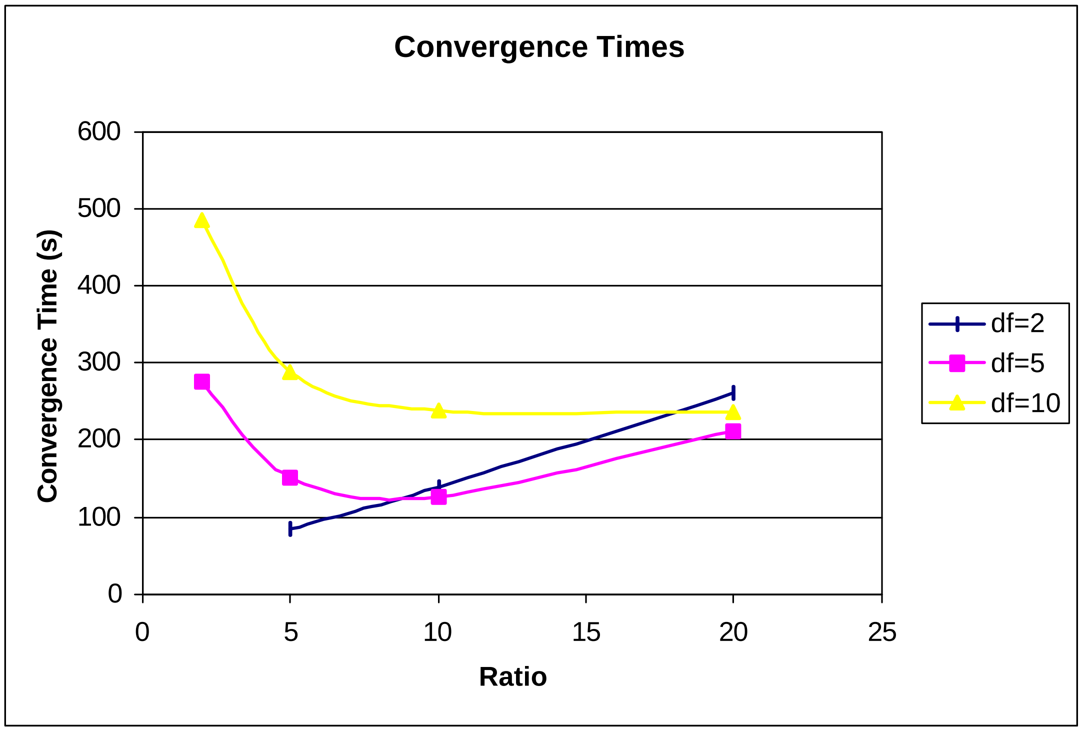

Figure 4-7 - Clock Synchronization Convergence Times

There does not appear to be a tradeoff between convergence time and number of overshoots in all cases. Figure 4-8 plots the number of overshoots by d_f values and ratios. Better d_f values also have fewer overshoots. Within a given d_f value, higher ratios do yield fewer overshoots at the expense of increased convergence time. This clearly demonstrates the phase component's tendency to over steer, and the frequency component's ability to keep it in check.

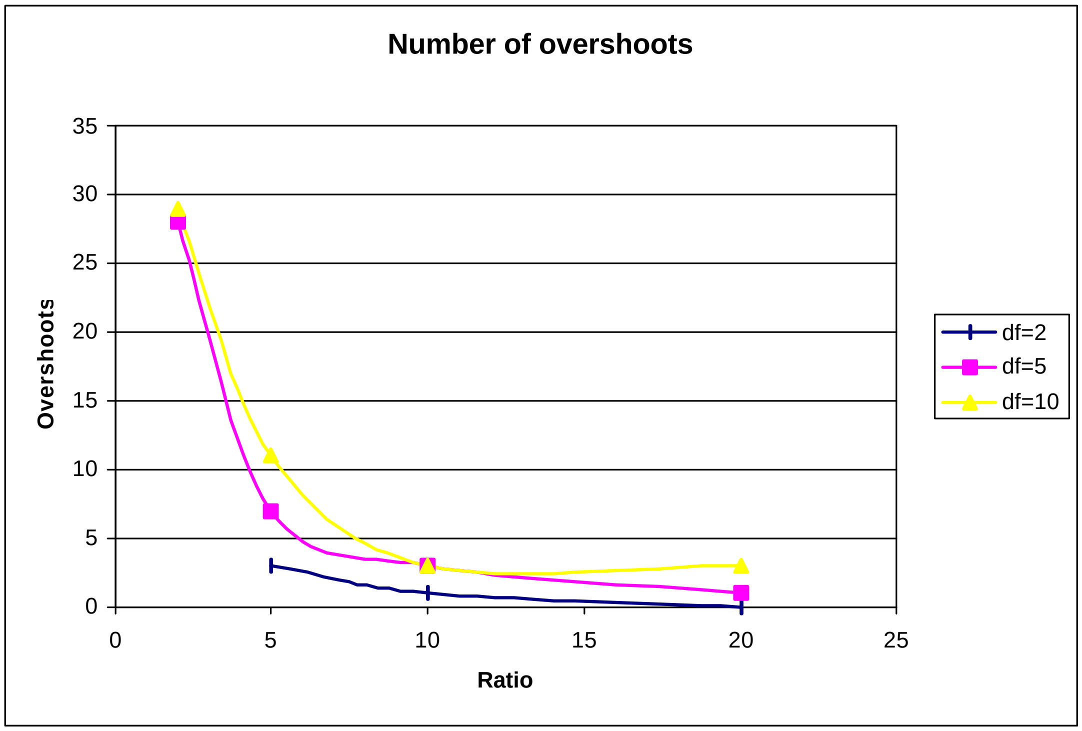

Figure 4-8 - Clock Synchronization Overshoots

4.4.2 Long Term Stability Test

We must also examine the OSKAR clock's ability to remain running if the external time source fails. Relying on 100% availability of the GPS signal would be a serious design flaw. Remember that the GPS signal is used only to discipline the internal clock, not to provide the primary time source. So if the external time source is disconnected, the OSKAR clock will continue to free run based on its last corrected frequency.

4.4.2.1 Test Conditions

To perform this test, we once again use the sample program `clock.c`. An IBM PC350-100Mhz machine executes the program. At startup time, the GPS is available. The system is allowed to synchronize to within 10µs of UTC. At 12/20/2000 17:28:06 GMT, the GPS was disconnected. Eight days later, at 12/28/2000 23:14:00 GMT, the GPS was reattached. The program output from before and after the interruption is captured for examination.

4.4.2.2 Expected Results

The PC clock will free run based on its last calibrated frequency value. When the GPS is reconnected, the clocks should be close to each other. Exactly how far off is impossible to predict, because we are dealing with the error rates of an oscillator. If the oscillator were perfect, there would be no offset at all.

Most inexpensive crystal oscillators, such as those used in computer clocks, have a published tolerance of +/- 50 parts per million (PPM). This equates to +/- 4.32 seconds per day. So, we will anticipate that after 8 days, OSKAR's clock should be no more than 34.56 seconds off. An greater offset indicates a serious problem in the OSKAR clock's ability to keep time. A lesser offset can be considered within the bounds of oscillator error.

4.4.2.3 Actual Results

```
977333286.00003   977333286.00000   1000001        3      0      0
BAD PPS Signal
BAD PPS Signal
 978046406.306797  978046406.000000  1000001   306797      0   1533
 978046407.306798  978046407.000000  1000001   306798      0   1533
```

Figure 4-9 - OSKAR Output from Long-Term Stability Test

Figure 4-9 shows the output of the program. After free running for 8 days, the PC clock was only off by 306,797µs. This represents 0.44 PPM, which is far less than the expected 50 PPM. This demonstrates OSKAR's ability to calibrate inexpensive hardware to better than rated tolerances.

4.5 Application: One-Way Delay Measurement

It is useful to test OSKAR performing a real world application. In this test, we use two OSKAR machines to make delay measurements across a wide area network. Most network performance measurements involve sending a packet to a remote host and timing how long it takes to get back. Thus, all that can be measured is the round trip time. One usually divides the time in half and assume that transmit and receive delays were equal.

Since OSKAR has globally distributed time synchronization via GPS, we can use timestamps from the two machines to measure a one way delay. For the purpose of demonstration, we assume that both GPS units are simultaneously producing identical time signals.

4.5.1 Test Conditions

Two OSKAR machines are set up at different geographic locations. Each is connected to a Trimble GPS unit and synchronized to GPS time. Both are connected to the internet via their Ethernet cards.

Once per second, machine A sends a UDP packet to machine B. The packet contains a timestamp of the time the packet is sent. Machine B receives the packet and marks the time it has received it. By subtracting the send time from the receive time, we can determine how long it took the packet to make the trip.

Because of logistical difficulties, the tests were conducted in less than ideal networking conditions. Both OSKAR machines were behind Linux firewalls. So, in addition to network routing delays, we also have firewall delays.

4.5.2 Expected Results

An ICMP ping between the two firewalls indicates the round trip time is 379ms. From this we can bound the one way time between 0 and 379ms. However, the ping time does not include routing packets through the firewalls. This routing time is unknown. Assuming 100ms per segment, our total one way time could reach as high as 579ms.

4.5.3 Actual Results

The test was executed for 30 minutes. 1800 sample measurements were collected. The resulting times were grouped into 10ms buckets and plotted as a histogram in Figure 4-10. The median time was approximately 330ms. In no case was the delay ever less than 250ms. In a few cases, the delay exceeded 425ms. This is consistent with what one would expect from a network, where occasional “traffic jams” cause packets to be delivered late, most arrive at about the same time, and in no case do packets ever arrive faster than is physically possible.

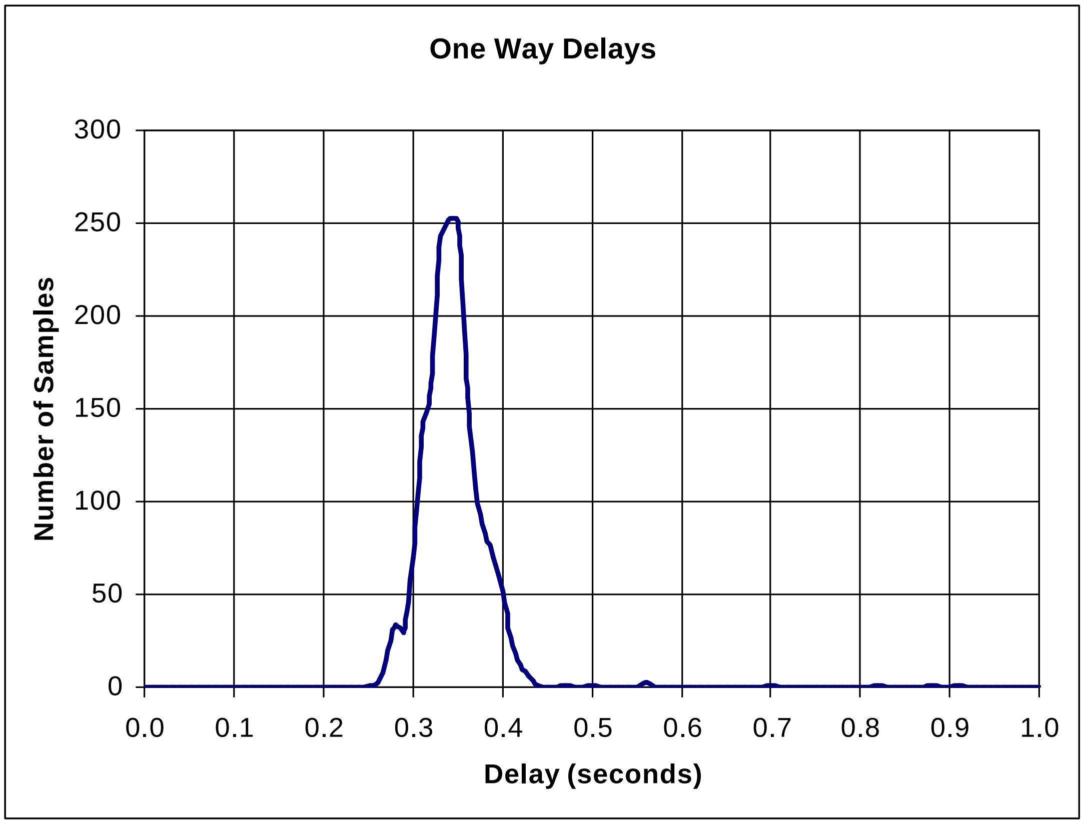

Figure 4-10 - Histogram of One-Way Delays

5 Conclusion

OSKAR is a useful test bed for hard real time systems. It is implemented using simple, readable, documented code. It implements advanced real time concepts, such as dynamic priority scheduling and ceiling protocols. The battery of tests conducted and described in Chapter 4 demonstrate that the system performs as described.

OSKAR has practical uses. Researchers can use it to test out new real time ideas. The `busy_wait()` call allows obscure scheduling situations to be created. With a little modification, OSKAR can be used as a proving ground for new scheduling algorithms and ceiling protocols.

OSKAR's globally distributed clock synchronization allow it to be used for a number of applications. It can be used to measure relative times of events at two different geographic locations within good error bounds (100µs). One such application could be the measurement of one-way network delays. Distributed systems researchers can also make great use of OSKAR. Global clock synchronization allows time based distributed resource sharing, locking, and coprocessing.

OSKAR is equally well suited in the hands of the student. For two semesters, students at North Carolina State University have used OSKAR to get a hands-on feel for rate monotonic scheduling and priority ceiling protocols. Students studying conventional operating systems can also benefit by having a simple system to examine and tinker with.

There are still a number of features that need to be added to OSKAR. OSKAR would greatly benefit by having an industry standard C runtime library (such as glibc) ported to it. A floppy disk boot loader and disk drivers would allow OSKAR systems to break free of the network and be used in far away remote locations, or in the bellies of robots. The kernel could be modified to allow memory protection between processes. It would not be difficult to add dynamic process creation. And, in this age of network-centric computing, more networking protocols are always welcome.

Bibliography

[GAL95]. Gallmeister, Bill O. (1995). *POSIX.4: Programming for the Real World*. Sepastopol, California: O'Reilly and Associates.

[IBM85]. International Business Machines (1985). *Personal Computer AT Technical Reference*. (pp. 1-22)

[LIU73]. Liu, C.L. and Layland, James W (1973). "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *Journal of the ACM*. Vol 20 no 1. (pp. 46-61).

[MIL94] Mills, David L (1994). *UNIX Kernel Modifications for Precision Time Synchronization* Electrical Engineering Technical Report, University of Delaware.

[MS00]. Microsoft Corporation (2000). *Upgrading to Windows 2000*. http://www.microsoft.com/windows2000/upgrade/upgradereqs/default.asp

[QNX01]. QNX Software Systems Ltd (2001). *QNX Neutrino datasheet*. http://www.qnx.com/literature/pdf/qnx_neutrino.pdf

[RUB98]. Rubini, Allesandro (1998). *Linux Device Drivers.* Sepastopol, California: O"Reilly and Associates. **(**p.194)

[SHA90]. Sha, Lui et al (1990). "Priority Inheritance Protocols: An Approach to Real-Time Synchronization"**.** *IEEE Transactions on Computers*. Vol 19 no 9. (pp 1175-1185).

[SIL94]. Silberschatz & Galvin (1994). *Operating System Concepts.* 4th ed. New York: John Wiley & Sons. (p.78).

[STE94]. Stephens, W. Richard (1994). *TCP/IP Illustrated*. Vol 1. Addison Wesley. (pp 53-55).

[WR01a]. WindRiver Systems (2001). *VxWorks datasheet.* http://www.windriver.com/pdf/vxworks-ds.pdf

[WR01b]. WindRiver Systems(2001). *pSOSystem 3 datasheet*. http://www.windriver.com/pdf/psosystem3_ds.pdf

[YOD97]. Yodaiken, Victor (1997). *The RTLinux Manifesto*. New Mexico Institute of Technology. http://www.rtlinux.org/documents/papers/rtmanifesto/rtlmanifesto.html

Appendices

Appendix A - Clock Synchronization Results

This section contains the results of each clock synchronization test. The test inputs and results are summarized in Table 4-3. Section 4.4 explains the two graphs created by each test run.

Test run 1 was conducted with an offset of +5 seconds. d_j was 200 and d_f was 10. Figure A-1 displays the convergence graph from this run. Figure A-2 displays the overshoot detail from this run.

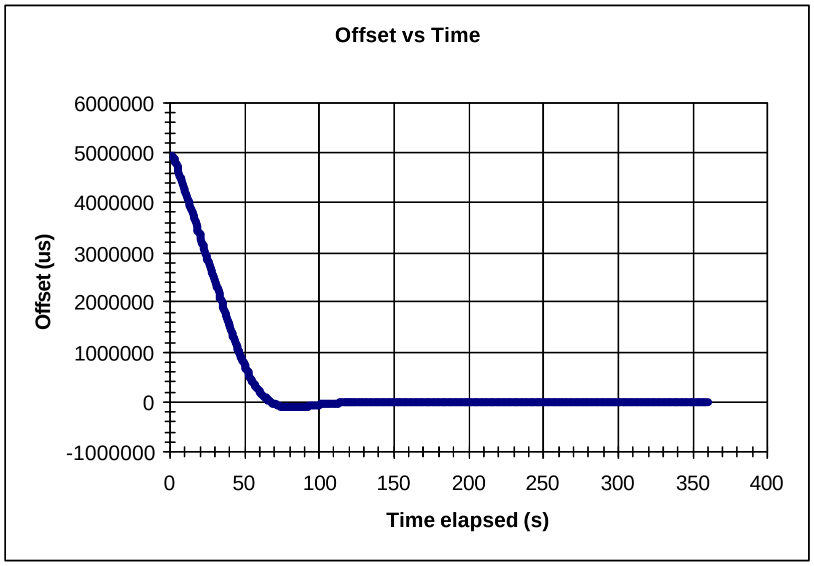

Figure A-1 - Offset vs. Time for Test Run 1

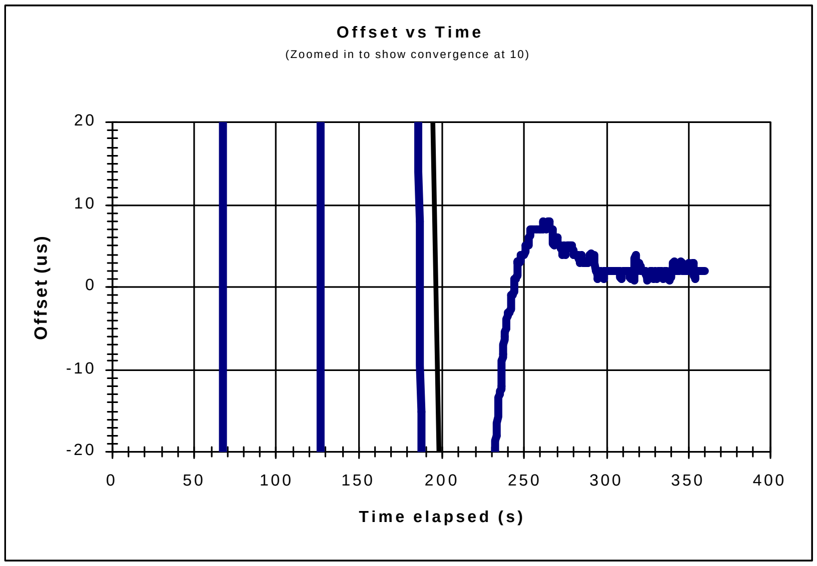

Figure A-2 - Overshoot Detail for Test Run 1

Test run 2 was conducted with an offset of +5 seconds. d_j was 100 and d_f was 5. Figure A-3 displays the convergence graph from this run. Figure A-4 displays the overshoot detail from this run.

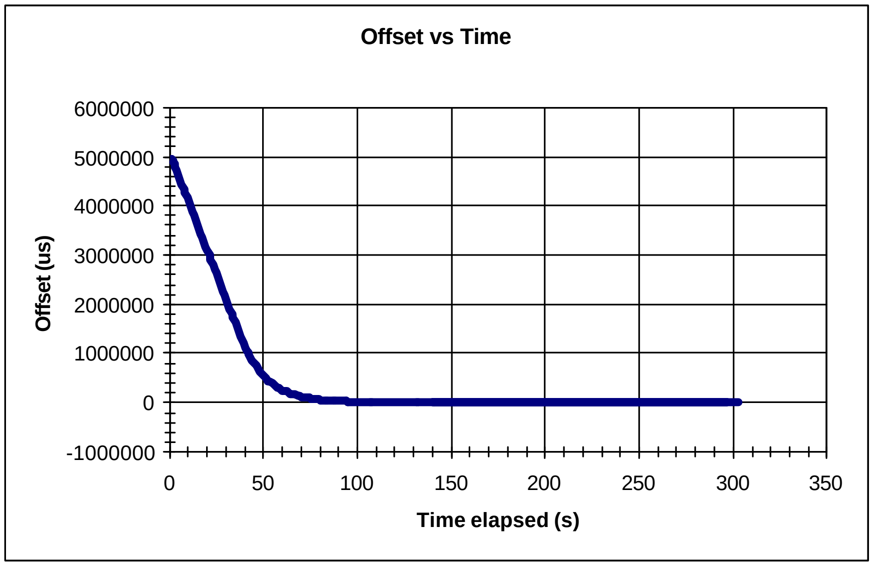

Figure A-3 - Convergence Graph for Test Run 2

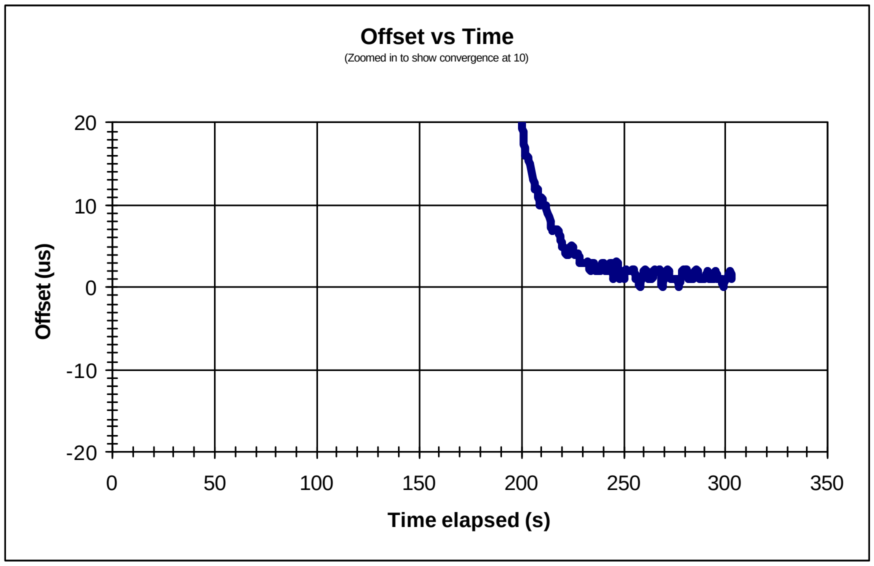

Figure A-4 - Overshoot Detail for Test Run 2

Test run 3 was conducted with an offset of +5 seconds. d_j was 40 and d_f was 2. Figure A-5 displays the convergence graph from this run. Figure A-6 displays the overshoot detail from this run.

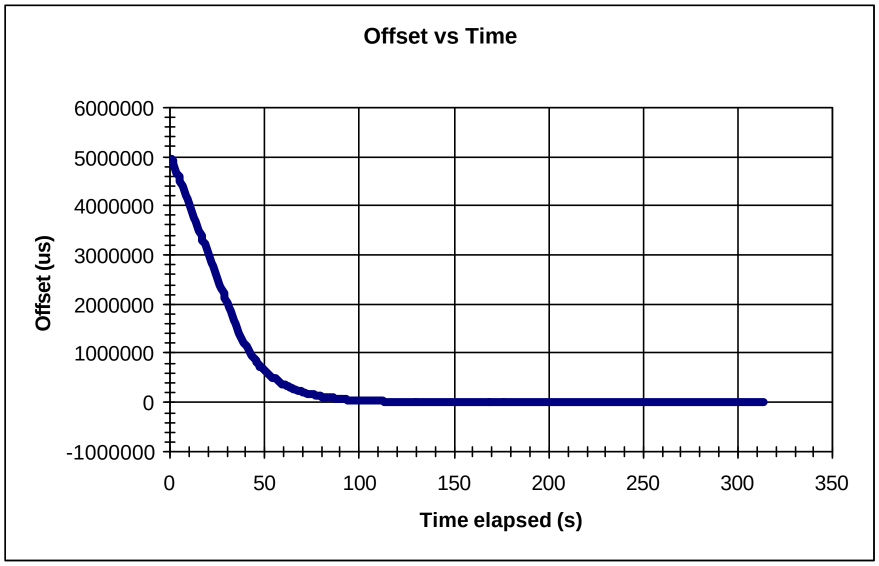

Figure A-5 - Convergence Graph for Test Run 3

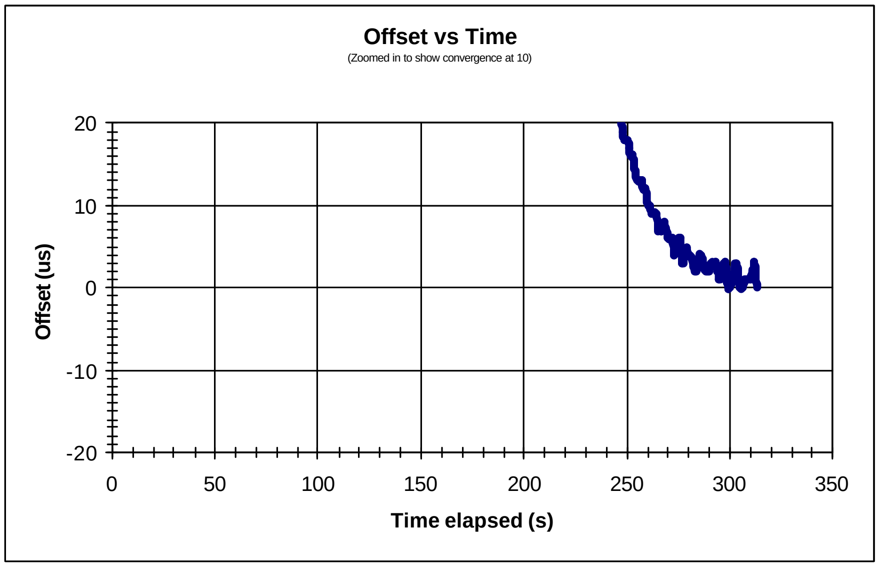

Figure A-6 - Overshoot Detail for Test Run 3

Test run 4 was conducted with an offset of +5 seconds. d_j was 100 and d_f was 10. Figure A-7 displays the convergence graph from this run. Figure A-8 displays the overshoot detail from this run.

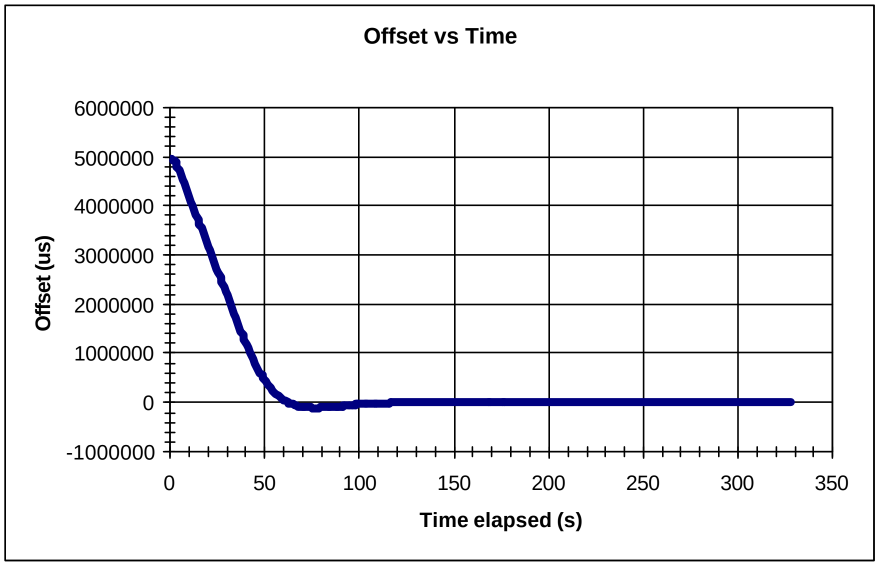

Figure A-7 - Convergence Graph for Test Run 4

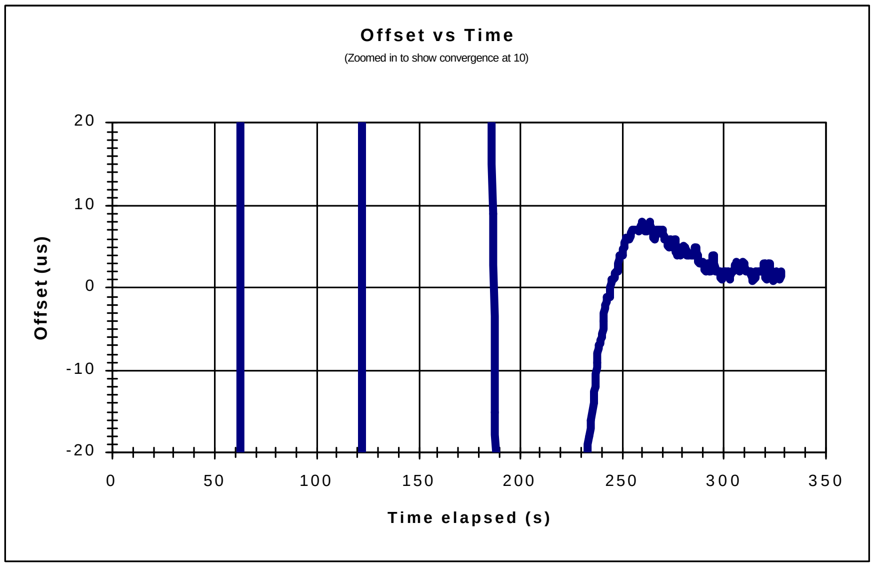

Figure A-8 - Overshoot Detail for Test Run 4

Test run 5 was conducted with an offset of +5 seconds. d_j was 50 and d_f was 5. Figure A-9 displays the convergence graph from this run. Figure A-10 displays the overshoot detail from this run.

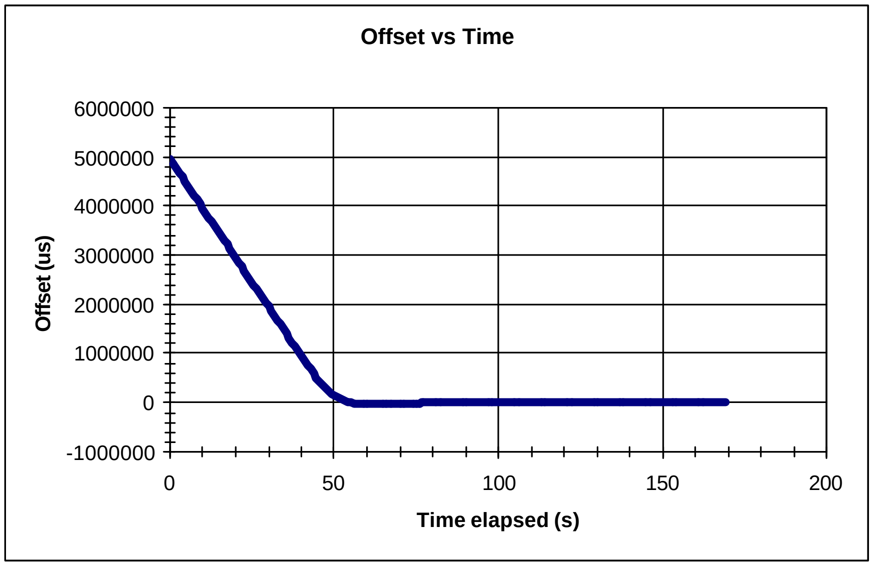

Figure A-9 - Convergence Graph for Test Run 5

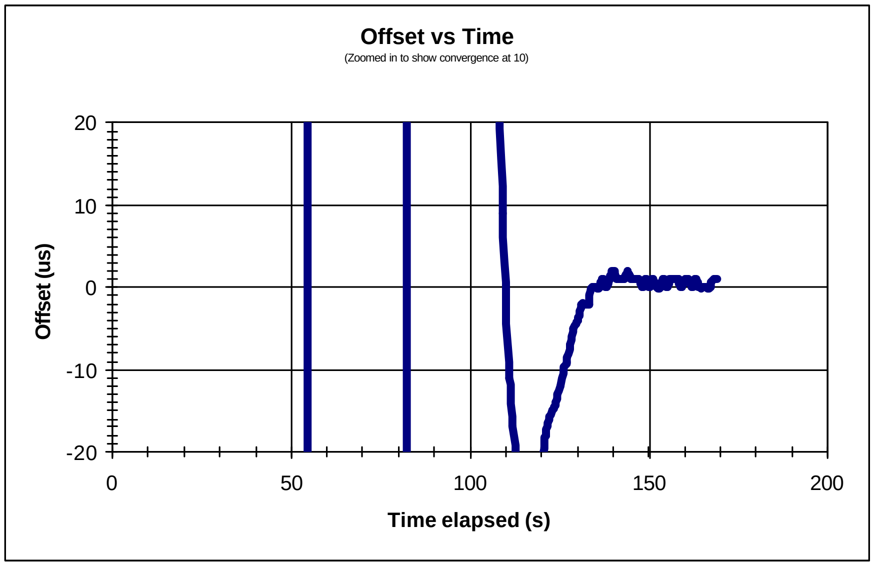

Figure A-10 - Overshoot Detail for Test Run 5

Test run 6 was conducted with an offset of +5 seconds. d_j was 20 and d_f was 2. Figure A-11 displays the convergence graph from this run. Figure A-12 displays the overshoot detail from this run.

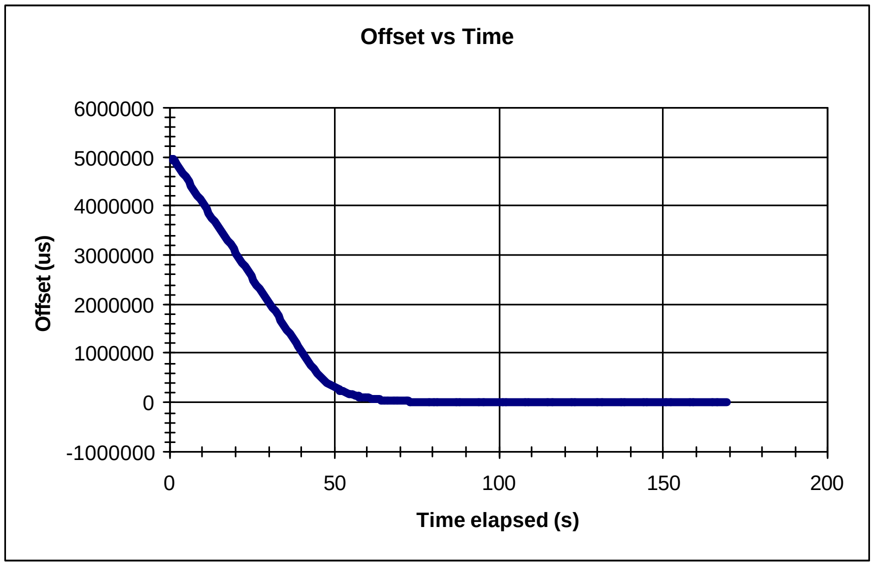

Figure A-11 - Convergence Graph for Test Run 6

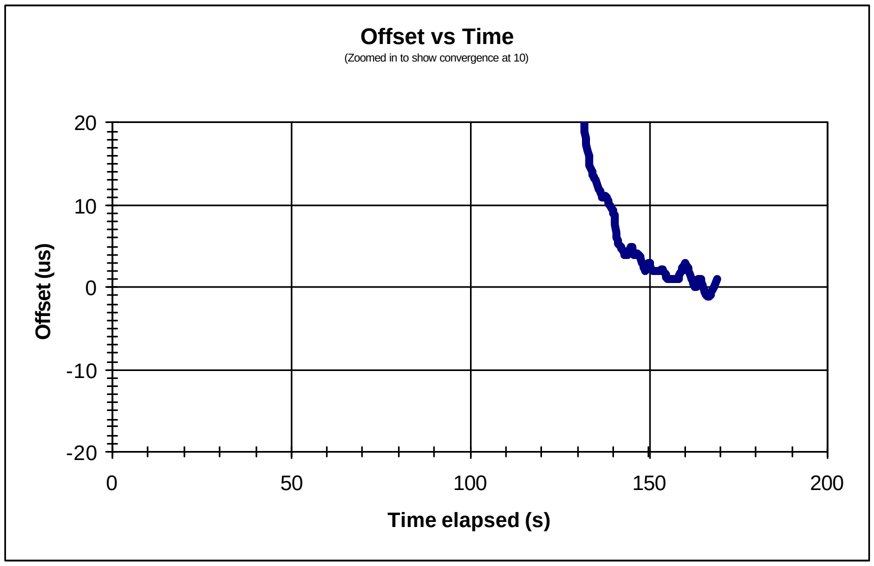

Figure A-12 - Overshoot Detail for Test Run 6

Test run 7 was conducted with an offset of +5 seconds. d_j was 50 and d_f was 10. Figure A-13 displays the convergence graph from this run. Figure A-14 displays the overshoot detail from this run.

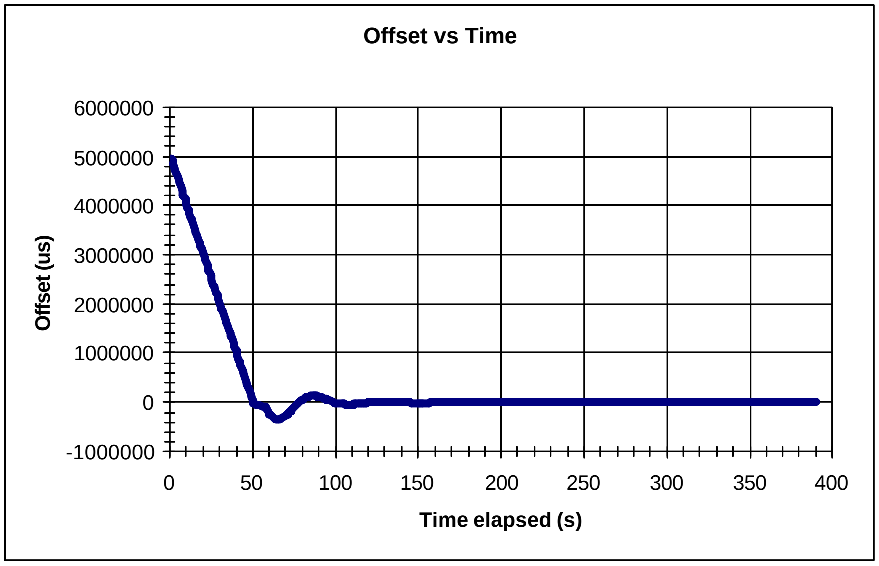

Figure A-13 - Convergence Graph for Test Run 7

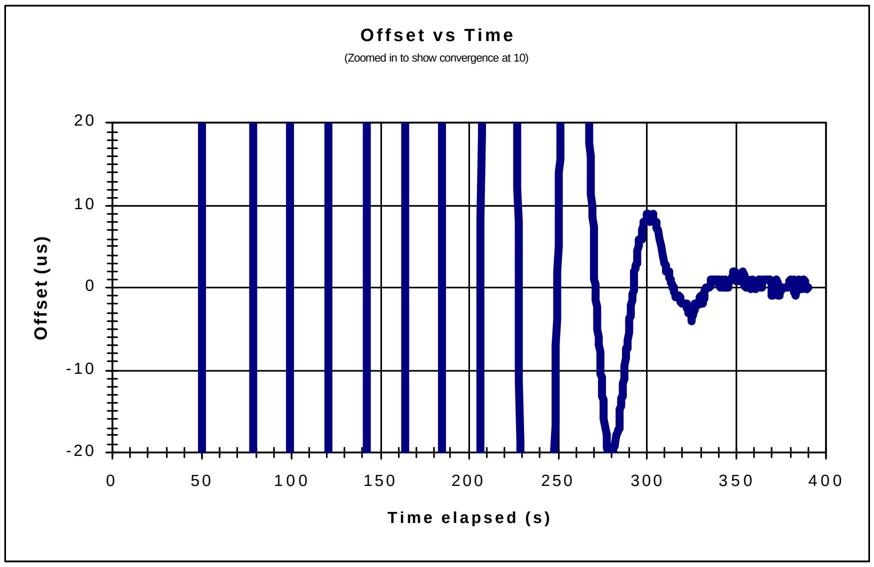

Figure A-14 - Overshoot Detail for Test Run 7

Test run 8 was conducted with an offset of +5 seconds. d_j was 25 and d_f was 5. Figure A-15 displays the convergence graph from this run. Figure A-16 displays the overshoot detail from this run.

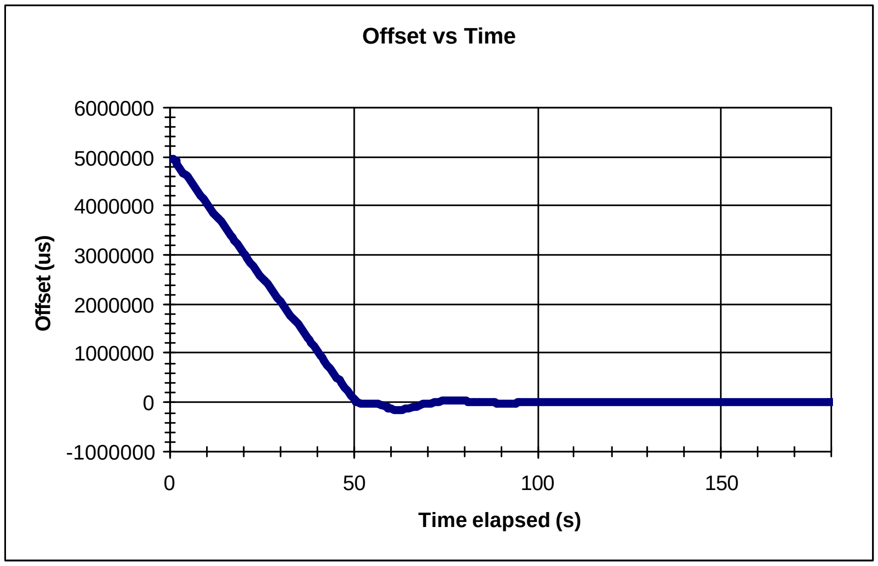

Figure A-15 - Convergence Graph for Test Run 8

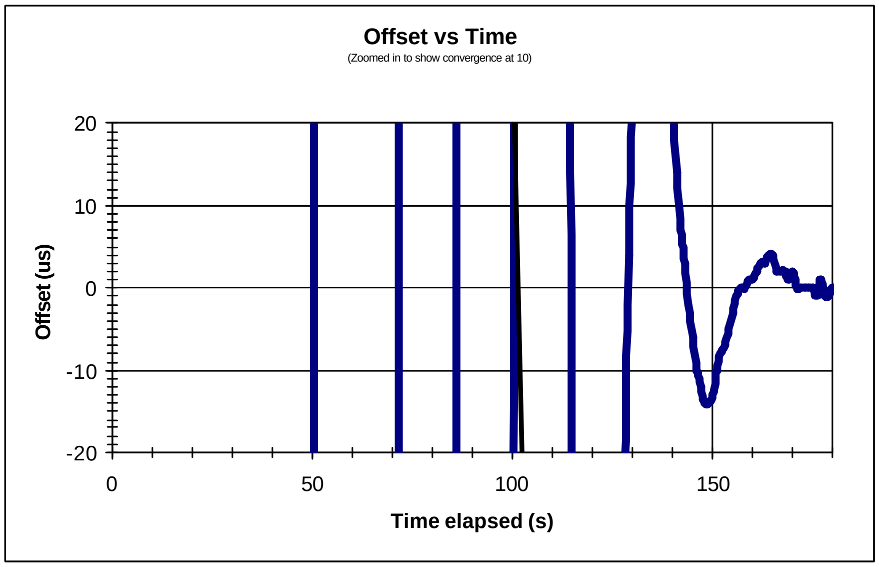

Figure A-16 - Overshoot Detail for Test Run 8

Test run 9 was conducted with an offset of +5 seconds. d_j was 10 and d_f was 2. Figure A-17 displays the convergence graph from this run. Figure A-18 displays the overshoot detail from this run.

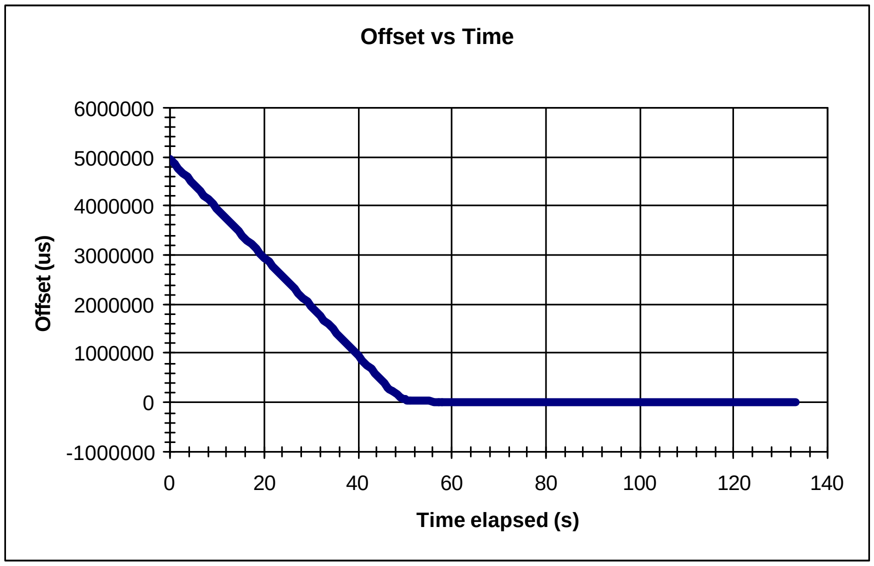

Figure A-17 - Convergence Graph for Test Run 9

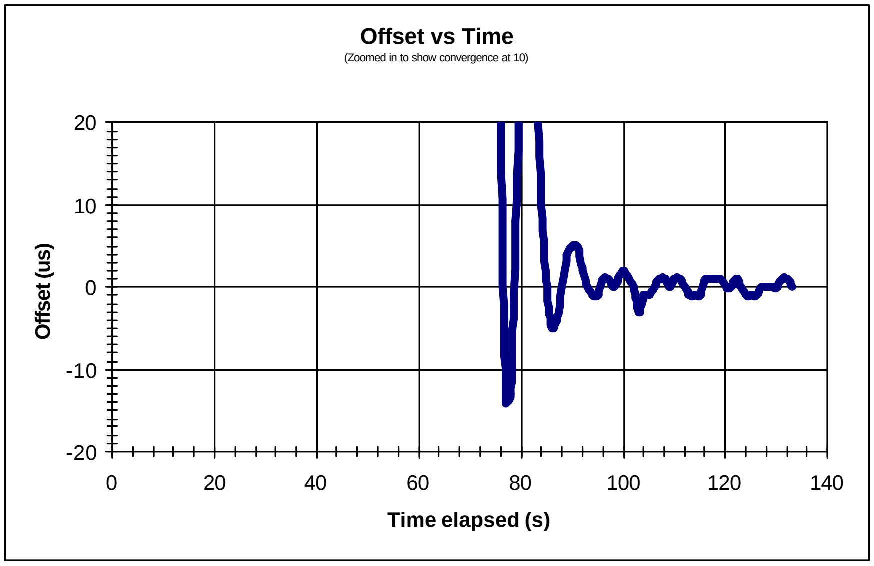

Figure A-18 - Overshoot Detail for Test Run 9

Test run 10 was conducted with an offset of +5 seconds. d_j was 20 and d_f was 10. Figure A-19 displays the convergence graph from this run. Figure A-20 displays the overshoot detail from this run.

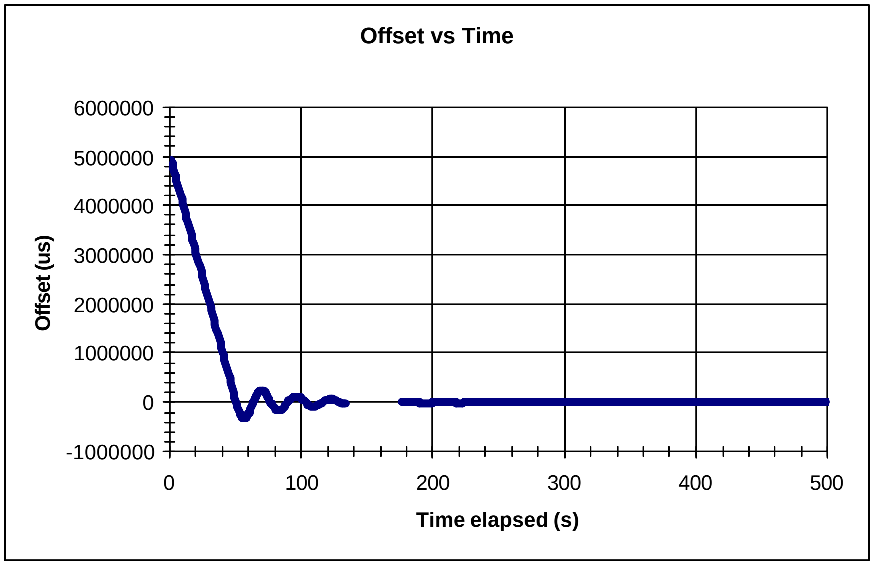

Figure A-19 - Convergence Graph for Test Run 10

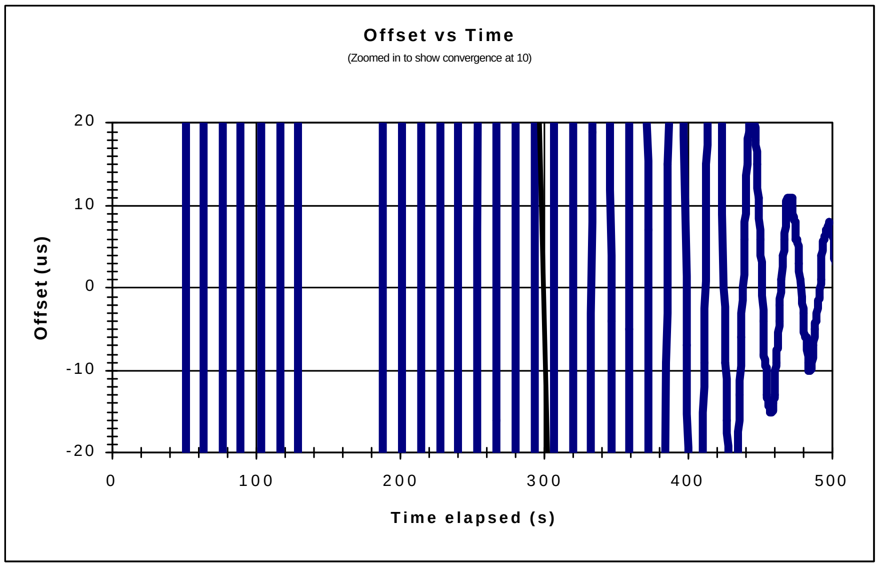

Figure A-20 - Overshoot Detail for Test Run 10

Test run 11 was conducted with an offset of +5 seconds. d_j was 10 and d_f was 5. Figure A-21 displays the convergence graph from this run. Figure A-22 displays the overshoot detail from this run.

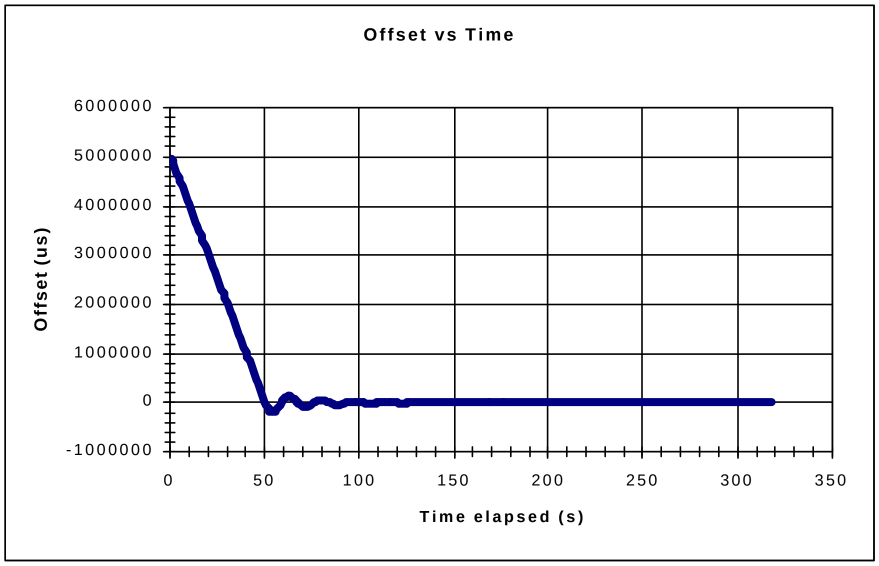

Figure A-21 - Convergence Graph for Test Run 11

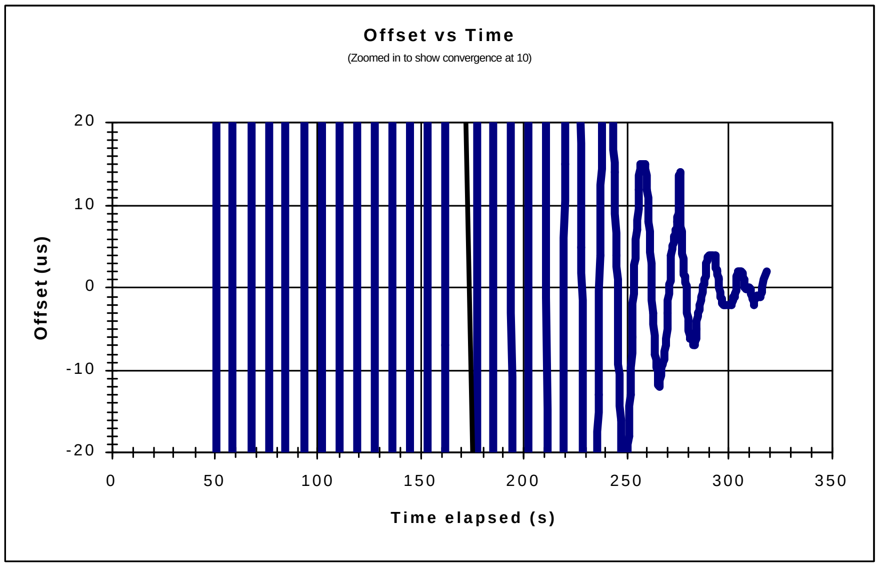

Figure A-22 - Overshoot Detail for Test Run 11

Test run 12 was conducted with an offset of +5 seconds. d_j was 4 and d_f was 2. Figure A-23 displays the convergence graph from this run. The algorithm never converged, so no overshoot detail is provided.

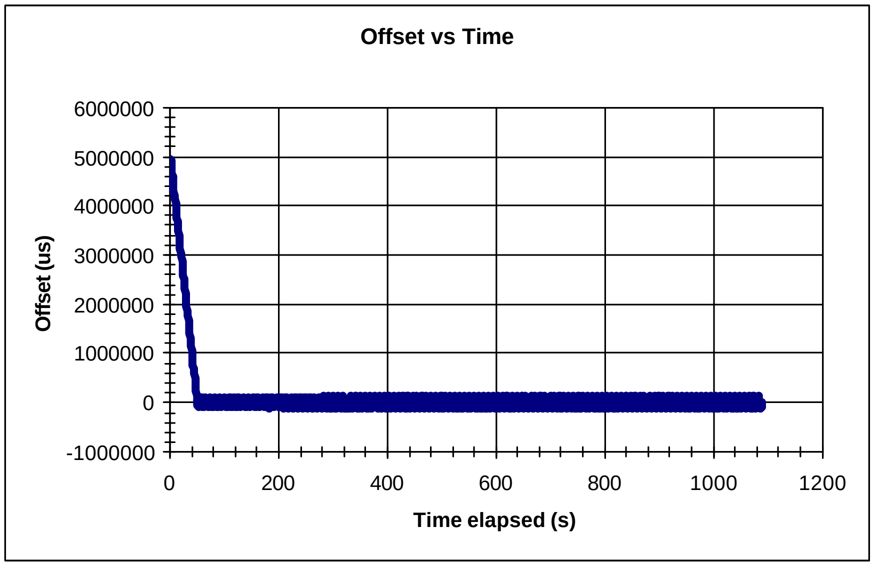

Figure A-23 - Convergence Graph for Test Run 12

Test run 13 was conducted with an offset of -5 seconds. d_j was 10 and d_f was 2. Figure A-24 displays the convergence graph from this run. Figure A-25 displays the overshoot detail from this run.

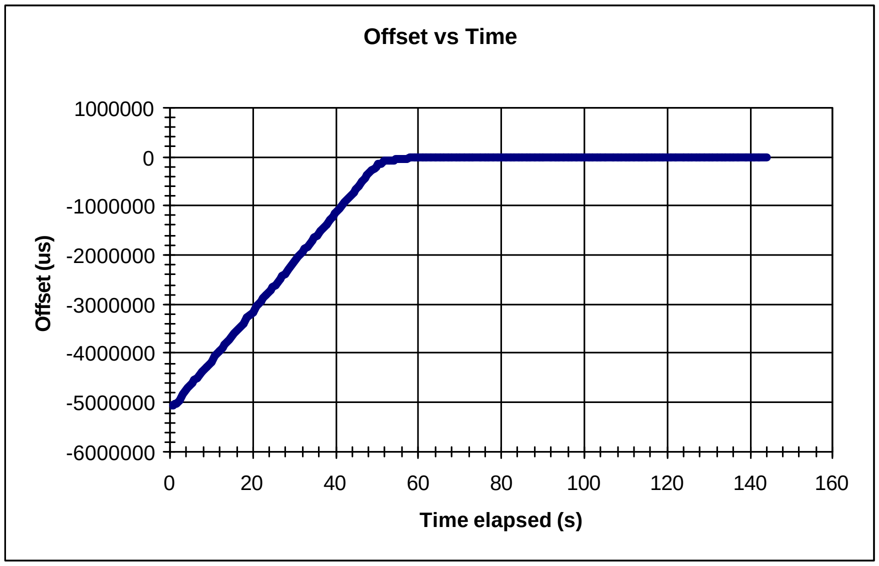

Figure A-24 - Convergence Graph for Test Run 13

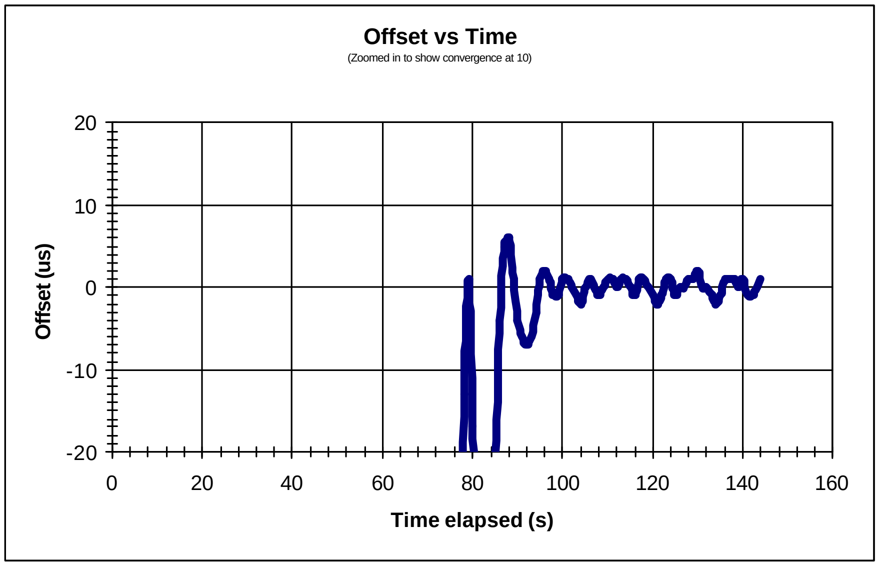

Figure A-25 - Overshoot Detail for Test Run 13

Test run 1 was conducted with an offset of 0 seconds. d_j was 10 and d_f was 2. Figure A-26 displays the convergence graph from this run. Figure A-27 displays the overshoot detail from this run.

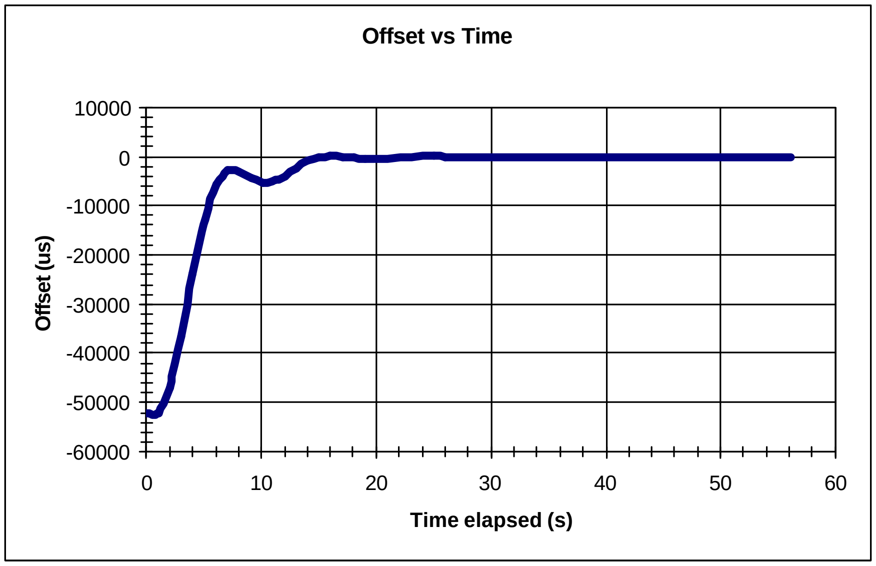

Figure A-26 - Convergence Graph for Test Run 14

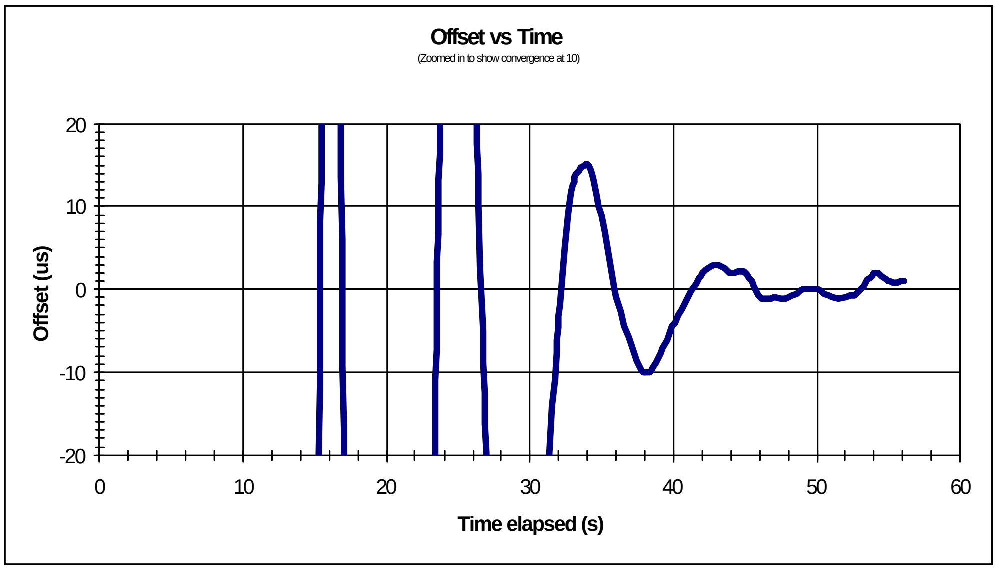

Figure A-27 - Overshoot Detail for Test Run 14

Appendix B - OSKAR Documentation and Availability

Version 3.0 of OSKAR is included with this document. Click here to download it. It is packaged as a compressed TAR file. To use OSKAR, download this file to a Linux machine and issue the command:

```
    tar –xzf oskar-3.0.tar.gz
```

at the Linux command prompt. Once unpacked, see the file called "readme" in the "doc" subdirectory.

More recent versions of OSKAR can be obtained from the OSKAR web site at (http://www.mindspring.com/~pinnix/oskar).

The OSKAR User's Guide is included with the distribution. Look under the "doc" subdirectory for the file "index.html". An online copy is available at (http://www.mindspring.com/~pinnix/oskar/doc.)