1. Introduction

Various projects involving the interfacing of I/O ports of a Computer system require Real-time I/O access on Windows XP. But Windows XP inherently does not provide any Real-time solutions and third party Real-time solutions for Windows XP are extremely costly. Keeping this in view we have been assigned the task of developing a hard Real-time solution which provides interfacing of I/O ports of a computer system to applications running on Microsoft Windows XP.

1.1 Terms of Reference

The terms of reference of the project are:

- Real-time interfacing of I/O ports of a computer system
- Context Switch - less than 1 millisecond
- Interrupt Latency 5-10 milliseconds
- Cost Effective - No licensing fee
- Final access should be through Windows XP
- Scalability & Portability – Source code should be available
- Ports to be interfaced
    - Parallel
    - Serial
    - USB
    - Ethernet
    - FireWire
    - PS/2
    - Audio

2. Study of various technologies available

The various technologies that were analyzed are:

- Windows XP Hard Real-time Kernels: RTX, Hyperkernel, and InTime provide hard real-time performance on Windows XP by modifying the Windows HAL. But these kernels require development and run-time licenses.

- Windows CE .NET: This also provides hard real-time access, but requires licenses.

- Windows XP and 2003 Server DDK (Kernel-mode): Driver development using Windows DDK is a time-consuming process. Drivers so developed will provide soft real-time performance only.

- Windows XP Win32 API (User-mode): Drivers developed using Win32 API will not provide even soft real-time performance.

- Linux Hard Real-time Kernels: Micro-kernels like RTLinux, and RTAI provide hard real-time performance on Linux systems. Real-time drivers for some ports are available. RTAI provides LXRT interface through which processes can be run in Hard real-time from the user mode. Thus, ports for which real-time drivers are not available can be interfaced in real-time using LXRT, and ordinary Linux drivers

3. Real-time Systems Basics

A real-time system is one in which the correct operation of the system depends not only on the results that are delivered, but when they are delivered. Real-time does not necessarily mean fast**;** rather, it refers to how deterministic the response time characteristics of the system are. That is, the important measure is not average response time but worst-case response time. Real-time systems are further classified as hard or soft real-time systems. A hard real-time system is one in which the response time determinism requirement is absolute; for a soft real-time system, some small deviations are tolerated.

3.1 Shortcomings of Windows XP

The Microsoft Windows XP platform has been designed as a general-purpose operating system, suitable for use both as an interactive system on the desktop and as a server system on a network. It has not been developed with Real-time applications kept in mind and also there is no official Microsoft version of Windows XP that is Real-time.

Microsoft Windows XP is a message-driven, event-polling system, with nonpreemptive scheduling.

The shortcomings of Windows XP in real-time applications are:

- Priority inversion occurs in Windows XP, particularly in interrupt processing. This is a major stumbling block in Real-time application development.
- Windows XP has too few thread priorities (32 in number) and also the priorities are changed dynamically which results in our having only 6 priority levels in our hands.
- Windows XP makes opaque and nondeterministic scheduling decisions which is a basic glitch in Real-time application because determinism, which is the basic step in any Real-time application, will not be possible.
- Windows XP protects the hardware by providing indirect hardware access through the Hardware Abstraction Layer (HAL.DLL), due to which a lot of features of the hardware gets locked and applications accessing hardware get significantly slow.
- Paging in Windows XP reduces the speed of any application.

3.2 Achieving Hard Real-time in Windows XP

While developing a Hard Real-time extension for Windows XP the following points have to be kept in mind.

- The Hard Real-time system should be able to preempt Windows XP anywhere, at least outside critical Windows XP interrupt-processing code.
- Thus the Hard Real-time applications should have the highest possible priority. Its priority should be higher that that of the Windows XP kernel itself.
- It should be protected from interrupts. No interrupt should be able to preempt a real-time application.

- It should be protected from the frequent phenomenon of a Windows crash.
- The Hard Real-time extension should always reside in the non-paged system memory. Therefore the extension should be as compact as possible.
- Direct assembly level type access should be available to the hardware so that any hardware can be accesses directly and can be run at any supported speed or frequency.

Since most of the shortcomings of the Microsoft Windows XP platform are due to its thread model and thread scheduler, the Hard Real-time extension should have its own thread model with its own scheduler. Likewise, the Windows XP platform's synchronization objects such as events, semaphores and mutexes lack the necessary real-time semantics (in particular, they neither ready threads waiting on an object in priority order nor prevent priority inversion). For these reasons, the extension should implement its own synchronization objects. Generally the Real-time threads are assigned higher priorities and Windows XP threads are given the lowest possible priority so that when the system completes the execution of all Real-time threads it returns to executing normal Windows XP threads until a Real-time thread comes in, in which case the Windows XP thread is preempted.

3.3 Windows XP based Real-time Kernels

The most widely used solution is to use a technology called a double kernel technology wherein the Hard Real-time kernel complements the Windows XP kernel and runs parallel to it. This Hard Real-time kernel should be completely independent of the Windows XP kernel so that a crash in the Windows XP kernel does not hamper the Real-time kernel.

The notion of preempting high-level Interrupt Request Number (IRQ) activity of Windows XP and its drivers for unbounded periods of real-time activity may be dangerous. Yet, such events are commonplace and Windows XP is designed to handle them: high-interrupt request level (IRQL) events intrude on lower-IRQL ones, bus-mastering by DMA peripherals and System Management Mode processing defer even the highest-level interrupt processing, and PCI devices may stall CPU accesses to the input/output (I/O) space. Thus, from the Windows XP platform point of view, Hard Real-time kernel activity that steals its cycles is equivalent to taking and coming back from an interrupt.

Functional needs of the real-time subsystem would include inter-process communication (IPC) with the Microsoft Win32 subsystem, access to the Windows XP kernel functionality (interrupt management, port I/O, shutdown/crash handlers), and compatibility with Win32 at the source code level.

A number of Real-time Kernels are available which integrate with Microsoft Windows XP to provide hard real-time.

- Ardence RTX
- Nematron Hyperkernel
- TenAsys Intime

3.3.1 Ardence RTX

Ardence RTX is implemented as a collection of libraries (both static and dynamic), a real-time subsystem (RTSS) realized as a Windows XP kernel device driver, and an extended HAL (Refer fig 1). The subsystem implements the real-time objects and scheduler previously mentioned. The libraries provide access to the subsystem through a real-time API, known as RtWinAPI. RtWinAPI provides access to these objects. The RtWinAPI is callable from the standard Win32 environment and from within RTSS. Although using RtWinAPI from Win32 does not provide the determinism available within RTSS, it does allow much of the application development to be done in the more user-friendly Win32 programming environment rather than that provided by the DDK. All that is necessary to convert a Win32 program to an RTSS program is to re-link with a different set of libraries.

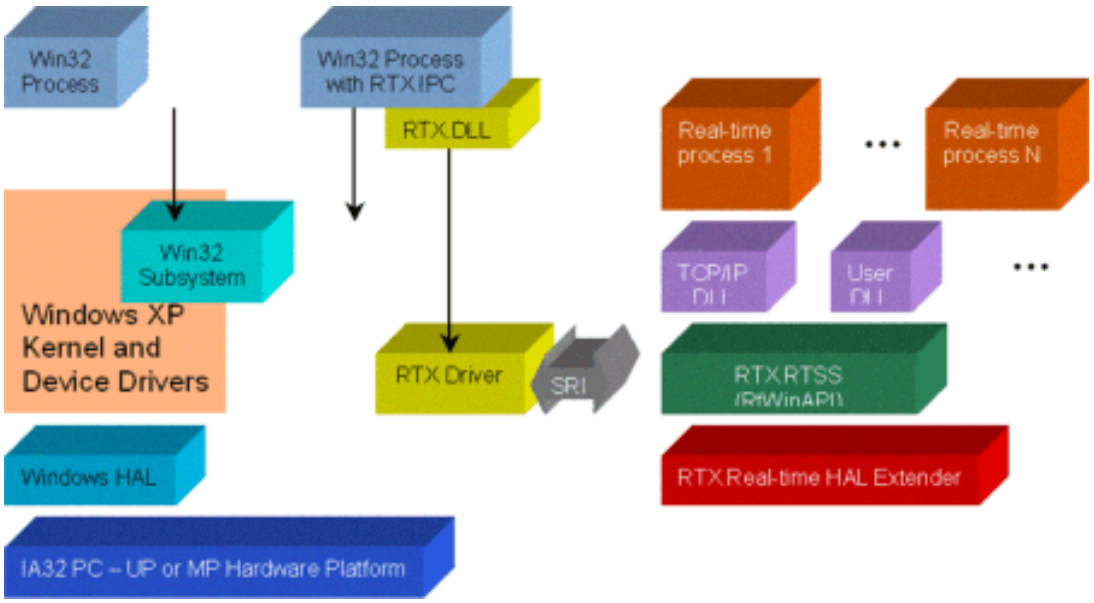


Fig 1
RTX Architecture

The Windows XP Service Control Manager directly loads RTX process and dynamic-link library (DLL) executable images -- into kernel nonpaged memory.

The Real-Time Hardware Abstraction Layer (HAL)

The HAL is the one piece of the Microsoft Windows XP platform system whose source is available for modification and extension. The HAL extension driver, starting at OS initialization time, performs dynamic HAL detection in memory, intercept interrupt, timer, and shutdown-related calls, and redirect them to their RTX counterparts (Refer fig 2).

RTX extends the HAL for three purposes:

- To add interrupt isolation between Windows XP and RTSS threads.
- To implement high-speed clocks and timers.

- To implement shutdown handlers.

Interrupt isolation means that it is impossible for a Windows XP thread or a Windows XP-managed device to interrupt RTSS. It is also impossible for a Windows XP thread to mask an RTSS-managed device. The HAL ensures that these conditions are met by controlling the processor's interrupt mask. When running an RTSS thread, all Windows XP-controlled interrupts are masked out. When a Windows XP thread calls to set the interrupt mask, the HAL, which is the software that actually manipulates the mask, ensures that no RTSS-controlled interrupt is masked out.

The Windows XP platform provides timers with a minimum period resolution of 1 millisecond. The RT-HAL lowers this to .1 milliseconds and provides a synchronized (with the timer) clock with a granularity of .8 s or better.

The real-time HAL also provides Windows XP shutdown management. An RTSS application can attach a Windows XP shutdown handler for cases where Windows XP performs an orderly shutdown, or crashes with the so-called Blue Screen of Death. An orderly shutdown allows RTSS to continue unimpaired and resumes when all RTSS shutdown handlers return. In a blue-screen stop, RTSS shutdown handlers run with certain limitations, unable to call Windows XP services (for example, new memory allocation). In practice, it means that a shutdown handler should clean up and reset any hardware state, possibly alert an operator, or switch to a hot spare when the system stops, due to either a normal shutdown or a crash.

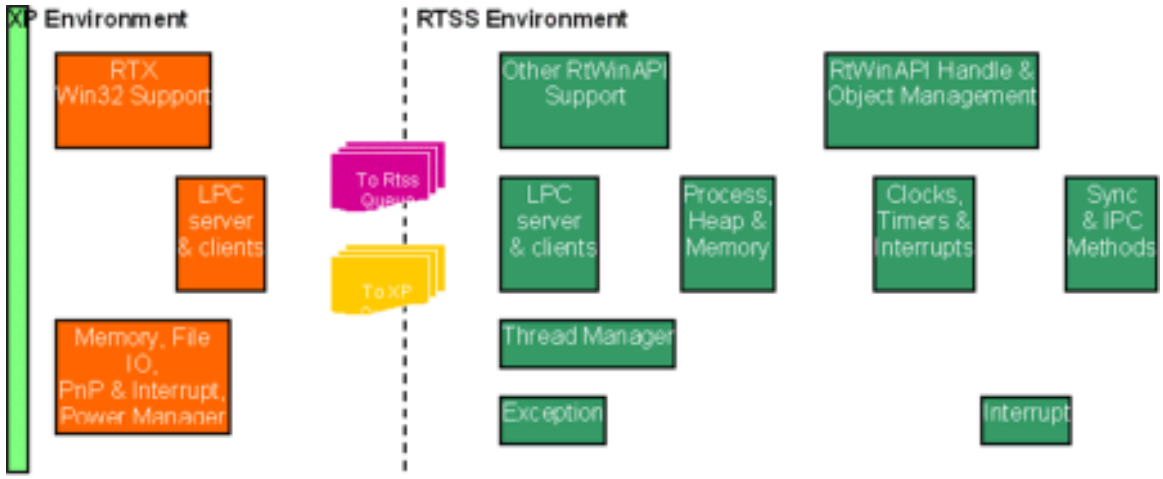


Fig 2
RTSS Detailed Architecture

RTX and Interrupt Latency

A switch from Windows XP to RTSS happens on an interrupt, either from the RTX high-speed clock or from another device generating RTX interrupts. Therefore, achieving RTX ISR determinism requires reducing Windows XP interrupt latency. Let us examine the sources of Windows XP platform ISR latencies without RTX present.

The most significant latency is IRQ masking by the Windows XP kernel and drivers, routinely done for periods up to several milliseconds through Windows XP KeRaise/LowerIrql calls. Windows XP and RTX interrupt processing naturally, masks interrupts, thereby adding to ISR latency.

RTX Interrupt Latency Reduction Techniques

RTSS entirely eliminates latencies from IRQL masking by the Windows XP platform and Windows XP drivers. The RT HAL performs interrupt isolation, reprogramming the programmable interrupt controller (PIC) when switching between the Windows XP platform and RTSS. The result is that RTX interrupts can always interrupt Windows XP, while RTX masks all Windows XP interrupts while RTSS is running.

Processor-level interrupt masking, on the other hand, cannot be defeated, other than through the perilous use of x86 NMIs (non-maskable interrupts). RTX adopts a static solution by hooking gratuitous cases of interrupt preemption to use IRQ locks instead. The RTX Dynamic Hook functionality scans the HAL for signatures of such operations, hooking them to use spin locks (or IRQ-based synchronization on a uniprocessor) instead.

These techniques provide worst-case interrupt latencies of less than 30 microseconds on typical 800-megahertz (MHz) PC platforms.

RTSS Scheduler

The RTSS scheduler implements a priority based preemptive policy with priority promotion to prevent priority inversion. The RTSS environment provides for 128 priority levels, numbered from 0 to 127, with 0 the lowest priority. The RTSS scheduler will always run the highest priority thread that is ready to run (in the case of multiple ready threads with the same priority, the thread which has been ready the longest will run first). An RTSS thread will run until a higher priority ready thread preempts it, or until it voluntarily relinquishes the processor by waiting, or the time quantum (default is infinite) specified for the thread has expired and another thread at the same priority is ready.

The scheduler has been coded with the requirements of real-time processing in mind. Most importantly, its operation is low latency, and is unaffected by the number of threads it is managing. Each priority has its own ready queue, maintained as a doubly linked list. This allows the execution time of insertion (at the end of the list) and removal (from anywhere in the list) to be independent of the number of threads on the list. A bit array keeps track of which lists are nonempty, and manipulating this bit array is done by high-speed, assembly-coded routines.

While an RTSS thread is running, all Windows XP-managed interrupts, in addition to any interrupts managed by threads of a lower priority than the current thread, are masked out. Conversely, all interrupts managed by higher priority threads are unmasked, allowing for a higher-priority thread to preempt the current thread. In addition to these device interrupts, other mechanisms that can cause the currently running thread to be preempted are the expiration of a timer that causes a higher

priority thread to become ready, or the signaling of a synchronization object (by the currently running thread) for which a higher priority thread is waiting.

Win32-RTSS IPC

Inter-environment IPC is a key feature of RTX, allowing tightly integrated applications where hard real-time processes run in the more resource-intensive RTSS environment. The remainder of the application runs in the Win32 subsystem. This section describes the IPC design.

RTSS proxy model

IPC, as other Windows XP-RTSS communication, uses the SRI channel. Given that the SRI channel prevents Windows XP threads from queuing directly for RTX objects, RTX uses proxy processes and threads to support blocking IPC from Win32. When a Win32 thread accesses an RTX object, RTSS uses a proxy thread on its behalf. This model is clean and economical, its advantages being:

No state-keeping on the Windows XP side for blocking IPC requests.
No special-casing in RTSS for external Win32 wait requests.
Handle and object cleanup for Win32 process and thread termination is handled automatically by RTSS proxy process/thread cleanup.

Fast Timer Support

On all PC platforms, real-time HAL provides clock resolution of 1µsec or better, and timer period of 100µsec or better. If no RTSS applications are executing, there are no timing differences between systems with RTX installed and systems without RTX installed.

Dynamically Linked Libraries

RTSS supports Win32 DLL API (LoadLibrary, GetProcAddress). Currently, all the static and global data in an RTSS DLL is shared between all RTSS processes attached to that DLL.

3.3.2 TenAsys INtime

INtime is implemented as a real-time operating system which shares the hardware platform with the Windows operating system. Components installed on Windows include a Windows kernel driver and a number of Windows services. The driver manages memory for the kernel and real-time applications to run in, and manages the communications interface between the two systems (Refer fig 3 and fig 4).

A Windows service loads the real-time kernel into the allocated memory and then causes a context switch from Windows to the INtime kernel. A low-priority thread is created whose function is to switch the machine context back to the Windows system. Because this thread will be preempted by all other real-time threads, as far

as the real-time kernel is concerned, the Windows system is the real-time system's idle task.

When in the INtime context, any real-time interrupts are handled directly, and all other interrupts are masked at the interrupt controller. When the context returns to Windows, these interrupts are unmasked to allow normal processing by Windows. When a real-time interrupt occurs in the Windows context, the Windows IDT is patched so as to cause a context switch to the INtime context so the interrupts may be handled.

The system timer is usually shared between Windows and INtime. The hardware timer is reprogrammed to interrupt at a higher rate than Windows requires, and the Windows handler is only invoked as required. Typically, the INtime kernel takes timer interrupts at a much higher rate (a period as low as 100µs is possible) to allow for finer granularity timing services. Optimization of the timer handling avoids unnecessary context switches between Windows and INtime.

The INtime kernel provides operating system services for the real-time virtual machine. It provides the real-time services expected of such a kernel, consisting of a priority-based preemptive scheduler with priority-based interrupt handling. Thread priorities range from 0 (highest priority) to 255 (lowest priority) and threads below a configurable priority threshold can time-slice with threads of equal priority. The kernel scheduler has been optimized for maximum interrupt performance.

The Application Environment

The INtime applications environment was designed with the following features:

- Each process has a flat, virtual address space which is isolated from other processes' address space. The code in a process is executed in user mode (IA privilege level 3), so that accidental access to system objects and data is prevented.
- Each process has direct access to the processor's input/output (I/O) space, and it is possible to gain direct access to physical memory.
- Interrupts may be handled directly in any process, by means of system calls and user-written code for the interrupt handler and thread.
- Faults are handled on a per-process basis. The faulting thread is suspended by default, and a record is written to a global mailbox. The default action may be changed so that either the static system debugger or the application debugger can be entered at the faulting address. Processes may also choose to be notified of faults in that process.
- Multiple APIs are provided to make system services available to processes. These are described in detail in the following sections.
- Windows services (access to the registry, event log, and file system) are provided to real-time applications by forwarding the requests to a Windows service to perform the requested action.

Real-Time API

The standard API provided by the INtime kernel provides objects suitable for synchronization and communication, along with the means to share memory

between processes, including between INtime and Windows processes. All objects are referenced by a handle, and all handles are global to the real-time system. The real-time API provides parameter checking and in-line status reporting.

C Library and C++ Support

INtime provides an American National Standard Institute (ANSI-compatible) C library, and support for the EC++ (Embedded C++) ANSI standard, with the Standard Template Library (STL).

Other Services

A Transmission Control Protocol/Internet Protocol (TCP/IP) stack is provided which is derived from the BSD Net3 architecture. Ethernet drivers are provided for Intel, 3Com, and Realtek interface adapters, as well as for NE2000-compatible ISA interfaces. Serial links are supported through Serial Line Internet Protocol (SLIP).

USB support is provided as a development kit for developing USB client software. Standard drivers are provided for Universal Host Controller (UHCI), Open Host Controller (OHCI), and Enhanced Host Controller (EHCI) interfaces.

Other drivers are provided for a range of standard and industrial interfaces.

Reliability Features

INtime features that enhance reliability include exception handling for non-catastrophic system faults, and the Distributed System Manager (DSM) which can can monitor both INtime and Windows process, and detect and assist when the Windows operating system fails.

Exception Handling

One of the primary concerns with the design of the INtime kernel was to provide a non-catastrophic means of handling system exception faults (general protection fault, page fault, stack fault, and so on) and cause as little harm as possible to the real-time kernel and to the Windows system. Accordingly the real-time system has a default exception handler which writes a record to a global mailbox detailing the faulting thread and its circumstances, and then suspends the faulting thread. The mailbox may be monitored from the INtime or the Windows systems, and there are tools for extracting information from the fault record, including register dump, stack tracing, and relevant debug information.

Distributed System Manager

The Distributed System Manager ("DSM") is a co-operative multi-process application which manages an entire INtime system. Any process (Windows or INtime) may choose to monitor any other participating process or be monitored by other participating processes. Notification messages are sent to a monitoring process when a monitored process is deleted, or the communications path to that process is disrupted, or if the Windows system or INtime system is disrupted.

This service may be used to provide system-level integrity.

Windows Stop Processing

An important benefit of the DSM is that it can detect when the Windows platform fails. In this case, the DSM software in the INtime kernel notifies all participating processes of the failure of Windows, and additionally suspends the thread responsible for scheduling Windows. At this point the INtime kernel can continue to execute its processes indefinitely and those processes can choose to take special action in this case, such as bringing peripherals to a known state. At some later point, a real-time process could choose to resume the Windows thread and allow Windows to continue to shut down and reset the machine.

Scalability Features

The INtime operating system is capable of supporting large, complex applications. On the shared architecture as described, it has been found that a typical platform will allow the INtime kernel to consume 50 percent to 60 percent of the CPU bandwidth before the Windows graphical user interface (GUI) and other services begin to be affected adversely. An advantage of the independent kernel in INtime is that it can be relocated to another CPU or another platform, allowing the application to scale as its demands on the platform increase. The interface paradigm changes from the virtual machine architecture to shared memory and then physical communications links such as Ethernet or serial line, but the software interface between the Windows and real-time applications does not change. This allows the same binaries to be used no matter what the hardware architecture chosen.

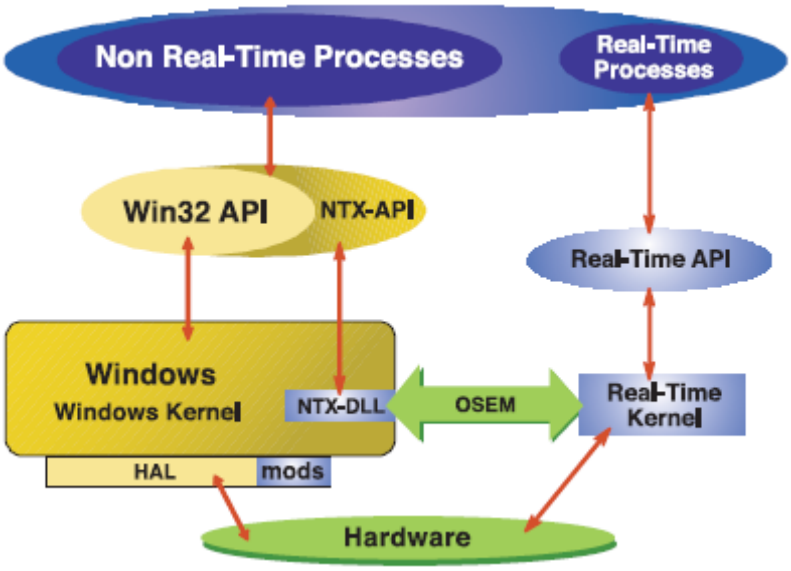


Fig 3
TenAsys Intime Bridge between Real-time Kernel and Windows Kernel

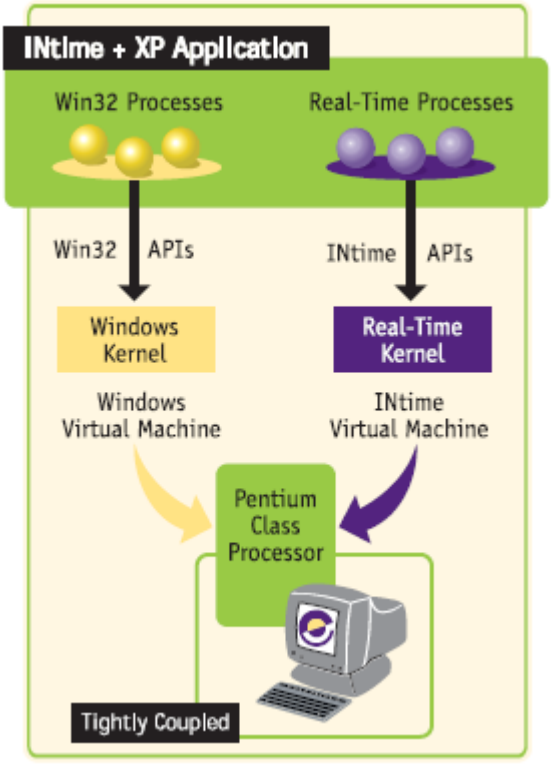


Fig 4
TenAsys Intime Architecture

3.3.3 Nematron HyperKernel

HyperKernel is a self-contained execution environment. That means that HyperKernel threads run within the HyperKernel environment and are not subject to Windows NT/2000 scheduling delays. In reality, the Windows NT/2000 scheduler is not even aware of HyperKernel threads. HyperKernel has its own scheduler, its own set of services, and its own internal kernel (Refer fig 5).

HyperKernel uses a method that runs the two operating systems in parallel - standard Windows XP and the HyperKernel Real-time Subsystem (RTSS). Each executes in its own memory space. A layer is inserted underneath the HAL to handle interrupt revectoring and fast timer controls. HyperKernel will survive "blue screen" STOP conditions and run indefinitely, allowing for an orderly recovery of the system or transfer of user interface functions to a separate computer on a network.

HyperKernel uses a priority-driven preemptive scheduler to control which threads are executing. There are 32 levels of priorities (0=lowest, 31=highest). The scheduler itself uses a simple rule to decide which thread is executed; the thread with the highest priority that's ready to run will be executed. If multiple threads have the same priority, a round-robin method is used to execute them.

HyperKernel does not require any modifications to the Windows NT/2000 HAL. All Windows XP files remain intact after a HyperKernel installation enabling the user to

easily maintain the Windows NT/2000 system without disrupting the HyperKernel Real-Time Environment. The HyperKernel API is structured the same way as the Win32 API and exists as a set of functions for the developer to use with the standard Microsoft Visual C compiler. Special real-time functions are treated as extensions to the Win32 API but follow the same semantics.

HyperKernel doesn't require any special development tools. HyperKernel applications are developed using the industry-leading Microsoft Visual C/C++™ compiler. HyperKernel is a self-contained execution environment. That means that HyperKernel threads run within the HyperKernel environment and are not subject to Windows NT/2000 scheduling delays. In reality, the Windows NT/2000 scheduler is not even aware of HyperKernel threads. HyperKernel has its own scheduler, its own set of services, and its own internal kernel.

Using HyperKernel's setup program, the application developer can control how much memory is used by HyperKernel. The minimum amount is 1 MB. The upper bound is limited to the capacity of the hardware less 8 MB for Windows NT/2000.
Typical context switching time for HyperKernel threads is 3-4 microseconds. As long as interrupts are enabled, the typical interrupt latency is about 5 microseconds.

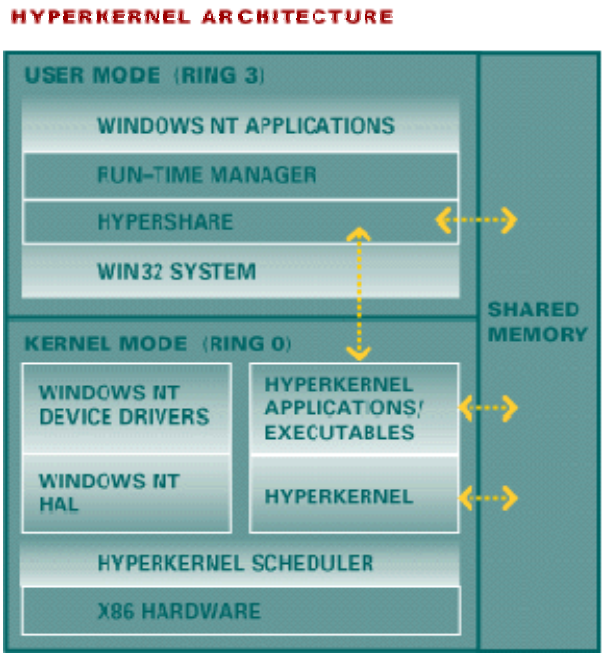


Fig 5
Nematron HyperKernel Architecture

4. Real-time Linux

The real-time Linux scheduler treats the Linux operating system kernel as the idle task. Linux only executes when there are no real time tasks to run, and the real time kernel is inactive. The Linux task can never block interrupts or prevent itself from being preempted. The mechanism that makes this possible is the software emulation of interrupt control hardware. When any code in Linux tries to disable interrupts, the real time system intercepts the request, records it, and returns it to Linux. In fact, Linux is not permitted to ever really disable hardware interrupts, and hence, regardless of the state of Linux, it cannot add latency to the interrupt response time of the real time system. When an interrupt occurs, the real time kernel intercepts the interrupt and decides what to dispatch. If there is a real time handler for the interrupt, the appropriate handler is invoked. If there is no real time interrupt handler, or if the handler indicates that it wants to share the interrupt with Linux, then the interrupt is marked as pending. If Linux has requested that interrupts be enabled, any pending interrupts are enabled, and the appropriate Linux interrupt handler invoked - with hardware interrupts re-enabled. Regardless of the state of Linux: running in kernel mode; running a user process; disabling or enabling interrupts; the real-time system is always able to respond to an interrupt. Real-time Linux decouples the mechanisms of the real time kernel from the mechanisms of the general purpose Linux kernel so that each can be optimized independently and so that the real-time kernel can be kept small and simple. From a maintenance perspective, this de-coupling allows the Real-Time Linux kernel to be easily and quickly adapted to follow changes in the mainstream Linux kernel.

Real-time Linux has been designed so that the real time kernel never waits for the Linux side to release any resources. The real time kernel does not directly request memory, share spin locks, or synchronize data structures, except in tightly controlled situations. For example, the communication links that are used to transfer data between real time tasks and Linux processes are non-blocking on the real time side. There is never a case where the real time task waits to queue or dequeue data. One of the fundamental design philosophies of Real-time Linux is to let the Linux operating system do as much as is practicable. Typical examples include system and device initialization, and blocking dynamic resource allocation. Any thread of execution that can be blocked when there are no available resources cannot have real time constraints. Real-time Linux relies on the Linux loadable module mechanism to install components of the Real-Time system, which keeps it extensible and modular. Loading a Real-Time module is not a real-time operation, and so Linux can do it. The primary function of the Real-Time kernel is to provide direct access to the raw hardware for real time tasks so that they can execute with minimal latency and maximal processing resource, when required.

4.1 RTLinux and RTAI

There are two primary variants of hard real-time Linux available: RTLinux and RTAI. RTLinux was developed at the New Mexico Institute of Technology by Michael Barabanov under the direction of Professor Victor Yodaiken. Real-Time Application Interface (RTAI) was developed at the Dipartimento di Ingeneria Aerospaziale, Politecnico di Milano by Professor Paolo Mantegazza. Under both RTLinux and

RTAI [1], all interrupts are initially handled by the real time kernel and are passed to Linux only when there are no active real time tasks. Changes to the Linux kernel are minimized by providing the kernel with a software emulation of the interrupt control hardware (Refer fig 6). Thus, when Linux has disabled interrupts, the emulation software will queue interrupts that have been passed on by the real time kernel. This is achieved by installing a layer of emulation software between the Linux kernel and the interrupt controller hardware, and replacing all occurrences of cli, sti, and iret in the Linux source code with emulating macros. When the Linux kernel would normally disable interrupts, this event is logged by the emulation software, but interrupts are not actually disabled. When an interrupt occurs, the emulation software checks to see whether Linux has interrupts enabled, if so the interrupt is delivered to the Linux kernel. If not, the interrupt is held pending the Linux kernel re-enabling interrupts. In this way, the Linux kernel does not have direct control over interrupts, and cannot delay the processing of real time interrupts, as these interrupts do not pass through the emulation software. Instead, they are delivered direct to the real time kernel. This also means that the scheduling of real time tasks cannot be delayed by Linux.

Fundamentally, RTAI, RTLinux and applications written to take advantage of them operate in the same way. The Real Time kernel, all their component parts, and the real time application are all run in Linux kernel address space as kernel modules. As each kernel module is loaded it initializes itself ready for system operation. The kernel modules can be removed from the kernel on completion of the real time system operation. Kernel modules can be loaded and unloaded dynamically, either by an application or by taking advantage of the automatic module loading features of Linux itself. The advantages of running the real time system in Linux kernel address space is the task switch time for the real time tasks is minimized, and Translation Look-aside Buffer (TLB) invalidation is kept to a minimum as are protection level changes. Another advantage of making use of kernel loadable modules is that it aids system modularity. For example, if the scheduler is unsuitable for a particular application, then the scheduler module can be replaced by one that meets the needs of the application. One of the main disadvantages of running in Linux kernel address space is that a bug in a real time task can crash the whole system, as there is no separate protected memory space for an individual RT task.

4.2 Real Time Application Interface (RTAI 3.0x)

RTAI [1] provides guaranteed hard real-time and supports several architectures:

- x86 (with and without FPU and TSC)
- PowerPC
- ARM (StrongARM; ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x)
- MIPS
- CRIS (port is under construction - ask on mailing list for details)

Its salient features are (Refer fig 6):

- Linux 2.4 and 2.6 kernels are supported.
- RTAI provides a worst case Context Switch Time of 4.0 - 6.5 µs.
- Priority based Preemptive FIFO scheduling policy is used.
- LXRT support which can be used to provide real-time support in user mode.
- It also provides simultaneous one-shot and periodic schedulers, both inter-Linux and intra-Linux shared memory, POSIX compatibility, inter-task synchronization, semaphores, mutexes, message queues, RPCs, mailboxes and, the ability to use RTAI system calls from within standard user space.

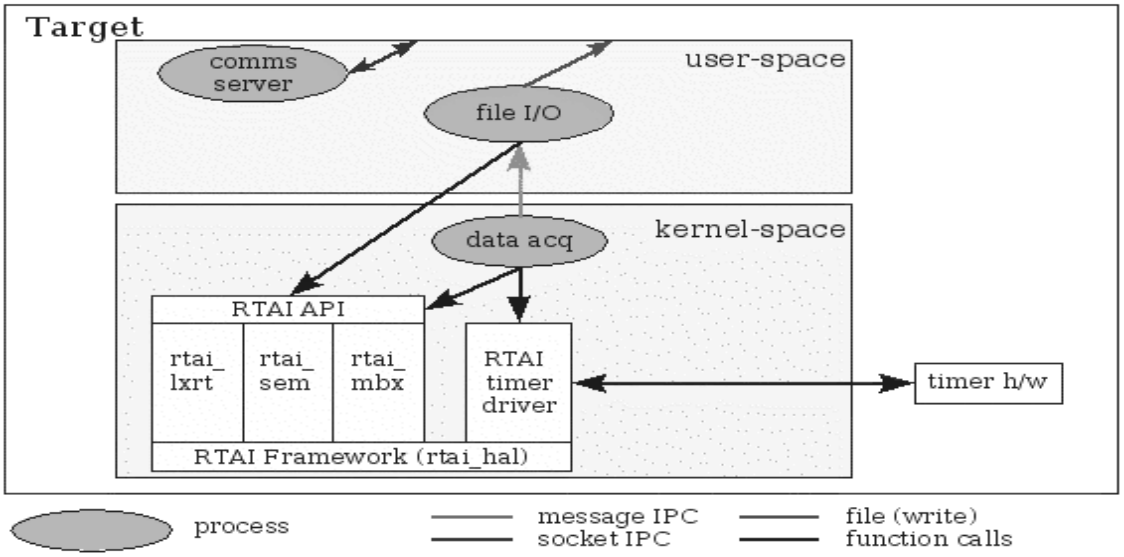


Fig 6
Architecture of RTAI

5. Ports Interfaced

The following ports have been interfaced (in Hard Real-time) by us:

5.1 Parallel Port

PC parallel port is a very useful I/O channel for connecting your own circuits to PC [6]. The PC's parallel port can be used to perform some very amusing hardware interfacing experiments. The port is very easy to use when some basic tricks are known. This part of the report deals with those tricks in easy to understand way.

PC parallel port is a 25 pin D-shaped female connector in the back of the computer (Refer fig 7 and fig 8). It is normally used for connecting a computer to a printer, but many other types of hardware for that port is available today. Not all 25 are needed always. Usually only 8 output pins (data lines) and signal ground are used. The pins are shown in the table below.

Pin	Function
2	D0
3	D1
4	D2
5	D3
6	D4
7	D5
8	D6
9	D7

Pins 18,19,20,21,22,23,24 and 25 are all ground pins. To make a "null-modem" or loop-back connection for the parallel port any output pin (pin 2-9) is connected with the IRQ pin (pin 10 = ACK).

Fig 7
D-25 Parallel Port Connector (Male)

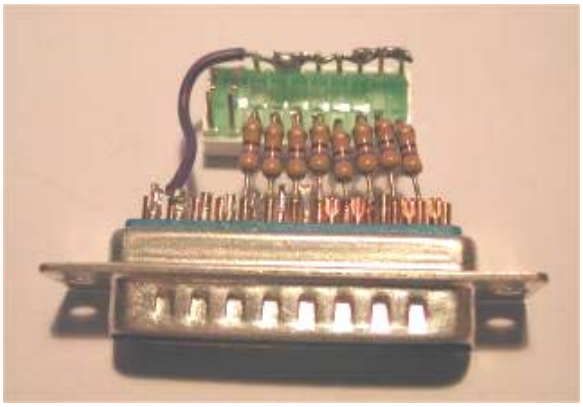


Fig 8
D-25 Parallel Port Connector (Female)

The data output pins (pins 2-9) sink 24 mA, source 15 mA, and their high-level output is minimum 2.4 V. The low state for both is maximum 0.5 V. Pins 1, 14, 16, and 17 (the control outputs) have open collector drivers pulled to 5 V through 4.7 kilo ohm resistors (sink 20 mA, source 0.55 mA, high-level output 5.0 V minus pull-up). Generally, the parallel ports follow this standard.

5.2 Serial Port

The Serial Port is harder to interface than the Parallel Port [7]. In most cases, any device connected to the serial port (Refer fig 9, fig 10, table I, table II, and table III) will need the serial transmission converted back to parallel so that it can be used. This can be done using a UART. Using serial data transfer offers following advantages compared to parallel:

- Serial Cables can be longer than Parallel cables. The serial port transmits a '1' as -3 to -25 volts and a '0' as +3 to +25 volts where as a parallel port transmits a '0' as 0v and a '1' as 5v. Therefore the serial port can have a maximum swing of 50V compared to the parallel port which has a maximum swing of 5 Volts. Therefore cable loss is not going to be as much of a problem for serial cables than they are for parallel.

- There is no need for as many wires as in parallel transmission. If the device needs to be mounted a far distance away from the computer then 3 core cable (Null Modem Configuration) is going to be a lot cheaper that running 19 or 25 core cable. However the cost of the interfacing at each end must be taken into account.

- Infra Red devices have proven quite popular recently. There are many electronic diaries and palmtop computers which have infra red capabilities build in. However its hard to imagine transmitting 8 bits of data at the one time across the room and being able to (from the devices point of view) decipher which bits are which? Therefore serial transmission is used where one bit is sent at a time. IrDA-1 (The first infra red specifications) was capable of 115.2k baud and was interfaced into a UART. The pulse length however was cut down to 3/16th of a

RS232 bit length to conserve power considering these devices are mainly used on diaries, laptops and palmtops.

- Microcontrollers have also proven to be quite popular recently. Many of these have in built SCI (Serial Communications Interfaces) which can be used to talk to the outside world. Serial Communication reduces the pin count of these MPUs. Only two pins are commonly used, Transmit Data (TxD) and Receive Data (RxD) compared with at least 8 pins if you use 8 bit Parallel method (a Strobe may be required).

5.2.1 Hardware Properties

Devices which use serial cables for their communication are split into two categories. These are DCE (Data Communications Equipment) and DTE (Data Terminal Equipment.) Data Communications Equipment are devices such as modem, TA adapter, plotter etc while Data Terminal Equipment is a Computer or Terminal.

The electrical specifications of the serial port are contained in the EIA (Electronics Industry Association) RS232C standard. It states many parameters such as:

- A "Space" (logic 0) will be between +3 and +25 Volts.
- A "Mark" (Logic 1) will be between -3 and -25 Volts.
- The region between +3 and -3 volts is undefined.
- An open circuit voltage should never exceed 25 volts. (In Reference to GND)
- A short circuit current should not exceed 500mA. The driver should be able to handle this without damage.

Above is no where near a complete list of the EIA standard. Line Capacitance, Maximum Baud Rates etc are also included. For more information please consult the EIA RS232-C standard. However, the RS232C standard specifies a maximum baud rate of 20,000 BPS!, which is rather slow by today's standards. A new standard, RS-232D has been recently released.
Serial Ports come in two "sizes", There are the D-Type 25 pin connector and the D-Type 9 pin connector both of which are male on the back of the PC, thus a female connector on the device is required. Below is a table of pin connections for the 9 pin and 25 pin D-Type connectors.

Table I
Serial Pinouts (D25 and D9 Connectors)

D-Type-25 Pin No.	D-Type-9 Pin No.	Abbreviation	Full Name
Pin 2	Pin 3	TD	Transmit Data
Pin 3	Pin 2	RD	Receive Data
Pin 4	Pin 7	RTS	Request To Send
Pin 5	Pin 8	CTS	Clear To Send
Pin 6	Pin 6	DSR	Data Set Ready
Pin 7	Pin 5	SG	Signal Ground
Pin 8	Pin 1	CD	Carrier Detect
Pin 20	Pin 4	DTR	Data Terminal Ready
Pin 22	Pin 9	RI	Ring Indicator

Table II
Pin Functions

Abbreviation	Full Name	Function
TD	Transmit Data	Serial Data Output (TXD)
RD	Receive Data	Serial Data Input (RXD)
CTS	Clear to Send	This line indicates that the Modem is ready to exchange data.
DCD	Data Carrier Detect	When the modem detects a "Carrier" from the modem at the other end of the phone line, this Line becomes active.
DSR	Data Set Ready	This tells the UART that the modem is ready to establish a link.
DTR	Data Terminal Ready	This is the opposite to DSR. This tells the Modem that the UART is ready to link.
RTS	Request To Send	This line informs the Modem that the UART is ready to exchange data.
RI	Ring Indicator	Goes active when modem detects a ringing signal from the PSTN.

It has the receive and transmit lines connected together, so that anything transmitted out of the Serial Port is immediately received by the same port (loopback).

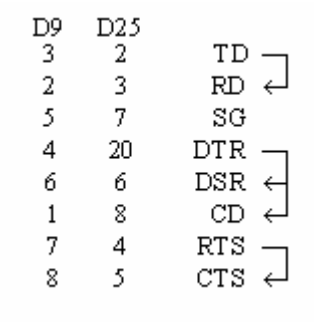


Fig 9
Null-modem connection for serial port

Table III
Port Addresses & IRQs

Name	Address	IRQ
COM 1	3F8	4
COM 2	2F8	3
COM 3	3E8	4
COM 4	2E8	3

Fig 10
D-sub 9 male and female connectors

6. TCP communication

The TCP model is derived from the Open Systems Interconnection Reference Model (OSI model).

6.1 OSI Model

The Open Systems Interconnection Reference Model (OSI Model or OSI Reference Model for short) is a layered abstract description for communications and computer network protocol design, developed as part of the Open Systems Interconnect initiative. It is also called the OSI seven layers model (Refer fig 11).

The OSI model divides the functions of a protocol into a series of layers. Each layer has the property that it only uses the functions of the layer below, and only exports functionality to the layer above. The OSI reference model is a hierarchical structure of seven layers that defines the requirements for communications between two computers. The model was defined by the International Organization for Standardization. It was conceived to allow interoperability across the various platforms offered by vendors. The model allows all network elements to operate together, regardless of who built them. By the late 1970's, ISO was recommending the implementation of the OSI model as a networking standard.

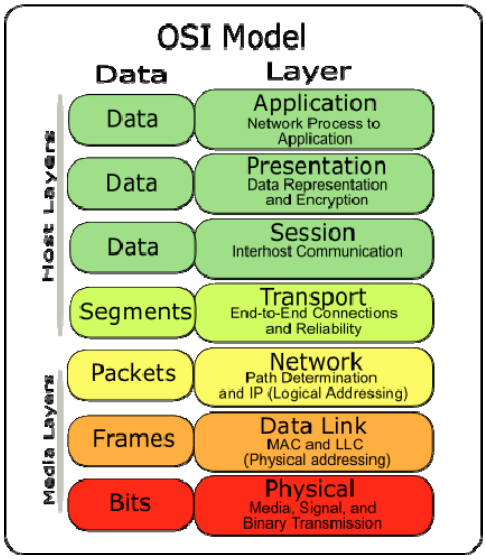


Fig 11
The OSI networking model

6.2 The TCP model

Commonly, the top three layers of the OSI model (Application, Presentation and Session) are considered as a single Application Layer in the TCP/IP suite [9]. Transmission Control Protocol (TCP) is a connection-oriented, reliable-delivery byte-stream transport layer communication protocol (Refer fig 12). TCP connections contain three phases: connection establishment, data transfer and connection termination. A 3-way handshake is used to establish a connection. A four-way

handshake is used to disconnect. During connection establishment, parameters such as sequence numbers are initialized to help ensure ordered delivery and robustness.

TCP uses the notion of port numbers to identify sending and receiving applications. Each side of a TCP connection has an associated 16-bit unsigned port number assigned to the sending or receiving application. Ports are categorized into three basic categories: well known, registered and dynamic/private.

While it is possible for a pair of end hosts to initiate connection between themselves simultaneously, typically one end opens a socket and listens passively for a connection from the other. This is commonly referred to as a passive open, and it designates the server-side of a connection. The client-side of a connection initiates an active open by sending an initial SYN segment to the server as part of the 3-way handshake. The server-side should respond to a valid SYN request with a SYN/ACK. Finally, the client-side should respond to the server with an ACK, completing the 3-way handshake and connection establishment phase.

Through the use of sequence and acknowledgement numbers, TCP can properly deliver received segments in the correct byte stream order to a receiving application. Sequence numbers are 32-bit, unsigned numbers, which wrap to zero on the next byte in the stream after 232-1. One key to maintaining robustness and security for TCP connections is in the selection of the ISN. A 16-bit checksum, consisting of the one's complement of the one's complement sum of the contents of the TCP segment header and data, is computed by a sender, and included in a segment transmission.

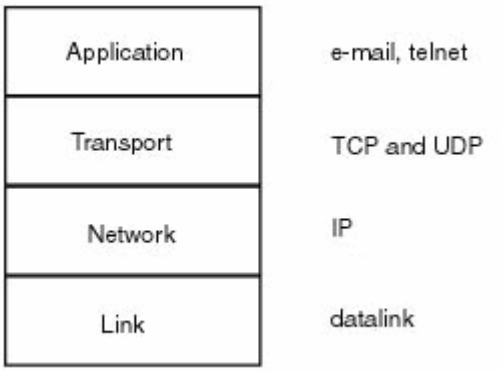


Fig 12
The TCP model

7. Method followed

The following method was used in the implementation of this real-time system:

A stable version (vesuvio) of RTAI (v3.1) on Suse Linux 9.1 (Linux Kernel 2.6.8.1) [2], [3] was used (Refer fig 13). For the ports serial and parallel LXRT with ordinary Linux drivers were used for real-time access [1].

Finally, the data retrieved from any of the ports on the RTAI system was sent to a Windows system over an Ethernet connection using appropriate TCP programs.

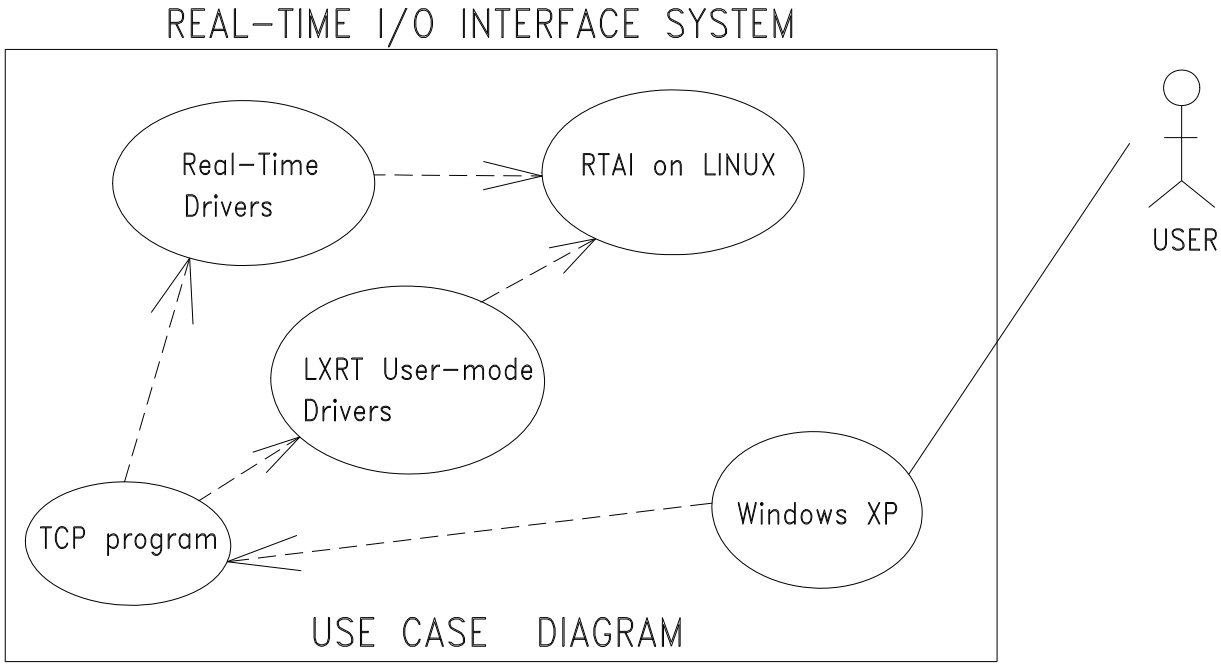


Fig 13
UML – Use Case Diagram of Proposed Real-Time I/O Interface System

The steps followed are as follows [5]:

1. Appropriate version of RTAI was selected (v3.1), and the compatible Linux kernel version (2.6.8.1) was used on Suse Linux 9.1.
2. The Linux distribution was installed.
3. A clean, kosher, version of the Linux kernel was downloaded.
4. The tar file for the RTAI release was downloaded and unpacked.
5. Using the patch file in the RTAI release that corresponds to the downloaded kernel source, the kernel source was patched to incorporate the RTAI modifications.
6. The Linux kernel was built and installed.
7. RTAI was configured, built and installed (to /src/realtime).
8. Program for interfacing parallel port using LXRT was written.
9. Program for interfacing serial port using LXRT was written.
10. Program for interfacing USB port (USB mouse) using LXRT was written.
11. Program for TCP communication between Windows XP and Linux was written in C

12. TCP communication module was integrated with the each port interfacing module

7.1 Steps for installing RTAI 3.1

```
$ cp linux-2.6.8.1.tar /usr/src
$ tar –xvzf rtai-3.1.tar
$ cd /usr/src
$ tar –xvzf linux-2.6.8.1.tar
$ cd linux-2.6.8.1
$ patch –p1 < rtai-3.1/rtai-core/arch/i386/patches/hal7-2.6.8.1.patch
$ make menuconfig
```

remove SMT support, profiling support and auto versioning

```
$ make modules modules_install
$ make install
```

Patched linux kernel is now installed.
Reboot the system to boot from patched linux kernel

```
$ make menuconfig
$ make
$ make install
```

RTAI is now installed. To test it, follow the steps given below

```
$ cd /usr/realtime/testsuite/kern/latency
$ ./run
$ cd ../switch
$ ./run
```

7.2 Hard Real-time performance in user-space using LXRT

The code fragment below illustrates how a user space process can be made hard real-time in kernel space using LXRT [1]. A Hard Real-time Buddy process is created in the kernel space. Then, the program is locked in the memory with no paging possible. Once the user space code has finished executing in Hard real-time, the program is brought back to Soft real-time and the Buddy process is deleted.

```
RT_TASK *handler;
if (!(handler = rt_task_init_schmod(nam2num("HANDLR"), 0, 0, 0, SCHED_FIFO,
0xF)))
{
      printf("CANNOT INIT HANDLER TASK > HANDLR <\n");
      exit(1);
}
rt_allow_nonroot_hrt();
mlockall(MCL_CURRENT | MCL_FUTURE);
rt_make_hard_real_time();
/*
User Space code
*/
rt_make_soft_real_time();
rt_task_delete(handler);
```

The timing calculation for loopback times, context switches and interrupt latencies can be done using the real-time clock of RTAI. *get_cpu_time_ns()* returns the current cpu time in nano seconds. Using this function, the time difference between two events can be calculated.

```
tsr=get_cpu_time_ns();
/*
event
*/
tsr=get-cpu_time_ns()-tsr;
printf (“Time taken by event = %ld”\n,tsr);
```

7.3 Parallel port interfacing in Linux and RTAI

The code fragment below illustrates how to interface a parallel port [6] in Linux and RTAI. First, printer port base address is defined at 0x378 and the port is initialized using *ioperm()*. With a null modem connection, we send data to the port using *outb()* and read back the same data using *inb().*

```
#define base 0x378          /* printer port base address */
#define value 255           /* numeric value to send to printer port */
main(int argc, char **argv)
{
      if (ioperm(base,1,1))
      fprintf(stderr, "Couldn't get the port at %x\n", base), exit(1);
      outb(value, base);
      printf (“%c”, inb(base));
}
```

7.4 Serial Port interfacing in Linux and RTAI

The code fragment below illustrates how to interface a serial port [7] in Linux and RTAI. First, printer port base address is defined at 0x3F8 and the port is initialized using *ioperm()*. With a null modem connection, we send data to the port using *outb()* and read back the same data using *inb().*

```
#define PORT1 0x3F8
void main(void)
{
      int c;
      int ch;
      ioperm (PORT1,6,1);
      outb(value, base);
      while(1) {
            c=inb(PORT1+5);/* Check to see if char has been received.*/
            if(c&1) {
                  ch=inb(PORT1); /* If so, then get Char         */
                  printf (“%c”, ch);
                  break;
            }
      }
}
```

7.5 USB mouse interfacing in Linux and RTAI

The code fragment below illustrates how to interface USB mouse [8] in Linux and RTAI. First, connection to the USB mouse is initiated using the file descriptor and the *open()* function. Data is read from the port using the *read()* function and the connection is closed using the *close()* function.

```
int fd;
fd = open(“/dev/mouse”, O_WRONLY | O_CREAT | O_TRUNC, mode);
bytes_read = read(fd, buf, nbytes);
close(fd);
```

7.6 TCP communication in Linux and Windows

The Socket for TCP communication has to be initialized using the *socket()* function [9]. Next, for the client program a connection is made to the server using the IP of the server and the port selected. Before initializing the client program, the server program should always be run in order to listen to incoming connections from clients. Data is sent using the *send()* function and is received using the *recv()* function. In-built error checking is done by the TCP itself as the sending side always receives confirmation of the number of bytes received successfully by the receiving side. Lastly, the connection is terminated using the *close()* function.

8. Problems faced during RTAI Installation

- Fedora Core 4 did not install on P4-915 motherboard with SATA Hard-drive

- Ubuntu (Debian) and Suse installed on P4-915-SATA

- FC4 installation on PIII-810-Barracuda and P4-865-SATA

- To install RTAI, first Linux Kernel has to be patched. Finding appropriate patches for any Linux Kernel is a difficult task. A Linux kernel which has not been patched with the ADEOS patch may build successfully but will not run properly. Sometimes patches included with the RTAI package (bz2 or gz) do not work properly. Thus proper patching of the Linux kernel is the most important step in the whole process of installing RTAI.

- On Ubuntu Linux

    - RTAI 3.2 on 2.6.10and 2.6.9 failed because of faulty patch
    - RTAI 3.1 worked on 2.6.8.1
    - RTAI/Fusion 0.8.3 on 2.6.12 and 2.6.10 failed most probably because of faulty patch
    - RTAI/Fusion 0.8.1 worked on 2.6.12

- On Ubuntu Linux

    - rt-com (serial driver) failed to compile on RTAI 3.1 on 2.6.8.1
    - usb4rt (real-time usb driver) compiled properly on Fusion 0.8.1 on 2.6.12 but the sample programs based on the driver failed to execute.

- Then suspecting a clash between RTAI installation and the motherboard (915), we shifted to an 865 motherboard

- Suse Linux on 865 motherboard

    - RTAI 3.1 on 2.6.8.1 worked properly
    - RTAI/Fusion 0.8.3 failed 2.6.12 / 2.6.10 failed to run properly due to faulty patches
    - RTAI/Fusion 0.8.1 on 2.6.12 worked properly

- We downloaded around 10-15 programs from www.captain.at. These programs implemented real-time access to parallel port. These programs had been written for various versions of RTAI. We compiled these programs exactly according to instructions given and on the exact version of RTAI specified. But only 3 programs compiled successfully but none ran properly even though we made the proper NULL MODEM connection using D25 connector.

- Finally we wrote a program for non-real-time parallel port access on Linux. Once this program functioned properly we ported it to RTAI using LXRT functions. This program written using LXRT on RTAI 3.1 (Linux Kernel 2.6.8.1) compiled and

ran properly. It is providing real-time access to the parallel port with context switch time on 4-5 micro-seconds and round-trip time of around 24-30 micro-seconds.

- We plan to implement a real-time serial-port access program in a similar manner using LXRT on RTAI 3.1

9. Conclusions

As per the commitments given in the mid-term report, we have implemented hard real-time interface of parallel, serial, and even USB (mouse) ports on RTAI 3.1 (Suse Linux 9.1 on kernel 2.6.8.1). The loop-back times of the parallel, serial, and USB (mouse) ports are around 7-10 µs, 85-95 µs, and 10-15 µs respectively with context switches around 3-7 µs for each port. The above loop-back times are much better than those being achieved currently (around 30 ms) using DIO cards. Even the context switch we have achieved is better than that specified in the terms of reference (equal or less than 1 ms).

Moreover, the TCP communication system for transfer of data from various ports on the Linux machine to the Windows machine has been implemented and integrated with the main modules which interface the ports. It has been observed that a buffer mechanism with an optimal buffer size (depending on the actual application) speeds up the TCP communication at least 2-3 times. But byte-by-byte communication can also be performed if needed. The TCP communication is hard real-time on the Linux side and soft real-time on the Windows side. The speed of this communication can be drastically increased by using more advanced and sophisticated hardware, especially gigabit Ethernet system.

Finally, the next batch can implement real-time interface for the other ports, like audio and fire-wire. Even the USB interface currently implemented can be improved. And apart from TCP communication, another method of communication between Linux and Windows can be implemented using Windows emulators for Linux, like Wine.

Appendix A

SWOT Analysis of the Project

Strength	Development on RTAI Linux
Weakness	No license-free comprehensive solution available
Opportunity	Developing real-time drivers in a multitasking environment
Threat	Development of drivers for all ports and TCP communication with Windows within the given time constraint

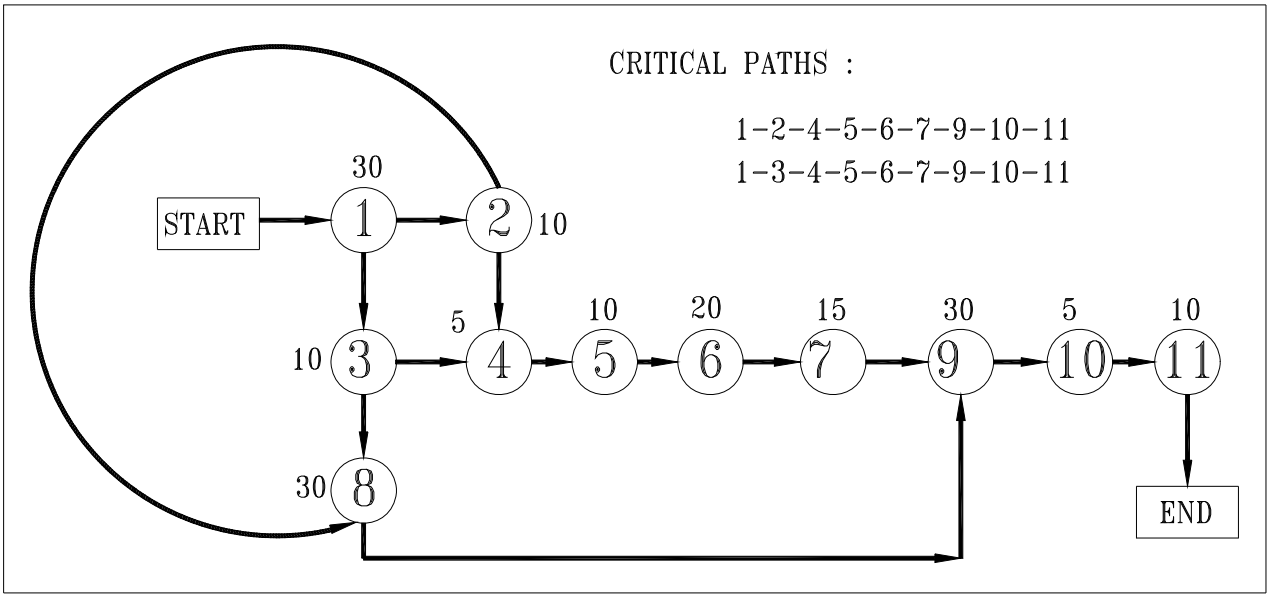
Appendix B

Gantt chart for execution of the project

No.	Activity	Primary Person	Secondary Person	Est. time days	Prede-cessor	Jul	Aug	Sept	Oct	Nov	Dec
1	Design and Analysis	AS	AN, RA	30							
2	Feasibility presentation	AS, AN, RA		10	1						
3	Feasibility report	AS, AN	RA	10	1						
4	Download and Installation	RA	AN, AS	5	2, 3						
5	Configuration	RA	AN, AS	10	4						
6	Driver Installation and configuration	AM	AN, AS	20	5						
7	Testing and tracing	RA	AS	15	6						
8	TCP programming	PR	AN, AS	30	2, 3						
9	Final testing	All		30	7,8						
10	Final presentation	AN, AS	RA	5	9						
11	Final report	AN, AS	RA	10	10						
Legend: AS - Ashish, AN - Anand, RA - Ravinder, PR - Prakash, AM - Amit											

Appendix C

Pert Chart for execution of the project



No.	Activity
1	Design and Analysis
2	Feasibility presentation
3	Feasibility report
4	Download and Installation
5	Configuration
6	Driver Installation and configuration
7	Testing and tracing
8	TCP programming
9	Final testing
10	Final presentation
11	Final report

Appendix D

Interrupts and Potential sources of delay

There are two potential sources of delay in responding to a hardware interrupt. In order to explain the first problem, you need to understand what happens when the Intel processor receives an external interrupt.

- An external interrupt signal is sent to the PC's interrupt controller
- The controller notifies the CPU that an interrupt is pending
- The CPU completes the current instruction
- The CPU saves the state of the current task
- The CPU disables any further interrupts
- The CPU loads ISR information from the interrupt descriptor table
- The CPU executes the ISR
- The ISR enables interrupts (STI instruction)
- The ISR completes
- The CPU restores the state of the previous task
- The CPU resumes execution of the task

There are problems with this methodology:

A poorly written ISR may not re-enable the interrupts (STI) in a timely fashion; this will prevent other interrupts from being received by the processor. Microsoft's response to this problem is to use Deferred Procedure Calls (DPCs). The idea is to process as little as possible in the ISR, signal the DPC, then re-enable interrupts and return from the ISR. The problem is that all the DPCs are collected into a queue with no provisions for priorities. This means that your high priority interrupt gets put into the queue with all the other interrupt DPCs, including mundane events such as interrupts for movement of the mouse.
A task (in Windows NT/2000's case, it would have to be a kernel driver task or the kernel itself) may issue the CLI (Clear Interrupt) instruction to the processor. The CLI instruction disables the CPU from receiving any interrupts until the STI instruction is issued. This prevents the CPU from recognizing the notification from the interrupt controller until the task re-enables interrupts.

A couple of known tasks that cause interrupt delays are the Windows NT/2000 kernel itself and the ATAPI disk driver. The Windows NT/2000 kernel periodically disables interrupts using the CLI instruction. And the ATAPI driver can disable interrupts for extended periods of time. We can't really fault Microsoft too much for that, they had to write a disk driver that was globally compatible with all IDE drives.

The second potential source of delay is the SMI interrupt used on the x86 processors. The SMI is handled completely by the hardware and is invisible to the operating system and applications. The SMI (System Management Interrupt) is a non-maskable interrupt that preempts and disables all other interrupts. The SMI interrupt is used to trigger the processor into SMM (System Management Mode). The exact details of this mode are decided by the motherboard manufacturer and the

BIOS developer, and the exact frequency and duration of the SMI is different for every computer manufacturer.
Given the above information, one can understand why guaranteed interrupt latency is dependent upon a number of open issues, including the exact hardware configuration being used, the device drivers, and your specific application requirements.

References

1) RTAI 3.1 Reference Manual

2) Kernel 2.6.8.1 documentation

3) Ubuntu 5.04 documentation

4) Suse Linux 9.1 documentation

5) www.captain.at

6) Parallel port interfacing made easy: Simple circuits and programs to show how to use PC parallel port output capabilities by Tomi Engdahl

7) Interfacing the Serial / RS232 Port by Craig Peacock

8) Open Group Base Specifications Issue 6 by the IEEE and the Open Group

9) Programming IP Sockets on Linux by David Mertz