UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INGENIERÍA

Departamento de Electrotecnia

SISTEMAS JERÁRQUICOS DE TIEMPO REAL PARA ADQUISICIÓN DE DATOS Y CONTROL

Alejandro Luis VEIGA

Tesis presentada para obtener el grado de

MAGISTER EN INGENIERÍA

Director: Dr. Miguel Mayosky

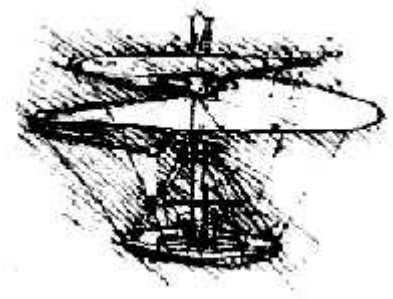
Codirector: Ing. Nolberto Martínez

La Plata, septiembre de 1999

Indice

What's an OS without an "X" on the end of it?

Bill O. Gallmeister, 1995



Introducción

Desde la década del 80 hasta la actualidad los sistemas de cómputo han experimentado una verdadera revolución. Hasta entonces, las computadoras eran costosas y voluminosas, y las organizaciones disponían de sólo unas pocas de ellas, por llegar su costo a los cientos de miles de dólares.

Dos avances tecnológicos fueron los causantes de dicha revolución. El primero de ellos fue el desarrollo de nuevos microprocesadores, que evolucionaron desde las primitivas estructuras de 8 bits hasta las más modernas de 64, con costos cada vez menores. La relación rendimiento/costo de dichos dispositivos se ha incrementado en diez órdenes de magnitud, lo que hace posible disponer actualmente de unidades muy potentes a muy bajo costo. Además, la arquitectura de bus de las computadoras ha evolucionado paralelamente, permitiendo una acorde interacción de dichos microprocesadores con dispositivos externos.

El segundo avance tecnológico determinante fue la aparición de las redes de datos de área local (LAN, Local Area Networks) y de área amplia (WAN, Wide Area Networks). Las primeras permiten interconectar cientos de máquinas a cortas distancias, con velocidades que llegan a los 100Mbits/segundo con tecnología standard. Las segundas permiten la interconexión de LANs a través de largas distancias, con velocidades que van desde 64 Kbits/segundo hasta algunos Gbits/segundo en algunas redes experimentales.

Como resultado de estos avances tecnológicos, es posible disponer en la actualidad de grandes capacidades de cálculo y modernas redes de datos a un costo cada vez menor. Desde el punto de vista de la aplicación en automatización, y especialmente para los sistemas de adquisición de datos y control, este panorama se presenta como muy alentador. La disponibilidad de grandes potencias de cómputo y conectividad, montadas en compactas estructuras de buses, permiten disponer de unidades dedicadas muy poderosas para tareas específicas (*embedded systems*). Este aspecto ha revolucionado el mercado de la automatización industrial, con la aparición de tecnologías distribuidas, que se encuentran constantemente en desarrollo.

Sin embargo, la evolución del hardware es inútil si no se ve respaldada por una adecuada evolución del software que lo acompaña. A pesar del vertiginoso avance de las tecnologías de hardware, el proceso de automatización o 'informatización' de una planta no es tan simple como debería. Esto se debe fundamentalmente a que se encuentran aún en desarrollo las técnicas de software que sean capaces de aprovechar al máximo los recursos disponibles.

A pesar de disponerse de tecnología de hardware a bajo costo, la tarea de integrar una serie de procesos, ya sean industriales o de laboratorio, suele presentar inconvenientes que se deben principalmente a la

falta de normalización de las interfaces de software de los diferentes componentes. El sistema operativo del sistema de cómputos debería encargarse de dicha normalización, pero esto no es cierto en todos los casos.

Existen algunas características imprescindibles que el sistema operativo debe presentar. Fundamentalmente, debe tratarse de un sistema multitarea. Esto significa que el mismo sistema operativo debe proveer los mecanismos para la ejecución simultanea de dos o más programas de software, sin que el programador deba encargarse de esta tarea tan complicada. Actualmente no tiene sentido imaginar un sistema de control que tenga una computadora dedicada a cada tarea. El próximo capítulo se encarga de profundizar en este tema. Por ahora sólo se comentará lo siguiente. Dentro de las características de los sistemas multitarea, de los cuales existe una gran variedad en el mercado, existe una que es crucial para el tipo de aplicación que se está planteando: si existe una sola CPU y varias tareas a realizar, ¿cuál es la estrategia que debe utilizarse para que dichas tareas compartan la CPU? ¿Debe hacerse un reparto equitativo de los recursos compartidos entre las tareas, o existen algunas tareas más importantes que otras que requieren mayor atención?

Esta es la principal diferencia entre los que pueden llamarse *sistemas operativos probabilísticos* y *sistemas operativos determinísticos*.

Los *sistemas operativos probabilísticos o de propósitos generales*, tales como Windows, UNIX u OS2, se encargan de hacer un reparto equitativo de los recursos. La performance general de las tareas está garantizada a largo plazo, pero los tiempos no están acotados. En general, estos sistemas operativos no disponen de herramientas adecuadas para interactuar cómodamente con el hardware de laboratorio y garantizar estrictamente su funcionamiento. Los paquetes de software de adquisición de datos y control para dichos sistemas son generalmente soluciones propietarias, que requieren que todo el hardware sea del mismo fabricante para funcionar adecuadamente. Un ejemplo es el paquete *LabView* de *National Instruments* para *Windows*, que requiere un driver específico que solo está disponible para los elementos de hardware de dicho fabricante. Si se desea incorporar hardware diseñado específicamente para un experimento, debe solicitarse a National la confección de dicho driver. Este tipo de sistemas es adecuado para un gran número de aplicaciones (procesamiento de textos, bases de datos, redes, etc.) dada su gran versatilidad y disponibilidad de software, pero para las aplicaciones de laboratorio muestran serias deficiencias.

Los *sistemas operativos determinísticos o de tiempo real*, como QNX, Lynx u OS9000, son la segunda opción, de los cuales existe también una gran variedad en el mercado. Su principal característica es que permiten establecer una estructura jerárquica dentro del grupo de tareas a ejecutar simultáneamente. A las tareas de mayor importancia (con más estrictos requerimientos de tiempo) se les dedica mayor cantidad de recursos. Esta estructura es mucho más adecuada para la implementación de sistemas de adquisición de datos y control. Pero, en general, se trata de sistemas cerrados y costosos, cuyo objetivo principal es la optimización de los recursos de hardware, perdiendo versatilidad. Las herramientas de desarrollo para estos sistemas operativos son limitadas y no se dispone de gran variedad de software de aplicación.

Sería deseable entonces disponer de un sistema operativo con la versatilidad de los probabilísticos y la estructura jerárquica de los determinísticos.

Como puede verse, al encararse el diseño de un sistema de adquisición de datos y control para un experimento de laboratorio, la selección del hardware se presenta actualmente como menos restrictiva y con mayores posibilidades de elección que la selección de una estructura de software adecuada.

Marco de trabajo

El Laboratorio de Electrónica del Departamento de Física no es una excepción dentro del panorama anteriormente descripto. En nuestro laboratorio se plantea cada vez más frecuentemente la necesidad de integrar un experimento en un único sistema de adquisición de datos y control, con una única interfaz de usuario. Los experimentos han crecido desordenadamente con el transcurso del tiempo, al ir agregándose equipos con diferentes interfaces y técnicas de comunicación. La necesidad de los usuarios es explotar al máximo el potencial de los equipos disponibles en el laboratorio, siendo los requerimientos casi siempre los mismos: que los datos de un equipo sean visibles en otro, que un equipo sea capaz de comandar a otro equipo o que basándose en el estado de un primer equipo se tomen decisiones sobre un segundo. Se plantea así la necesidad de integrar el experimento bajo el comando de una única entidad encargada de la

automatización, distribución de las tareas y comunicación entre los diferentes elementos del experimento. Fundamentalmente, esta entidad debe proveer una única interfaz de usuario, para simplificar así la operación, como así también conectividad para la operación y acceso a datos remoto. Debe ser además lo suficientemente versátil como para asimilar rápidamente los cambios que constantemente se suceden en los experimentos (sensado de nuevas variables o incorporación de equipo adicional).

Otro aspecto importante de dicha entidad a cargo del experimento es la confiabilidad a largo plazo. En general, los experimentos de laboratorio requieren prolongados períodos de adquisición de datos. La mayoría de los experimentos se desarrollan ininterrumpidamente durante días, e incluso semanas. Si una combinación de hardware/software es responsable de la ejecución, su desempeño a largo plazo debe estar de acuerdo con las necesidades.

Otras características que restringen aun más la selección son las referidas a la disponibilidad de componentes de hardware y de software. El sistema seleccionado debe ser lo suficientemente abierto como para prescindir de componentes propietarios monopolizados por algún fabricante, ya sean de hardware o de software.

La estructura de hardware de una computadora personal con arquitectura Intel (de aquí en más será referida como PC) se presenta como una alternativa muy atractiva para manejar la adquisición de datos y el control de un experimento. Aún para reducidos presupuestos, se dispondrá de una configuración muy potente a bajo costo. La normalización de las interfaces ha llevado a que casi cualquier instrumento de laboratorio pueda ser conectado a una PC, para su comando o adquisición de datos. Las interfaces mas comúnmente utilizadas en el equipamiento de laboratorio son RS-232, GPIB, Ethernet, Centronics o CAMAC; los adaptadores correspondientes para PC pueden ser adquiridos en el mercado local, existiendo una amplia gama de productos de diferentes fabricantes. Además se encuentran disponibles, a muy bajo costo, placas de entrada/salida analógico/digitales para la PC, que permiten un eficiente monitoreo de las variables de interés del sistema a controlar.

Para aprovechar toda la potencialidad de este hardware es necesario contar con un sistema operativo que permita tanto un manejo eficiente de los periféricos que atienden a las distintas variables del experimento (incluyendo operación en tiempo real y adecuadas herramientas para desarrollo de software), como una adecuada interfaz con el usuario y acceso remoto. Ninguno de los sistemas operativos usuales para computadoras personales cumplen con esos requisitos y si bien existen sistemas operativos especialmente diseñados, se trata de sistemas cerrados de escasa portabilidad y alto costo, como se mencionó anteriormente.

En este panorama el sistema operativo Linux aparece como una opción muy atractiva. Se trata de un sistema operativo de propósitos generales de distribución gratuita, tipo UNIX, que tiene disponible el código fuente del sistema completo. Es independiente, ya que no contiene código escrito por los autores originales. Linux puede utilizarse en computadoras con arquitectura x86, Pentium, Alpha, ARM, MIPS, PowerPC y SPARC. Es bien conocida su capacidad para adaptarse a todo tipo de redes, gran variedad de software disponible, distribución gratuita, entorno X completo, y principalmente su altísima compatibilidad con los sistemas UNIX comerciales, a partir de la introducción de los standard POSIX [IEEE]. Este sistema operativo proporciona una plataforma muy adecuada para el desarrollo de software. Asimismo, una gran cantidad de herramientas de programación se encuentra disponible en el dominio público, además del código fuente completo del sistema.

Un aspecto menos explorado de este sistema operativo es su versatilidad para interactuar con el hardware, desde el punto de vista del usuario. La disponibilidad de las fuentes del kernel de estructura modular hace a este sistema muy apropiado para el control y monitoreo de experimentos. Debe destacarse también la cantidad y la calidad de la información y bibliografía disponible, generalmente on-line. Si a esto se le suma la disponibilidad de las más variadas herramientas de programación, altísima conectividad y entorno X, el Linux se presenta prácticamente como la única opción de bajo costo para este tipo de implementaciones. Si bien Linux carece de capacidad de procesamiento en tiempo real, existen actualmente proyectos dedicados a mejorar dichas propiedades en este sistema operativo, que inicialmente fue concebido como un sistema de propósitos generales. La mayoría de estos proyectos, a pesar de encontrarse constantemente en desarrollo, ya se encuentran operativos y confiables, especialmente el proyecto Real-Time Linux.

Esta tesis se encarga de explorar las diferentes posibilidades que el panorama hasta aquí descripto presenta para la implementación de sistemas de adquisición de datos y control. El sistema operativo

Linux se utilizará como la herramienta de implementación. Para ello se estudian los aspectos de Linux relacionados con este tema y se los utiliza en una aplicación específica: la automatización de un espectrómetro Mössbauer. Se seleccionó este experimento por incluir varios aspectos de diseño que normalmente se encuentran en todas las implementaciones que se presentan en nuestro laboratorio. La espectrometría Mössbauer es una de las líneas de trabajo del Departamento de Física de la UNLP, y el Laboratorio de Electrónica ha estado siempre involucrado con las diferentes implementaciones que se han llevado a cabo para distintos experimentos.

En lo que resta de este capítulo se procede con una introducción general a este trabajo de tesis, intentando presentar los aspectos más importantes tenidos en cuenta al procederse con una automatización utilizando el sistema operativo Linux.

Estructura general

En esta tesis se introducen, en primera instancia, los aspectos fundamentales de los sistemas de tiempo real, junto con un análisis de cómo los sistemas UNIX en general, y el Linux en particular, se insertan dentro de los sistemas de adquisición y control para experimentos. Se profundiza luego el estudio de la capacidad del Linux y sus extensiones de tiempo real en el manejo de tareas con estrictos requerimientos en sus tiempos de ejecución. Se presentan algunas mediciones efectuadas que permiten caracterizar la capacidad de respuesta de dicho sistema operativo a interrupciones externas, como así también a eventos sincronizados con el reloj interno de la PC. Se dedica además un capítulo al estudio de algunos aspectos internos del sistema operativo Linux, y especialmente a su capacidad de interactuar con el hardware, a través de sus device drivers. Se presenta también una descripción de las herramientas disponibles para el desarrollo de interfaces gráficas, programas de control y conectividad a través de redes locales e Internet.

Ante ciertas restricciones encontradas en el manejo de tareas de tiempo real por parte del Linux, se presenta una alternativa de hardware/software para el procesamiento estrictamente dependiente del tiempo. Se trata de una placa de interfaz diseñada especialmente, y bautizada *µDAQ*. Está compuesta por hardware dedicado (un microcontrolador autónomo y su correspondiente interfaz al bus) montado sobre una placa AT, más un sistema operativo de tiempo real para administración interna, diseñado a tal efecto. Estas placas tienen asignadas tareas específicas, que son ejecutadas independientemente del sistema operativo Linux, que se encarga de recolectar los datos provenientes de éstas.

Con dicho grupo de herramientas se procedió a la integración total de un experimento. En la *figura 1* se muestra la *estructura general* propuesta para la implementación de una experiencia genérica de laboratorio, utilizando los elementos previamente mencionados.

El *hardware* está compuesto por una PC standard, con las placas para bus AT o PCI necesarias para el monitoreo de variables, comando de instrumentos de laboratorio y la conexión a la red local, como por ejemplo adaptadores GPIB, RS-232, Ethernet y otros.

Nótese que las placas montadas sobre el bus de la PC constituyen una *capa de hardware* que se presenta entre el programa de aplicación y el experimento. Esta capa de hardware debe mantenerse alejada del usuario. Esto significa que los detalles del funcionamiento del hardware deben ser manejados por el sistema operativo, de tal manera que el usuario, o quien se encarga de programar la secuencia de trabajo, no necesite conocer los pormenores de su funcionamiento. Se intenta eliminar la programación en bajo nivel del hardware dentro del programa de aplicación, ya que este procedimiento es generalmente complicado y a la vez riesgoso, por estar manejando recursos propios del sistema operativo.

Para conseguir este funcionamiento es necesario presentar al programa de usuario una adecuada interfaz de software para los dispositivos de hardware. Esta interfaz se encarga de normalizar la interfaz a los dispositivos, simplificando su utilización.

El sistema operativo Linux, como todos los sistemas tipo UNIX, presenta una abstracción del hardware utilizando el sistema de archivos, o *filesystem*. Esto permite que los dispositivos de hardware sean manejados, desde el punto de vista del usuario, como archivos ordinarios, con sus operaciones básicas. Para que esto sea posible, debe proveerse al sistema operativo con los adecuados manejadores de dispositivos o *device drivers* para cada elemento de hardware. Para los dispositivos standard (puerta paralelo, puerta serie, interfaz GPIB) los drivers están disponibles en las distribuciones standard de Linux.

En el caso que se agregue un nuevo componente de hardware no-standard, y si se desea mantener la estructura presentada, es necesario proveer al sistema operativo con el device driver adecuado, que respete la filosofía de diseño, y que permita acceder al hardware a través del filesystem. Tal es el caso de la placa *µDAQ* mencionada anteriormente.

En la *figura 1* puede verse que la forma de acceso por parte del usuario está restringida a la red Ethernet, no siendo esto indispensable. La PC puede disponer de un monitor y un teclado, aunque el diseño está orientado a que los accesos se hagan únicamente a través de la red, utilizando la *capa Ethernet*, disponible en todas las PC standard. Esta filosofía permite aislar la interfaz de usuario del lugar donde se realiza el experimento, lo cual es conveniente en muchos casos.

La capa Ethernet se presenta manejada por una capa de nivel superior, dada por las herramientas TCP/IP, disponibles en Linux. Cualquiera sea el lenguaje de programación utilizado, se dispone en Linux de librerías o utilitarios que permiten al usuario utilizar la red local e Internet sin necesidad de programarla a bajo nivel, utilizando los niveles superiores del modelo de capas OSI (Open Systems Interface) [Tanenbaum, 1992]. Por ese motivo se presentan en la *figura 1* las capas Ethernet y TCP/IP como comunes a los diferentes lenguajes de programación.

En lo que se refiere estrictamente al *software* se presentan en la *figura 1* cuatro niveles diferentes de operación, habiéndose trabajado en cada uno de ellos. A continuación se procede con una breve descripción de éstos.

Bajo nivel: programación de device drivers e interacción directa con el hardware, al nivel del kernel de Linux. Este nivel de programación se utilizó para optimizar el acceso a dispositivos externos de hardware, haciendo uso de la capacidad de Linux de incorporar código de sistema operativo.

Nivel intermedio: utilización del lenguaje de programación C, haciendo una utilización intensiva de los recursos propios del sistema operativo, tales como filesystems, administración de procesos, entorno multiusuario/multitarea, etc. La programación en C proporciona, indudablemente, el mejor aprovechamiento de los recursos de Linux, al ser el lenguaje nativo de UNIX.

Alto nivel: utilización de lenguajes de programación más sofisticados, en general interpretados como Java, Tcl/Tk y otros. La utilización de lenguajes de alto nivel significa una sobrecarga para el sistema, pero provee una vía rápida y confiable para la elaboración de interfaces gráficas de usuario, que de otra forma serían muy complicadas de programar y su desarrollo consumiría gran cantidad de tiempo.

Muy alto nivel: esta técnica se trata de la utilización ordenada de programas de aplicación preexistentes en el sistema operativo, utilizando la capacidad de *shell programming*. Se utilizó fundamentalmente para proveer conectividad al sistema a través de la red local, fuera del espectro de tiempo real.

Esta estructura se propone, en forma general, para una mejor utilización de la gran variedad de recursos y herramientas disponibles en el sistema operativo Linux. El trabajo en *capas* de software y hardware permite utilizar las diferentes técnicas de programación en el entorno en que son más eficientes, resultando en una eficiencia general del sistema. El objetivo propuesto es analizar cada capa y estudiar las diferentes posibilidades que se presentan.

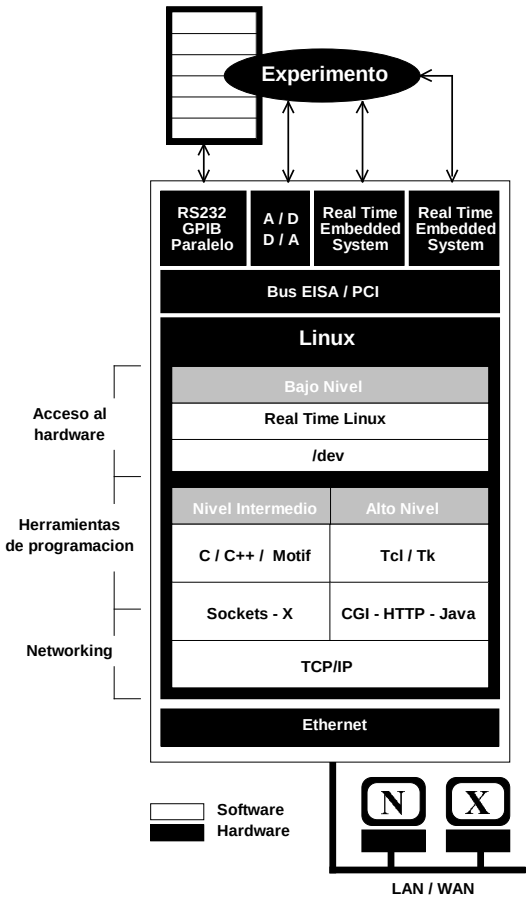


Figura 1: Configuración de hardware y software propuesta para supervisión y control de experimentos

A continuación, a modo de introducción en cada tema, se presentan los aspectos más relevantes de los recursos disponibles para la implementación de dichas capas de software, bajo una plataforma Linux. Los recursos disponibles se clasifican en

- *herramientas de programación disponibles,*
- *conectividad,*
- *capacidad de procesamiento en tiempo real,*
- *interacción con el hardware, e*
- *integración del sistema.*

Se presenta para cada punto un breve resumen del trabajo desarrollado en dicho campo, el cual será extendido a continuación en su capítulo correspondiente.

Los dos primeros puntos (*Herramientas de Programación* y *Conectividad*) son introducidos brevemente a continuación y no se les dedican capítulos específicos de la tesis. Esto se debe a que la bibliografía es abundante en dicho tema y las herramientas disponibles son bastante conocidas y portables. En el capítulo de implementación se hace referencia a dichas herramientas y a la filosofía de diseño utilizada.

Por el contrario, los dos puntos restantes (*Capacidad de Procesamiento en Tiempo real y Acceso al Hardware*, que son introducidos a continuación) son profundizados luego en sus capítulos correspondientes. El diseño e implementación de la placa *μDAQ* ocupa un capítulo aparte.

Herramientas de programación disponibles

La gran variedad de herramientas disponibles para el desarrollo de aplicaciones en el entorno gráfico X, standard en los sistemas tipo UNIX, ha transformado al Linux en una de las más versátiles plataformas de desarrollo de software. Algunos de los paquetes de software han alcanzado un nivel de madurez notable, siendo posible disponer de muy poderosas herramientas en el dominio público. Es el caso, por ejemplo, de los compiladores gcc y g++ del proyecto GNU (ver *Apéndice I*), que van rumbo a convertirse en el standard para dichas plataformas UNIX.

Pueden distinguirse dos grandes grupos de herramientas de programación, las de *nivel intermedio* y las de *alto nivel*, como se mencionó anteriormente. En el grupo de nivel intermedio se incluyen los compiladores para C, C++, Fortran o Pascal, siendo el más utilizado en el caso de UNIX el C compatible con los standard POSIX.1 [IEEE], lo que garantiza la portabilidad entre diferentes plataformas y la explotación a fondo de los recursos multitarea y multiusuario del sistema.

En el grupo de herramientas de programación de alto nivel pueden citarse los lenguajes de programación interpretados, como Java, Tcl/Tk, Phyton y otros. Si bien la performance de los programas escritos en estos lenguajes es menor (debido a la instancia adicional de interpretación necesaria), se obtienen enormes ventajas con respecto a la reducción en la complejidad, lo que permite encarar proyectos con interfaces gráficas muy elaboradas, múltiples hebras de procesamiento y conectividad a redes, sin un consumo excesivo de tiempo durante las etapas de desarrollo y depuración del sistema.

Un criterio para el desarrollo de software muy utilizado actualmente es el de escribir en lenguajes de nivel intermedio o bajo aquellas tareas en las que el tiempo de ejecución y la eficiencia son importantes y reservar los lenguajes de alto nivel para las interfaces gráficas e interacción con el usuario, que en general no tienen restricciones tan estrictas de tiempo. Este es un punto en el cual el sistema operativo Linux sobresale frente a otros sistemas. La gran variedad de lenguajes de programación disponibles, sumado a la capacidad de éstos de comunicarse eficientemente entre sí, permite utilizar cada lenguaje en el área que es más eficiente. De esta forma, puede utilizarse assembler para optimizar los drivers, C para los algoritmos de cálculo y acceso a los recursos del sistema, y Tcl/Tk para la interfaz gráfica con el usuario.

En el caso del lenguaje de programación Tcl/Tk, es posible, a través de un mecanismo simple, la confección de *extensiones*. Se trata de nuevos comandos (escritos y optimizados en C por el usuario y compilados dentro del intérprete) que pueden ser invocados desde los scripts de Tcl/Tk. Esto permite interactuar con el hardware a nivel del C, utilizando funciones desde un lenguaje de alto nivel. Es importante notar que las interfaces gráficas (GUIs) elaboradas para diferentes instrumentos pueden integrarse en un solo panel de control con pocas modificaciones, debido a la simple estructuración de los scripts de Tcl/Tk.

Sin embargo, debe tenerse en cuenta que la utilización de un lenguaje interpretado de alto nivel desmejora la performance respecto de la utilización del C. Sí bien las funciones de Tcl/Tk están implementadas en C y pueden estar tan optimizadas como uno desee, debe tenerse en cuenta que existe una instancia adicional de interpretación del script, que no existe en C. Se trata solamente de un llamado a subrutina adicional, luego de la lectura del comando en el script, pero la diferencia puede ser considerable cuando se trata de una operación repetitiva que se desea ejecutar a alta velocidad (por ejemplo un algoritmo de cálculo).

A modo de aplicación se escribieron las interfaces para algunos instrumentos disponibles. Se utilizó programación en C para la interacción con el hardware y los algoritmos de control o cálculo, y en Tcl/Tk para GUIs, manejo de archivos y acceso a redes. Se implementaron en Tcl/Tk las interfaces para el sintetizador HP3325A, y el multímetro HP3478A utilizando una extensión disponible para manejar el bus GPIB a bajo nivel. Utilizando el driver disponible en Linux para la puerta serie, e interactuando con él a través del C, se escribieron también GUIs en Tcl/Tk para el osciloscopio THS710, y controles de temperatura con interfaz serie diseñados en nuestro laboratorio.

Utilizando estas herramientas se realizó un trabajo de colaboración con el Fermi National Accelerator Laboratory (USA) en el marco del proyecto Auger, en los meses de octubre y noviembre de 1997. En el transcurso de este período se construyó y se puso en marcha un sistema de adquisición de datos integrado, sobre una plataforma Linux. Utilizando una extensión para Tcl/Tk para el acceso al hardware se escribieron GUIs para los osciloscopios HP54111 y HP54522, el frecuencímetro HP5385A y un Crate CAMAC. La interfaz GPIB del Crate de CAMAC permitió la interconexión de varios instrumentos

específicos para experimentos de altas energías, como escalímetros, entradas/salidas digitales, conversores A/D, contadores rápidos y timers. El programa de control del experimento fue escrito en C. Utilizando el driver para la puerta serie se agregaron módulos para una estación de clima y módulos sensores de temperatura 6B, ambos con interfaz RS-232. El sistema de adquisición se encuentra actualmente recolectando datos en los detectores experimentales de Chicago.

Todos los módulos fueron organizados en forma de una *librería* de instrumentos, integrados dentro de un intérprete Tcl/Tk adecuadamente recompilado, lo que permite incorporar o quitar instrumentos al panel de control del experimento sin grandes complicaciones.

En el capítulo de implementación se describe la implementación del intérprete en detalle, para el caso del experimento Mössbauer, junto con el desarrollo de algunos módulos de comando de instrumentos. Se presentan también algunos aspectos considerados en el momento de distribuir las tareas entre los lenguajes de alto y bajo nivel.

La versatilidad conseguida con esta técnica es el aspecto más relevante, ya que se consiguió reducir notablemente el tiempo de desarrollo y puesta a punto del software de aplicación e interfaz gráfica de usuario, sin reducir la performance integral del sistema de hardware/software.

Conectividad

El kernel de Linux ofrece conectividad completa con redes TCP/IP y dispone de drivers para todo tipo de adaptadores Ethernet. La plataforma de hardware Ethernet es la más utilizada para la implementación de LANs, y el protocolo TCP/IP se ha transformado prácticamente en un standard con la explosión de Internet.

Se dispone así de un paquete completo de conectividad, incluyendo servidores para diversos protocolos standard, tales como NFS, SMB, FTP, Telnet y HTTP, junto con sus respectivos clientes. Estos paquetes, sumados a la capacidad multitarea, presentan múltiples ventajas frente a DOS o Windows. Por ejemplo, los datos pueden ser accedidos mientras la adquisición está en proceso, debido a la capacidad de multitarea del sistema, y este acceso puede ser automatizado utilizando el FTP (File Transfer Protocol) nativo de Linux. O puede utilizarse el comando *telnet* para el monitoreo y control remoto de la computadora encargada del experimento. Llamaremos a esta forma de trabajo programación de *muy alto nivel* ya que muchos problemas pueden ser resueltos con una adecuada automatización de estas aplicaciones. Por ejemplo, la nueva especificación del protocolo HTTP contempla la posibilidad de habilitar un cierto grupo de comandos para que sean ejecutados remotamente, utilizando un navegador standard (Netscape, por ejemplo). Esta facilidad se denomina Common Gateway Interface (CGI) y está disponible en todas las distribuciones de Linux.

En este mismo sentido debe mencionarse la capacidad del lenguaje semi-interpretado Java de transmitir programas ejecutables semi-compilados desde el servidor hacia el cliente, y las nuevas especificaciones de HTML, que permiten incluir widgets (botones, paneles indicadores, etc.) en la confección de una página de web. Con estas herramientas, pueden construirse programas de aplicación (que no son más que combinaciones de programas de muy alto nivel) que permiten monitorear y controlar experimentos utilizando, desde el lado del cliente, solo una conexión a Internet y un navegador standard. De esta manera el sistema controlador puede ser solamente una CPU (sin teclado ni display) integrada al resto del instrumental del laboratorio. Esta CPU deberá soportar tanto al programa principal que interacciona directamente con el hardware, como otros programas accesibles por CGI que permiten modificar el estado del programa principal o solicitar el estado de las distintas variables del experimento. Debe destacarse también la capacidad de acceso remoto que provee el entorno gráfico X. Disponiendo de esta facilidad es posible utilizar como display de la interfaz gráfica de nuestro programa de aplicación el monitor de cualquier PC de la red que tenga instalado un cliente de X.

En lo que respecta a la programación en lenguajes de nivel intermedio y alto, se dispone para el enlace a redes de una librería completa de programación. Esta facilidad utiliza la configuración existente del sistema para manejar las capas 1, 2 y 3 del modelo OSI, por lo cual el programador solo debe preocuparse de trabajar en la capa 4, lo cual facilita notablemente la transferencia de datos entre máquinas en una misma LAN, e incluso Internet.

Para el caso de la automatización del experimento Mössbauer se agregó funcionalidad en este aspecto. Además del programa principal de control del experimento, se incorporaron diferentes canales de acceso

a través de la red local, que facilitan el monitoreo y comandado remoto, utilizando herramientas standard disponibles en la mayoría de las PC que integran la red del Departamento de Física. .

Con la configuración adecuada de la red TCP/IP de la PC con Linux encargada del experimento se implementaron accesos remotos de terminal para el comando y monitoreo de dicho experimento desde las PC de escritorio que en general utilizan el sistema operativo Windows en sus diferentes versiones. Además se agregó la posibilidad de extraer los datos resultantes del experimento mientras éste está ejecutándose, lo que permite el procesamiento off-line de éstos, utilizando los protocolos FTP (File Transfer Protocol) y NFS (Network File System).

Utilizando la facilidad CGI descripta anteriormente, se agregó la capacidad de monitorear y comandar el experimento a través de la red por medio de un navegador standard ejecutándose en la PC personal del usuario. La interfaz gráfica se encarga de mostrar en una página de web el valor de las variables principales y de las alarmas, junto con algunos controles, en forma de panel.

Debe recordarse que la interfaz de usuario principal del experimento es ejecutable en el entorno X de UNIX. Esto permite utilizar como display de dicha interfaz a cualquier computadora que disponga del software cliente de X. Este software no es standard en los sistemas tipo Windows, pero puede ser agregado como paquete. En cambio, todos los sistemas tipo UNIX disponen de esta facilidad, incluyendo las workstations de Sun, Hewlett Packard y Silicon Graphics, disponibles en nuestra red.

La variedad y la calidad de la conectividad que pudo conseguirse utilizando el sistema operativo Linux, permiten que pueda eliminarse la consola de la PC que se encarga del control del experimento, conservándose solo los accesos remotos para el monitoreo y control, ya sea a través de la interfaz CGI para los accesos desde Windows, o la interfaz X para los accesos desde UNIX.

Debe aclararse que una adecuada configuración de la PC Linux y de la red local, permiten que los accesos sean no solo desde computadoras que se encuentran en la misma red, sino que además se proporcionó acceso desde la red Internet. Esto permite el monitoreo y comando del experimento desde cualquier lugar del mundo, en caso de ser necesario.

Capacidad de procesamiento en tiempo real

Desde el punto de vista de los sistemas operativos de tiempo real, el Linux presenta serias deficiencias. Su esquema de scheduling es de tiempo compartido, mientras que para realizar tareas con estrictos requerimientos de tiempo es necesario un scheduler preemptivo, basado en prioridades. Otro problema es la falta de *memory lock* y resolución en las funciones de timers. La deshabilitación de las interrupciones por períodos largos de tiempo (con el objeto de distribuir los recursos entre las diferentes tareas) hace que el tiempo de respuesta a las interrupciones sea impredecible. Esto lo descarta para aplicaciones en que dicho tiempo de respuesta es crucial (sistemas de tiempo real *estrictos*). El sistema puede describirse como probabilístico, según la clasificación presentada anteriormente.

Se han planteado algunas soluciones a este problema. El proyecto *Real-Time Linux* [Barabanov, 1997] propone utilizar un pequeño sistema operativo de tiempo real que tiene al Linux como una de sus tareas de baja prioridad. Esta configuración permite la ejecución de tareas en un esquema preemptivo, basado en prioridades, sin que el Linux interfiera, lo que se logra capturando el manejo de interrupciones del sistema operativo. La desventaja de esta forma de trabajo es que para programar aplicaciones en el pequeño kernel de tiempo real no se dispone de los recursos propios del Linux, tales como archivos y acceso a redes. Solo se provee la capacidad de comunicar dichas tareas con el Linux a través de estructuras del tipo FIFO. Esta estructura, sin dudas, dista mucho de ser un sistema operativo de tiempo real, pero en la práctica provee la solución a una gran cantidad problemas frecuentes. En esta configuración no se dispone de archivos ni redes en tiempo real, pero en realidad son muy pocas las aplicaciones en que éstos son necesarios. De lo que no puede prescindirse, en general, es del acceso en tiempo real al hardware de adquisición y control.

Un caso clásico de aplicación es la adquisición de datos con una estricta base de tiempo. Una tarea de tiempo real, con la más alta prioridad, comanda el hardware, adquiriendo los datos. Para proveer el display y archivos, utiliza un FIFO para enviar al Linux los datos adquiridos. El Linux provee una tarea (que no es de tiempo real) que salva en archivos y hace el display, posiblemente en un lenguaje de alto nivel. Esta solución se presenta como muy viable en sistemas de adquisición de datos y control, en los cuales las tareas de tiempo real estrictas son generalmente unas pocas.

El comportamiento de la extensión Real-Time Linux fue encontrado satisfactorio en algunos aspectos, aunque se prefirió utilizar una estructura distribuida para el manejo de grupos de tareas que presentan requerimientos estrictos de tiempo.

Para ello se diseñó la placa *µDAQ* para el bus AT de la PC que contiene el microcontrolador de 8 bits 8051 de Intel. La misión de este dispositivo externo es tener a su cargo la ejecución de ciertas tareas con requerimientos estrictos de tiempo, independientemente del sistema operativo Linux. Para ello se desarrolló, utilizando los conceptos presentados en el *Capítulo I*, el sistema operativo de tiempo real *µKeox*. Se trata de un pequeño kernel operativo que implementa una estrategia de scheduling basada en interrupciones y se ejecuta en forma autónoma en el microcontrolador. La única comunicación con el sistema Linux que existe es en el caso de recibir su configuración inicial y en el momento de reportar los resultados. Este proceso de comunicación se implementa a través de los device drivers de Linux correspondientes.

De esta forma, el sistema operativo Linux delega las tareas con requerimientos más estrictos de tiempo en las placas *µDAQ*, conservando para sí la responsabilidad de monitorear su funcionamiento, retirar periódicamente los resultados, y establecer los canales de comunicación necesarios entre las diferentes placas presentes en el experimento.

Interacción con el hardware

El sistema operativo Linux se ha afianzado en su utilización como herramienta de *escritorio*, más allá de su conocida capacidad para funcionar como *servidor*. Esto es debido fundamentalmente a la amplia gama de software tanto científico como de propósitos generales disponible en el dominio público. Sin embargo, su capacidad para trabajar en bajo nivel no está siendo explotada en su totalidad por parte del usuario medio, a pesar de que, como en todo UNIX, se dispone de una gran variedad de recursos para interactuar con el hardware elegantemente. Las últimas versiones del kernel de Linux incluyen la posibilidad de instalar y remover *módulos* de dicho kernel mientras está siendo ejecutado. Esto hace que el proceso de escribir drivers se haya simplificado notablemente, no siendo ya necesaria la recompilación completa y la reinicialización del sistema cada vez que se desea modificar un driver (proceso muy tedioso). Esto permite al usuario escribir nuevas partes del kernel para agregarle la capacidad de interactuar con nuevo hardware, además de los módulos que el sistema provee para interfaces standard (RS-232, IEEE488, Ethernet, etc.).

La estructura de mapeo del hardware en el kernel a través de sus *devices (/dev)* hace que el acceso a los dispositivos externos sea muy simple desde aplicaciones escritas en C o en algún lenguaje de alto nivel. La programación de los drivers se hace también en C, aunque con un set muy reducido de comandos. A este tipo de programación la llamaremos de *bajo nivel*. El procedimiento para escribir dichos drivers es simple y se encuentra perfectamente documentado, no siendo muy diferente un driver para puerta paralelo de un driver para una placa A/D. Una ventaja de Linux frente a otros sistemas es que el código fuente de todos los drivers del sistema está disponible en el dominio público y puede ser modificado y utilizado como base para nuevos drivers. Una vez incorporada esta forma de trabajo (debe proveerse al Linux de un driver para cada dispositivo que debe manejar, aunque sea con funcionalidad mínima) los programas en C serán capaces de acceder a todo el hardware a través del filesystem como si fueran archivos. Por ello, los programas de aplicación son simples de escribir. Debe aclararse que ésta no es una característica propia del Linux, sino de todos los sistemas operativos tipo UNIX.

En resumen, la integración del hardware de la PC con el sistema operativo fue implementada completamente. Tanto los periféricos standard de la PC, como los diseñados específicamente para el experimento, son accesibles desde la programación de medio y alto nivel a través de mecanismos standard. Esto requirió un esfuerzo adicional en la normalización de todas la interfaces, pero redundó en una mayor flexibilidad en el momento de proceder con la integración de todos los componentes del sistema. En el capítulo de implementación se detalla la utilización de estas interfaces. Puede allí observarse la ventaja de dicha normalización.

Integración del sistema

En el momento de proceder con la integración de todos los elementos descriptos anteriormente fue cuando el sistema operativo Linux mostró sus principales cualidades. La aplicación elegida es una variante del experimento Mössbauer, cuya implementación se describe en detalle en el capítulo de implementación. El punto a destacar es la versatilidad en el desarrollo que proporciona la modularización anteriormente descripta.

Uno de los puntos clave de este desarrollo es que se empleó una considerable cantidad de esfuerzo en la modularización de las interfaces para el hardware. Ningún dispositivo fue manejado directamente utilizando lenguajes de bajo nivel, sino que se escribieron los device drivers correspondientes para cada elemento de hardware incorporado y se reorganizaron los device drivers disponibles para los dispositivos standard de la PC. De esta forma, el monitoreo de las variables del experimento se hace de una forma natural y ordenada, que no requiere de los 'oscuros' procedimientos que en general son necesarios para el manejo de hardware en sistemas operativos poco versátiles.

La programación de los algoritmos de control y la toma de decisiones, programados en C bajo las normas POSIX, permitió una buena explotación de la capacidad de cómputo de la PC con arquitectura Intel, obteniéndose los tiempos de respuesta y performance necesarios sin presentarse la necesidad de disponer de equipos de última generación.

La conexión a la red local e Internet se implementó naturalmente, utilizando los recursos disponibles en el sistema operativo Linux, procediéndose solamente con técnicas de programación de muy alto nivel.

Por último, la construcción de interfaces gráficas muy generosas fue posible gracias a la utilización de lenguajes interpretados que emplean el concepto de módulos gráficos reutilizables muy livianos, que no se presentan como un problema en el momento de convivir con los algoritmos de control y toma de decisiones.

Es importante destacar la variedad de lenguajes de programación utilizados en la resolución del problema, utilizando cada uno en la aplicación para la cual es mas adecuado: *bajo nivel* para los drivers (acceso directo a los registros), *nivel medio* para los algoritmos (programación en C), *nivel alto* para las interfaces gráficas (scripts en Tcl/Tk) y *muy alto nivel* para la conectividad (shell scripting).

La integración de semejante variedad de lenguajes para la resolución de un problema específico (en este caso la automatización de un experimento) sólo es posible en el marco de un sistema operativo con las características del Linux. Este procedimiento de diseño sería inaccesible en sistemas DOS, Windows o similares, o incluso en sistemas cerrados tales como QNX o Lynx, en los cuales tal variedad de herramientas no está disponible.

Puede notarse que el nivel necesario de conocimiento de los lenguajes de programación utilizados no es el de *experto*. Como las diferentes herramientas fueron utilizadas para el propósito que fueron desarrolladas (no se utilizan para hacer cosas *raras*, o trucos de *hackers*), un nivel de dominio moderado fue suficiente para llevar a buen fin la implementación.

Otro aspecto importante es que la modularización conseguida con los componentes de software y de hardware hace que estos sean totalmente reutilizables en otra aplicación, en forma conjunta o por separado. En el caso de plantearse un nuevo problema los componentes deberán reprogramarse, pero la estructura modular se mantiene. Puede ser necesaria la construcción de un nuevo elemento, pero si se mantienen las normas de diseño presentadas (lo cual implica un esfuerzo adicional) el tiempo empleado será recuperado en una futura aplicación del módulo.

Una visión global del sistema implementado permite reconocer que el resultado conseguido puede considerarse como un verdaderosistema dedicado o *embedded system*. Este concepto se refuerza si se imagina la PC con Linux como una placa madre, fuente de alimentación y placas de interfaz, montada en un rack y solo accesible a través de la red de datos. Puede considerarse como un elemento autónomo del experimento, con algunas de las partes de éste a cargo: monitoreo, acceso remoto, etc. Se trata una entrada o *gateway* para el experimento.

Organización de la tesis

Como introducción a la problemática de tiempo real, en el *Capítulo I* se presentan los aspectos más relevantes de los sistemas operativos en general, desde el punto de vista de los sistemas de tiempo real. El propósito principal de este capítulo es la unificación de diferentes conceptos que aparecen diseminados en la bibliografía, con el objeto de presentar los conceptos de diseño que se utilizaron en la implementación del experimento, fundamentalmente en el diseño del kernel de tiempo real para el microcontrolador 8051 de Intel presentado más adelante. Este capítulo presenta los elementos fundamentales de los sistemas operativos de tiempo real y sus principales diferencias con los sistemas operativos de propósitos generales. Se presta especial atención a los avances en el campo del 'scheduling', aspecto fundamental de los sistemas de tiempo real que se encuentra en constante evolución, por tratarse de un campo nuevo de investigación.

Entrando ya en el campo específico de los sistemas operativos, en el *Capítulo II* se presenta una descripción de las características principales de los sistemas UNIX y del Linux en particular, junto con una descripción de las diferentes extensiones de tiempo real para este último.

Continuando el tema del capítulo anterior, en el *Capítulo III* se procede con una descripción detallada del funcionamiento del kernel del Linux POSIX.1 y sus mecanismos de interacción con el hardware. Se detalla además el proceso genérico de creación de un driver POSIX.1 para este sistema operativo. A modo de ejemplo se presenta una modificación del módulo para la puerta paralelo, disponible en Linux, para poder utilizar esta interfaz como entradas y salidas digitales de la PC, como así también permitir la generación de interrupciones externas al equipo. A continuación se profundiza en el funcionamiento del sistema RTLinux en particular, y por último se presentan algunas mediciones realizadas, junto con un análisis comparativo, de la performance de los sistemas Linux POSIX.1 y RTLinux, en lo que respecta a su tiempo de respuesta a interrupciones y generación de tareas sincrónicas

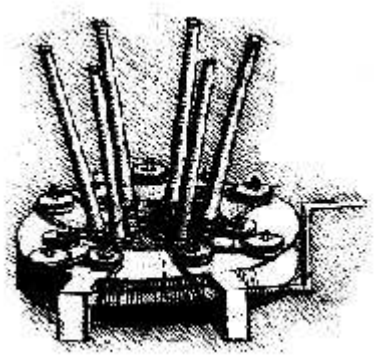
En el *Capítulo IV* se presenta la placa para multiprocesamiento *µDAQ*. Esta placa contiene un microcontrolador 8051 de Intel, con una interfaz de 16 bits para el bus AT de la PC. Las comunicaciones en ambos sentidos están basadas en interrupciones y los recursos no utilizados del microcontrolador están disponibles para diferentes implementaciones. En la primera parte del capítulo se detalla la estructura de hardware y los conceptos de diseño. La segunda parte presenta la implementación del kernel operativo de tiempo real *µKeox*, que se diseñó para la administración de tareas dentro del microcontrolador local de la placa. Por último, la tercera parte presenta el desarrollo del driver para Linux POSIX.1 que le permite establecer una comunicación bidireccional con el sistema operativo, y un ejemplo posible de utilización de la placa en el entorno RTLinux.

Como aplicación de los conceptos presentados en los capítulos anteriores, el *Capítulo V* muestra la utilización de dos placas *µDAQ* para la implementación del experimento Mössbauer, junto con las más relevantes consideraciones de diseño de éste. La primera de ellas tiene asignado el grupo de tareas relacionado con el control de la temperatura de la muestra, y la segunda se utilizó para implementar la recolección de datos. Ambas utilizan *µKeox* para la administración interna y para mantener su respectiva comunicación con el sistema operativo Linux, que se encarga de su monitoreo y organización de los datos, como así también de la interfaz de usuario y la conectividad.

Por último se presenta un capítulo de *Conclusiones* y algunos *Apéndices*.

Es verdad que la eternidad no es concebible, pero el humilde tiempo sucesivo tampoco lo es.

Jorge Luis Borges



Capítulo I
Sistemas de tiempo real

El estudio de los *sistemas de tiempo real* es una rama relativamente nueva de la ingeniería. Los primeros trabajos que mencionan estos términos tal cual se los utiliza en este capítulo son del la década del 80 [Stankovic, 1985, 1987]. Se trata, básicamente, de la reunión de varias técnicas preexistentes de análisis y diseño, con el propósito de organizar el desarrollo de sistemas compuestos por una combinación de hardware y software. El objetivo fundamental es que dicho desarrollo resulte en sistemas robustos y confiables, con un óptimo aprovechamiento de los recursos. Utiliza un enfoque diferente, con los mismos objetivos.

En este capítulo se presenta una introducción a los conceptos fundamentales relacionados con los sistemas de tiempo real, junto con las principales técnicas de diseño. Estas técnicas se aplicarán frecuentemente durante el desarrollo de esta tesis.

Conceptos básicos

En primera instancia se procede a presentar un conjunto de definiciones necesarias para entender el significado de la expresión *sistemas de tiempo real* [Laplante, 1992]:

Un *sistema* es una caja negra que tiene un grupo de una o más entradas y un grupo de una o más salidas.

Este sistema es *determinístico* si para cada estado y cada conjunto de entradas pueden ser determinados un único conjunto de salidas y el próximo estado del sistema.

El tiempo entre la presentación de un conjunto de entradas a un sistema y la aparición de todas las salidas asociadas se llama *tiempo de respuesta* del sistema.

Un sistema en *falla* es un sistema que no puede satisfacer uno o más de los requisitos presentados en la especificación del sistema.

Un *sistema de tiempo real* puede definirse, entonces, como un sistema que debe satisfacer restricciones explícitas en el tiempo de respuesta o arriesgarse a severas consecuencias, incluida la falla. Por lo tanto un sistema de tiempo real es un sistema que responde a un estímulo externo dentro de un tiempo especificado. Su eficiencia no solo depende de la exactitud de los resultados de cómputo, sino también del momento en que los entrega. La predictibilidad es su característica principal. A diferencia de los sistemas

tradicionales, que tienden a distribuir en forma equitativa los recursos disponibles entre las diferentes tareas a ejecutar, los sistemas de tiempo real deben asegurar la distribución de recursos de tal forma que se cumplan los requerimientos de tiempo.

Los sistemas de tiempo real pueden dividirse en dos tipos diferentes, en función de su severidad en el tratamiento de los errores que puedan presentarse:

Sistemas de tiempo real blandos o *Soft real-time systems:* pueden tolerar un exceso en el tiempo de respuesta, con una penalización por el incumplimiento del plazo.

Sistemas de tiempo real duros o *Hard real-time systems:* la respuesta fuera de término no tiene valor alguno, y produce la falla del sistema.

La idea de sistema de tiempo real no debe asociarse únicamente con la velocidad de respuesta del sistema. En cambio, *tiempo real* implica sí necesariamente que los tiempos de respuesta estén acotados. Debe conocerse exactamente el tiempo que le tomará al sistema responder a un determinado evento. Este tiempo debe ser invariable, fundamentalmente, y además debe ser lo suficientemente rápido como para no producir una falla, por supuesto.

Si bien los primeros trabajos en sistemas de tiempo real se enfocaron a arquitecturas simples y dedicadas, los actuales sistemas de computadoras de propósitos generales y sus respectivas aplicaciones se están transformando en sistemas complejos que generalmente requieren los atributos de los sistemas de tiempo real: múltiples tareas coexisten en el sistema y cada una de ellas tiene diferentes requerimientos de tiempo; se requiere de mecanismos eficientes de comunicación entre las tareas; acceso simultáneo a dispositivos y redes; configurabilidad; adaptabilidad; etc.

El objetivo de este capítulo es presentar los aspectos a tener en cuenta en el análisis, tanto de sistemas dedicados como de sistemas de computadoras, bajo requerimientos de tiempo real.

Presentemos algunas definiciones más antes de asociar la anterior definición de sistema de tiempo real con la idea de sistemas de computadoras.

Llamaremos *sistema de computadora* a todo sistema que incluya un microprocesador, tanto una computadora personal, como un sistema cerrado.

Un *sistema operativo* es una colección especializada de programas que provee los mecanismos para interrelacionar los distintas componentes de un sistema de computadora.

Un *embedded system*[1], o sistema dedicado, es un software utilizado para controlar un hardware determinado que es parte de un sistema mayor. Generalmente se trata de sistemas cerrados en los cuales el usuario solo tiene acceso a un conjunto limitado de funciones, o a ninguna. Estos sistemas tienen un conjunto específico de tareas asignadas que no puede modificarse si no es por medio de una reprogramación total. Un ejemplo clásico de un embedded system es el software de un microcontrolador que es parte de una planta, y que está a cargo, por ejemplo, de controlar la temperatura de la misma.

Un *sistema operativo de propósitos generales* difiere de un embedded system en que no tiene asignado un único conjunto de tareas a realizar. Puede ser reprogramado completamente por el usuario. Tal es el caso de una PC con sistema operativo UNIX, con un programa ejecutándose. Este sistema operativo permite ejecutar tareas tan disímiles como un procesador de texto o un control de temperatura.

Debe aclararse que en lo que sigue se hace referencia a sistemas de computadora con un único microprocesador, dejándose las estructuras multiprocesador para un análisis posterior.

Sistemas multitarea

Las aplicaciones de tiempo real deben interaccionar, generalmente, con dispositivos externos tales como sensores y actuadores, además del correspondiente monitor, teclado y disco rígido. Estas interacciones con dispositivos externos tienen la particularidad de que están sucediendo todas simultáneamente. La

[1] La amplia difusión de esta expresión excusa su falta de traducción al castellano.

misión de la aplicación es proveer una respuesta adecuada, a través de sus salidas, a cada una de las entradas, todas al mismo tiempo.

¿Cómo debe entonces estructurarse la aplicación para cumplir con dichos requerimientos? Existen unas pocas alternativas encabezadas por dos estructuras básicas: el *gran loop* o la estructura *multitareas*.

La primera opción maneja todos los eventos secuencialmente, en un orden predeterminado, dentro de una tarea única que se repite cíclicamente. Es la forma más simple de estructurar una aplicación, pero puede complicarse su diseño cuando el número de eventos a manejar es muy elevado. En este caso, el programa de aplicación debe encargarse de recorrer las múltiples tareas. Esto resulta en programas complicados y difíciles de mantener.

La segunda opción tiene, en cambio, al sistema operativo como protagonista. Éste se encarga de emular un entorno de ejecución para diferentes tareas que se ejecutan independientemente una de la otra. Cada tarea dispone de un cierto tiempo de acceso a los recursos, administrado por el sistema operativo. En este tipo de sistemas, el programador escribe las tareas a realizar en programas diferentes, más simples. El sistema operativo es el encargado de hacer que todos estos programas se ejecuten en un único microprocesador. En este tipo de sistemas debe procederse cautelosamente al asignarse el número de tareas, ya que la emulación a cargo del sistema operativo significa una sobrecarga para el sistema de computadora.

En el campo del tiempo real, tanto los sistemas operativos de propósitos generales como los embedded systems, deben proveer la capacidad de realizar múltiples tareas simultáneamente, sin que esto signifique una complicación excesiva para el programador. Este tipo de sistemas operativos se denomina, en forma general, *sistemas operativos multitarea*.

Analizando el problema de ejecutar varias tareas simultáneamente con un poco más de profundidad, puede verse que un sistema operativo multitarea debe proveer al menos tres funciones específicas, relativas a la administración de dichas tareas[2] (además del manejo de memoria y control de dispositivos de hardware, responsabilidad irrenunciable de todos los sistemas operativos, multitarea o no):

Scheduler[3] *de tareas:* es una tarea adicional, de mayor jerarquía que las demás, que se encarga de determinar en qué orden se ejecutará el resto de las tareas presentes en el sistema.

Dispatcher[3] *de tareas:* toma los recaudos necesarios para comenzar una tarea. Esto significa preservar el estado de los recursos que la tarea anterior estuviera utilizando en el momento de comenzar una nueva.

Comunicación: El sistema operativo debe proveer algún mecanismo de comunicación y sincronización entre los diferentes procesos que se están ejecutando simultáneamente.

El *kernel* o *núcleo* del sistema operativo es la porción más pequeña de sistema operativo que provee esas tres funciones.

Cada una de estas tres funciones del kernel multitarea está asociada a un problema diferente. Y dichos problemas tienen diferentes soluciones. Por ejemplo, si se dispone de tres procesos listos para ser ejecutados, ¿en qué orden deben ser ejecutados? ¿Cómo se reparte el tiempo de CPU entre los tres? ¿Se ejecuta primero el más urgente? Si se elige esta última opción ¿cuál es el más urgente? La toma de este tipo de decisiones está a cargo del scheduler del sistema operativo y son muy diversas las estrategias que pueden utilizarse. Más adelante en este mismo capítulo se describen las más importantes.

Pero ese no es el único problema que debe afrontarse al diseñar un sistema operativo multitarea. Si la misma CPU es utilizada por varios programas de usuario diferentes, y suponiendo que dichos programas se ejecutan en el orden indicado por el scheduler, al interrumpirse uno de los programas para ejecutar otro el kernel debe encargarse de guardar toda la información necesaria, tal que el programa desplazado

---

[2] Los términos *proceso* y *tarea* se utilizan indistintamente, sin hacer diferencias entre ellos.

[3] Se utilizarán a partir de aquí tanto el término *scheduler* como el término *dispatcher*, sin traducción, dada la amplia difusión de éstos. Un término equivalente en castellano podría confundir innecesariamente.

pueda ser retomado luego. Esta es la misión del dispatcher. Generalmente éstos se implementan utilizando para cada tarea en ejecución una estructura de datos llamada TCB (Task Control Block). Esta estructura contiene todos los datos asociados a una tarea, que son de relevancia para el sistema operativo, como por ejemplo un número de identificación, el estado de los registros del microprocesador en el momento de ser desplazada, un puntero a la última operación ejecutada del programa, etc.

Por último, otro problema que debe resolverse en un sistema operativo multitarea, es la sincronización y concurrencia. ¿Qué sucedería si dos programas de usuario que se ejecutan compartiendo la CPU a intervalos de tiempo, decidieran acceder a un recurso único de hardware simultáneamente, como por ejemplo una impresora? Para ello, un sistema operativo de este tipo debe proveer los mecanismos necesarios para administrar los recursos compartidos de una forma ordenada. Este tipo de mecanismos es utilizado también para sincronizar tareas entre sí, y se describen con profundidad en este mismo capítulo.

Como puede verse, la complejidad de un sistema operativo multitarea es bastante grande. El kernel debe encargarse de las tareas mencionadas, además de la administración de memoria y los manejadores de dispositivos (drivers). Por lo tanto, la utilización de un sistema tan complejo lleva apareada una sobrecarga del sistema, ya que una buena parte de la CPU debe utilizarse para la toma de decisiones y administración de las múltiples tareas. Pero esto no es una desventaja. Si se utiliza un sistema operativo que no provee estos mecanismos, solo podrá realizarse una tarea con una CPU. Y si en cambio se desean ejecutar varias tareas con éste, el programador de una aplicación se verá en la obligación de implementar por sí mismo los mecanismos de scheduling, dispatching y comunicación, cada vez que escriba una aplicación.

Por lo tanto, si bien los sistemas operativos multitarea son un poco más complicados de utilizar, esto se debe a la gran cantidad de herramientas que presentan al programador. Y en el momento de desarrollar una aplicación elaborada, resultan en definitiva más simples de operar.

A continuación se profundizará en las técnicas scheduling por ser uno de los puntos clave de los sistemas de tiempo real. Luego se mencionan los mecanismos de comunicación más comúnmente utilizados en los sistemas operativos standard. Posteriormente se presentan otros aspectos de importancia a tener en cuenta al analizar las propiedades de un sistema operativo de tiempo real, como relojes y entrada/salida en tiempo real. Para finalizar el capítulo se presenta un resumen de las técnicas utilizadas actualmente para la asignación de prioridades entre las tareas.

Técnicas de scheduling

Los kernel de tiempo real utilizan diferentes estrategias en lo que se refiere a la implementación del scheduler de las tareas que deben realizarse simultáneamente. Estas estrategias varían según la aplicación que se le dé al sistema, la cual puede variar considerablemente en su concepción. Se analizan a continuación las diferentes estrategias que pueden emplearse en el diseño de un kernel de tiempo real.

i. Sistemas cíclicos o Polled loop systems

En este tipo de sistemas, una única instrucción de test repetitiva es utilizada para testear un flag que indica que un evento asociado ha ocurrido.

Este método reduce la complejidad de utilizar múltiples tareas, pero puede complicarse si se trata de manejar un gran número de interacciones simultáneas. El mantenimiento de estos sistemas suele ser complicado, ya que se debe ser muy cauteloso al realizar una modificación.

Existe solo una tarea: la que verifica los flags y si alguno está activo, ejecuta una tarea asociada. Cuando esta tarea finaliza, se retoma la verificación de los flags. Por lo tanto no es necesario un scheduling ni comunicación entre tareas. Simplemente verifica secuencialmente una serie de flags, y si están activados ejecuta las tareas asociadas.

El tiempo de respuesta del kernel a un determinado evento es simple de calcular: debe suponerse el peor caso: cuando se activa un flag, están activos todos los demás y la verificación está haciéndose en el flag inmediato posterior en la secuencia. El tiempo de respuesta ante la presencia del flag correspondiente no es constante.

En realidad esta implementación no es multitarea. Ejecuta sólo una tarea a la vez y para realizar la próxima debe esperar que termine la actual. Esto puede ser una buena solución en sistemas rápidos que realicen tareas simples (sistemas sobredimensionados). En general, pueden obtenerse mejores tiempos de respuesta con las implementaciones que se describen a continuación, en las cuales surge la idea de múltiples tareas ejecutándose simultáneamente.

Debe aclararse nuevamente que estamos hablando acerca de sistemas de computadora que incluyen sólo un microprocesador. Esto quiere decir que sólo una instrucción de código de máquina puede ejecutarse simultáneamente. Al referirnos a múltiples tareas simultáneas, nos referimos a tareas que son interrumpidas para realizar otras tareas, y que luego son retomadas.

ii. Sistema multitarea cooperativo

Esta es una forma de aproximarse a la ejecución simultánea de varias tareas. Se trata de un esquema en el cual dos o más tareas son divididas en estados o fases. Las llamadas al dispatcher central se hacen después que cada fase ha concluido, o sea que una tarea entrega *voluntariamente* los recursos a otra cuando ha concluido una fase. Este procedimiento requiere de una programación muy cautelosa y hay veces que no es simple subdividir las tareas en fases. Este tipo de sistemas es muy sensible a la presencia de aplicaciones "maliciosas" que acaparen los recursos del sistema, sin cooperar[4].

iii. Sistemas basados en interrupciones

Primero un par más de definiciones:

> Un *evento* es un suceso que hace que el program counter cambie no secuencialmente. Se trata de un cambio del control de flujo.
>
> Una *interrupción* es una señal de hardware que inicia un evento.

El trabajo basado en interrupciones es una técnica de implementación de sistemas multitarea en la cual las tareas se ejecutan simultáneamente, administradas por un control central. Cada tarea es una entidad separada e independiente, realizando un trabajo en particular, a su propia frecuencia.

En este tipo de sistemas, el programa principal no ejecuta ninguna tarea, simplemente un loop. El scheduling de las tareas es manejado por medio de interrupciones de software o de hardware, mientras que el dispatching es realizado por las rutinas de manejo de interrupción. Hay dos formas de utilizar esta técnica para administrar tareas. La primera es utilizar un timer para que cada cierto tiempo genere una interrupción que produzca un cambio de tarea. La otra forma es asociar cada evento a una interrupción externa.

La programación de las tareas es más simple que en los casos anteriores y por lo tanto más robusta. Debido a la normalización que significa tener un scheduler y un dispatcher, independientes de las tareas de usuario, los programas de aplicación resultan además muy portables. Debe mencionarse también que en este entorno es simple agregar un nuevo proceso, lo cual no repercute en problemas de sincronización. Pero los sistemas operativos multitarea son más costosos en lo que se refiere a utilización de memoria y requieren de una permanente toma de decisiones, durante el tiempo de ejecución. Son por ello más lentos.

Debe tenerse en cuenta que estamos haciendo referencia a interrumpir tareas que se están ejecutando, para realizar otra tarea y luego retomar la primera. Como ya se mencionó, para que la primer tarea pueda continuar ejecutándose, debe tenerse la precaución de preservar la información indispensable que existía antes de entregar el control a la segunda tarea (esta operación se denomina un *context switch*). La información preservada se denomina *contexto* y es almacenada en una estructura tipo stack. La mínima información indispensable que una tarea necesita para retomar su ejecución es la siguiente:

> Contenido de los registros del procesador
> Contenido del *program counter*
> Contenido de los registros del coprocesador
> Registros de página de memoria

[4] Este comentario trasciende el aspecto técnico.

Un punto importante que debe aclararse es el siguiente: si se depende del sistema operativo para realizar la administración de tareas, debe tenerse en cuenta que cada vez que se presente un *evento* estarán presentes las tareas asociadas al scheduling y el dispatching. Por lo tanto el tiempo de respuesta a interrupciones externas se verá desmejorado, ya que existe una sobrecarga debido al sistema operativo. Una medida utilizada para caracterizar las bondades de un sistema operativo en este aspecto es el *tiempo de latencia*, que se define como el tiempo que se demora entre la aparición de la interrupción y la ejecución de la primera instrucción asociada a ella.

Existen varias estrategias de scheduling que pueden utilizarse, basadas en el uso de interrupciones.

Sistemas de tiempo compartido (time slice): A cada tarea se le asigna una cantidad fija de tiempo durante la cual puede ejecutar. Se utiliza un reloj interno de la computadora para iniciar una interrupción con una tasa correspondiente a esa "rebanada" de tiempo. Cuando llega una de estas interrupciones se produce un *context switch* y se cambia a la tarea siguiente. Puede haber diferentes prioridades para las tareas, lo cual se maneja cambiando el ancho de la rebanada de tiempo que se le entrega a cada uno o la frecuencia con que se le entrega dicha rebanada. Nunca un proceso se queda sin tiempo de CPU. Estos sistemas garantizan una performance promedio de las tareas (sistemas probabilísticos). Tienen un muy buen desempeño como sistemas de propósitos generales, pero el tiempo que demora una tarea en completarse es muy dependiente del resto de las tareas presentes.

Sistemas preemptivos basados en prioridades: Estos sistemas utilizan un esquema *preemptivo* en lugar de un scheduler *time slice*. Se habla de *preempt*[5] cuando una tarea de prioridad más alta es capaz de interrumpir la ejecución de una de prioridad más baja. Cada vez que una tarea de alta prioridad desea ejecutarse, interrumpe inmediatamente a la tarea de más baja prioridad que pudiera estar ejecutándose al momento de su arribo. Si la que se está ejecutando es de prioridad mayor, la tarea se pone en espera. Este tipo de scheduling es el más adecuado para sistemas de tiempo real, fundamentalmente cuando existen tareas breves de alta importancia. En el caso de la existencia de una tarea crítica se le asigna la más alta prioridad. Para ella, el tiempo de respuesta es solamente el del context switch. Por supuesto que este tipo de sistemas presenta una sobrecarga debido a la toma de decisiones del kernel. En este tipo de sistemas es posible predecir exactamente el tiempo que le toma a una tarea completarse (sistemas determinísticos). Un tema muy importante es el estudio de las diferentes técnicas de asignación de prioridades, tema que se trata al final del capítulo.

Sistemas tipo round-robin: la estrategia es similar a los del tipo preemptivo, con la diferencia que las tareas de igual prioridad se ejecutan dentro de un esquema de tiempo compartido.

iv. Sistemas foreground/background

Se trata de una mejora a los sistemas basados en interrupciones, donde se reemplaza al programa principal que no realiza ninguna tarea por una serie de procesos llamados *de background* o *de fondo*, que no son manejados por interrupciones. Estos procesos de background pueden ser interrumpidos en cualquier momento por los procesos *de foreground*, manejados por interrupciones. Los procesos de background pueden utilizar un esquema de polled-loop para realizar varias tareas. Esta es una solución muy adecuada para embedded systems.

En sistemas más sofisticados puede utilizarse un esquema *time slice* para los procesos del background y un esquema *preemptivo* para los del foreground. Existen actualmente sistemas operativos de tiempo real con extensiones para manejar complicados device drivers, redes, etc. Para implementar este tipo de sistemas operativos de utiliza una estructura tipo Task Control Block como se mencionó anteriormente.

[5] Las posibles traducciones de este anglicismo serían desagradables (preemptimiento, preeptación, preemtismo).

Mecanismos de coordinación entre tareas

Se introducen a continuación los posibles mecanismos previstos por diferentes sistemas operativos para proveer de comunicación y sincronización a los múltiples procesos activos. Se trata de un tema importante en cualquier sistema multitarea pues está relacionado directamente con el problema de que varias tareas compartan un mismo recurso (llamado también *problema de exclusión mutua*).

Existe una gran cantidad de mecanismos de coordinación en el amplio espectro de sistemas operativos disponibles. Ninguno de ellos implementa la totalidad de los mecanismos descriptos a continuación. Estos mecanismos se diferencian en la cantidad de programación necesaria para utilizarlos, en la flexibilidad del servicio que prestan, la velocidad, confiabilidad y funcionalidad. La clasificación que sigue a continuación es una recopilación de la escasa bibliografía referida al tema. Debe aclararse que la diferencia entre un mecanismo de comunicación y otro es a veces muy sutil.

Sin embargo, en general, existen cuatro categorías diferentes, cada una con características propias: *variables globales y memoria compartida*, *mensajes y mailboxes*, *semáforos y mutexes*, y por último *señales*. A continuación se presentan los aspectos fundamentales de cada una de estas categorías.

i. Variables globales y memoria compartida

Varios mecanismos pueden ser utilizados para pasar datos entre procesos en un sistema multitarea. La forma más simple y rápida es la utilización de variables globales o zonas de memoria compartida por dos o más procesos. Si bien son consideradas contrarias a un buen diseño de software, son muy utilizadas en operaciones que requieren alta velocidad. Se trata de mecanismos de muy bajo nivel.

Uno de los problemas relacionados con las variables globales es que una tarea de alta prioridad puede remover a una de prioridad más baja en el transcurso de la escritura, produciendo un dato incorrecto. Para evitar esta situación, deben protegerse con algún mecanismo adicional, como semáforos o mutexes. Por lo tanto son de implementación complicada.

Los distintos sistemas operativos implementan diferentes métodos de utilización de zonas de memoria compartida. Los kernel se encargan de presentar estas zonas con formatos similares a los archivos.

ii. Mensajes y mailboxes

La forma más general y abstracta de comunicación es el intercambio de mensajes. Un *mailbox* es una zona de memoria acordada por dos tareas para el intercambio de datos. Estas tareas dependen del scheduler para ser habilitados a escribir en esa zona a través de la operación *write o send*, o ser habilitados a leer con la operación *read o receive*. La diferencia entre utilizar receive o simplemente hacer un polling del mailbox hasta que el mensaje se haga presente, es que la tarea es suspendida hasta que el nuevo dato aparece. Por lo tanto no se desperdicia tiempo verificando el mailbox.

Con este mecanismo pueden intercambiarse tanto flags que protegen un recurso, datos, o bien punteros a zonas de memoria donde residen los nuevos datos. Incluso pueden utilizarse para implementar los mecanismos de sincronización descriptos en el punto siguiente.

Pipes y FIFOs

Se trata de mecanismos de comunicación entre procesos manejados por el kernel tal como si fueran archivos. Un *pipe* es un arreglo de dos descriptores de archivos, uno de escritura y otro de lectura. Se lee y se escribe en ellos de la misma forma que en archivos: con *read* y *write*. Estos descriptores de archivo son heredados por los procesos hijo, por lo tanto son la forma de comunicación entre procesos creados por nuestra aplicación. Su creación es muy simple: utilizando la llamada al sistema pipe se crean los dos descriptores de archivos y los datos se acceden de tal forma que el primer dato en entrar es el primero en salir.

Un *FIFO* es similar a un pipe, pero con un nombre en el sistema de archivos. Esto significa que cualquier proceso con los permisos adecuados puede acceder al pipe, sin necesidad de heredar el descriptor. Los FIFOs se abren con *open*, para solo lectura o solo escritura, dependiendo que extremo se desee. Pero primero deben ser creados, para lo que existe una llamada al sistema. También se denominan *named pipes*.

Los pipes y FIFOs tienen algunas desventajas. Pueden mencionarse, por ejemplo, que no puede darse prioridades a los mensajes, que no existe control sobre la estructura del pipe o que el tipo de datos es no estructurado.

Message queues

Los problemas presentes en los pipes, mencionados anteriormente, son en parte subsanados por la estructura de cola de mensajes o *message queue*. Éstas tienen una estructura de mensaje más marcada y dichos mensajes pueden ser priorizados. Además se dispone de cierto control sobre la cola de mensajes. Los mensajes pueden ser enviados sin conocer el destinatario. Tiene la ventaja, frente a la memoria compartida, que pueden implementarse mensajes sobre sistemas distribuidos. Incluso existe la posibilidad de que un proceso sea notificado cuando un mensaje fue puesto en la cola, sin necesidad de que éste pregunte.

El kernel es quien implementa estas facilidades, pero debe aclararse que en distintos sistemas operativos existen diferentes implementaciones de estas colas, no todas con las mismas propiedades, funcionalidad y performance.

Algunas limitaciones se mantienen. El tipo de comunicación está limitado al formato de cola; es complicado implementar un stack. Además su eficiencia es limitada, ya que el mensaje debe ser copiado desde el emisor al sistema operativo, y del sistema operativo al receptor; estas dos operaciones de copia consumen recursos, que son mayores cuanto más grande es el mensaje.

iii. Semáforos y mutexes

En sistemas multitareas un aspecto muy importante y crítico es el de los recursos compartidos y su implementación. En la mayoría de los casos, los recursos pueden ser utilizados por una sola tarea a la vez, y la utilización del recurso no puede ser interrumpida. Estos recursos se denominan *serially reusable*, e incluyen los periféricos, memoria compartida y la CPU. La CPU está protegida de utilizaciones simultáneas, pero el código que interactúa con el resto de dichos recursos no lo está. Ese código se denomina *región crítica*. Si dos tareas entran en una región crítica simultáneamente, un error catastrófico puede ocurrir. Esta situación se denomina *colisión*, y los sistemas operativos proveen diferentes mecanismos para evitarlas.

Semáforos binarios

Esta metodología de protección[6] involucra una variable especial llamada *semáforo S* y dos operaciones posibles sobre éste (*semaphore primitives*) *P(S)* y *V(S)*[7]. Un semáforo es una posición de memoria que sirve como lock para proteger una sección crítica. La operación P se utiliza para setear el semáforo y la operación V para resetearlo. La primera se llama también operación de *espera* y la segunda, operación de *señalización*. El efecto de la operación P es suspender el proceso que requiere del uso del semáforo hasta que éste esté en "0". La operación de señalización pone el semáforo en "0". Esta técnica evita que más de un proceso ingrese en la sección crítica. En esta categoría puede incluirse la técnica de *file locking*.

Semáforos contadores

Con la técnica descripta anteriormente, el semáforo sólo puede tomar los valores "0" o "1". Con la técnica de *semáforos contadores*, cada operación de señalización incrementa en uno el valor del semáforo, que puede tomar cualquier valor entero positivo. La operación de espera decrementa el valor del semáforo, o suspende la ejecución si es cero, hasta que éste se incremente.

Mutexes y variables de condición

Los *mutexes* son mecanismos que proveen exclusión mutua (los procesos se excluyen unos a otros) para el acceso a una zona crítica. Suele sugerirse la idea de imaginar a la sección crítica como una habitación en la que solo puede entrar un proceso. El mutex es la puerta a dicha habitación, que puede ser trabada o destrabada. El proceso que traba la puerta (el mutex) es el único que puede destrabarla. Este concepto es diferente del semáforo. Los mutex tienen asociada la idea de dueño, mientras que los semáforos no.

---

[6] Esta técnica fue introducida por Edsger Dijkstra en 1965. [Dijkstra, 1965].

[7] P proviene del alemán *proberen* y V de *verhogen*, testear e incrementar, respectivamente.

Las *variables de condición* están asociadas a los mutexes. Imagínese la situación en que un proceso traba un mutex, entra a la sección crítica y se da cuenta de que una condición necesaria aún no ha sido satisfecha por otro proceso. Entonces debe destrabar el mutex y volver luego. Esperando por una variable de condición, el mutex se destraba y se espera la condición atómicamente, en una acción no interrumpible. Cuando el otro proceso satisface la condición necesaria, modifica la variable de condición, y despierta al primer proceso, que vuelve a trabar el mutex y ejecuta la acción pendiente, con la condición satisfecha.

Locks de lectura/escritura

Esta técnica se basa en que varios procesos pueden leer simultáneamente una zona compartida, pero si se desea escribir en dicha zona se debe esperar a que ésta esté libre de lectores. Con el mismo criterio, si hay un proceso modificando la zona compartida, los lectores que aparezcan serán bloqueados hasta que el primero termine su trabajo.

iv. Señales

Las *señales* son parte integral del entorno multitarea de los sistemas tipo UNIX. Éstas son utilizadas con diferentes propósitos, como por ejemplo manejo de excepciones, notificación a los procesos de la ocurrencia de eventos asíncronos, eliminación de procesos en circunstancias anormales y comunicación entre procesos.

Una señal es el equivalente de software de una interrupción. Cuando un proceso recibe una señal, esto indica que algo ha sucedido que requiere su atención. Como un proceso puede enviar una señal a otro, ésta puede utilizarse como un método de comunicación. Desde este punto de vista son lentas, limitadas e interrumpen a los procesos asíncronamente, lo que requiere de un tratamiento especial por parte del receptor.

Las señales pueden ser enmascaradas, lo que les da una cierta flexibilidad en la utilización como comunicación entre procesos.

Al ser un mecanismo standard de todos los sistemas tipo UNIX, la información al respecto es abundante y no se tratará este tema con detenimiento. En el capítulo de bibliografía, en la sección de programación se presentan varios textos que se encargan del tema.

Otros aspectos de importancia en sistemas de tiempo real

Además de la estrategia de scheduling necesaria para administrar las tareas, y de la comunicación necesaria entre dichas tareas, hay otros aspectos a tener en cuenta en el momento de analizar las propiedades de tiempo real de un sistema operativo. Se presentan a continuación algunos puntos importantes respecto de la administración de memoria, relojes y entrada/salida del sistema.

Memory lock

En sistemas de tiempo real debe evitarse que las páginas de memoria (en un sistema de memoria virtual) asociadas a una tarea con restricciones estrictas de tiempo sean enviadas al disco por falta de memoria para otras tareas (este fenómeno se denomina *swapping*). Esto produciría un incremento indeterminado en el cambio de contexto, ya que no puede saberse a priori si las páginas fueron enviadas a disco o no. Para ello, la mayoría de los sistemas de tiempo real proveen facilidades para evitar que esto ocurra. Los sistemas más inteligentes proveen dos formas de trabar en la memoria. La primero hace un lock del proceso completo en memoria, pero en el caso de que se supere la capacidad de memoria disponible, se dispone de una segunda opción que permite hacer un lock en memoria solo de las porciones críticas del proceso.

Relojes y timers[8] de tiempo real

Como es lógico pensar, este es un punto muy importante en sistemas de tiempo real. Es fundamental que el sistema provea una base de tiempo adecuada para la sincronización interna de los procesos presentes. Debe aclararse aquí la diferencia entre *reloj* y *timer*. El primero es simplemente un contador que provee una base de tiempo, y el segundo es un contador que llegado a cierto estado, es capaz de notificar que esto ha sucedido. Para la implementación de un timer es necesario un reloj.

Un sistema de tiempo real debe ser capaz de medir internamente el tiempo con la resolución y precisión adecuada para el caso de aplicación. Es deseable también que el sistema sea capaz de reconocer cuándo un timer ha expirado varias veces (*timer overrun*) y que sea también capaz de generar una señal standard ante la expiración normal de un timer.

Un aspecto importante es la capacidad del sistema de sincronizar su reloj interno con el entorno, o sea con el "tiempo real". De la misma manera, en un sistema distribuido, es necesaria una adecuada sincronización entre los diferentes nodos. Los principales problemas que se presentan son los drifts debido a la temperatura e incluso la falla de los relojes, problemas que deben ser subsanados para evitar la falla total del sistema. Uno de los métodos utilizados es el empleo de una base de tiempo global, a la cual tienen acceso todos los nodos.

Otro método es la implementación de un promedio entre los relojes presentes en el sistema. Para ello se utilizan algoritmos que prevén los retardos debido a la comunicación y posibles relojes *maliciosos*. Uno de los algoritmos más utilizados es *Fault tolerant average algorithm* de Ochsenreiter y Kopetz, de 1987 [Buttazzo, 1997], que se encuentra disponible como un elemento de hardware que puede incluirse en el sistema.

Entrada/salida en sistemas de tiempo real

Es fundamental que, una vez realizado el procesamiento en tiempo real, la interacción con los dispositivos externos sea también acotada en tiempo. Es importante que el sistema operativo disponga de un sistema de entrada/salida sincrónico. Por ejemplo, el sistema de archivos debe tener acotados los tiempos de respuesta, si se desea que dichos archivos sean de "tiempo real".

Para la transmisión de datos entre el sistema de tiempo real y los diversos sensores y actuadores que pueden estar presentes en el sistema (incluso para la comunicación de datos entre distintos nodos de un sistema distribuido) existen diferentes técnicas de buses de tiempo real, que permiten disponer de sensores inteligentes. Éstos no solo transmiten el dato adquirido, sino que además envían la información acerca del momento en que dicho dato fue tomado (por ejemplo *Fieldbus*).

Dentro de los protocolos de comunicación que disponen de la capacidad de acotar los tiempos de transmisión pueden mencionarse el protocolo PAR (Positive Acknowledge or Retransmit), Implicit Flow Control, CSMA/CD (Carrier Sense Multiple Access/Collision Detection), CAN (Control Area Network), Tokenbus (por ejemplo Profibus), Central Master, y TDMA-TTP (Time Division Multiple Access – Time Triggered Protocol).

La asignación de prioridades dentro de un esquema preemptivo

Como ya fue mencionado anteriormente, en sistemas de tiempo real es deseable una estrategia preemptiva basada en prioridades. Pero, suponiendo que debe garantizarse el cumplimiento de los requerimientos de tiempo de un cierto grupo de tareas, ¿cómo deben ser asignadas las prioridades a las diferentes tareas para que se cumplan completamente los objetivos?

Hasta ahora se ha mencionado que a las diferentes tareas pueden asignársele diferentes prioridades, dentro de un esquema preemptivo, pero no se ha hecho referencia acerca de qué técnica utilizar para decidir qué prioridad asignar a cada tarea, para que se cumplan los requerimientos de todas ellas. El

[8] Utilizar como traducción de timer el término *temporizador* podría llevar a confusiones.

esquema preemptivo no garantiza el cumplimiento de los objetivos; es solamente el entorno más eficiente para la implementación. Algunos problemas de scheduling son irresolubles, y algunos otros tendrán solución solamente con una asignación específica de prioridades.

En algunos casos, la solución al problema de scheduling es simple. Por ejemplo, si sólo una de las tareas es crítica para el sistema, se le asigna a ésta la mayor prioridad y a las demás una prioridad menor. Simple, ¿pero que sucede si existen varias tareas que son críticas? ¿Cómo se diferencia entre las tareas que deben tener la mayor prioridad y las que deben tener la menor? ¿Es necesario alterar las prioridades dinámicamente para satisfacer los requerimientos de tiempo de la aplicación?

Existe actualmente un intenso debate (mayormente académico) acerca de qué algoritmo de scheduling es el más apropiado para sistemas de tiempo real. Mientras tanto, en el campo de aplicación, existe un grupo de técnicas de análisis y diseño que están siendo utilizadas. Algunas de ellas se presentarán a continuación.

Análisis rate-monotonic

Para tareas periódicas con requerimientos estrictos, se presenta a continuación una primera aproximación. Utilizando el *análisis rate-monotonic*, es simple dilucidar qué prioridad debe asignarse a cada proceso, y analizar si dichas tareas alcanzarán a satisfacer los requerimientos de tiempo[9].

Para que pueda aplicarse este tipo de análisis, deben cumplirse algunas otras condiciones, además de que las tareas sean periódicas. Se supone también que el tiempo límite para la ejecución de cada tarea es igual a su período, y que el momento de ejecución de cada tarea es al comienzo de éste. Esto significa que cada cierto intervalo de tiempo se ejecuta la tarea, y debe terminarse antes de que llegue el nuevo intervalo. Esto no siempre es cierto, y puede ser que la tarea tenga requerimientos más estrictos.

Se supone, además, que las tareas son independientes una de la otra, que el tiempo consumido en la toma de decisiones es despreciable y que las tareas no pueden suspenderse a sí mismas. Y, por supuesto, que las tareas tienen tiempo de ejecución acotado.

Dentro de este panorama un tanto ideal, la asignación de prioridades rate-monotonic es simple. Debe conocerse la frecuencia de ejecución de las tareas periódicas de tiempo real. Para tareas asincrónicas puede utilizarse el peor caso de arribo como dicha frecuencia, al solo efecto de la asignación de prioridades. A la tarea de mayor frecuencia se le asigna la mayor prioridad y a la de menor frecuencia la prioridad menor. Las tareas de frecuencia intermedia reciben prioridades intermedias, en función de su frecuencia. No debe asignarse la misma prioridad a dos procesos. Todas las prioridades asignadas a las tareas de tiempo real deben ser mayores que la prioridad de cualquier otra tarea presente que no sea de tiempo real. Si dos tareas tienen frecuencias idénticas, se asignan prioridades indistintamente.

Durante la ejecución, la decisión de qué tarea debe ejecutarse en cada momento también es simple de implementar: la tarea disponible con más alta prioridad es ejecutada.

Pero, lógicamente, la asignación de prioridades no alcanza para garantizar que todas las tareas cumplan sus requerimientos de tiempo (lo que llamaremos *factibilidad de la implementación*). Para realizar el análisis de factibilidad es necesario saber también cuánto tiempo demora cada tarea en ejecutarse. Como se mencionó anteriormente, este tiempo debe estar acotado, y debe ser conocido.

Conocer el tiempo que demora una tarea en ejecutarse no es fácil, ya que debe tenerse en cuenta que generalmente los diferentes sistemas operativos disponen de caches, que hacen que para ejecutar la tarea una única vez se necesite más tiempo que si se ejecuta cíclicamente. Los diferentes sistemas disponen, en general, de herramientas para la medición del tiempo de ejecución de un grupo de funciones.

Una vez conocida la frecuencia y el tiempo de ejecución de cada tarea, puede realizarse el test de factibilidad, que en el caso del análisis rate-monotonic es muy simple. Si la siguiente condición se cumple, el grupo de tareas cumple los requerimientos de tiempo.

---

[9] El análisis rate-monotonic es una técnica introducida en 1972 por los doctores C.L. Liu y J.W. Layland [Buttazzo, 1997]. En su momento pasó desapercibida hasta que los investigadores de la Universidad de Carnegie Mellon lo rescataron en la década del 80.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

Para cada tarea i, C_i es el tiempo de ejecución y T_i es el período. Nótese que el segundo término es aproximadamente igual al *ln 2*, lo que equivale a decir que es aproximadamente igual a *0.69*.

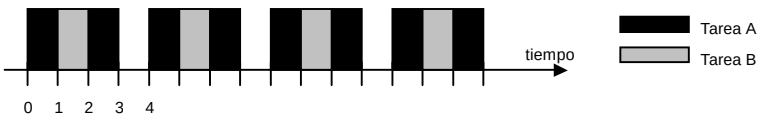
Por consiguiente, para saber si la implementación de un grupo de tareas (que cumplen los requisitos mencionados) es factible asignando prioridades según el análisis rate-monotonic, solo deben sumarse las relaciones C/T de cada tarea (a esta relación la llamaremos *factor de utilización*[10]). Si la suma resulta menor que 0.69, la implementación cumplirá los requerimientos de tiempo.

Por ejemplo, sea la siguiente aplicación que consta de cinco tareas, como se muestra en la siguiente tabla. La tarea de interfaz con el usuario U se ejecuta 10 veces por segundo. Dos tareas de tiempo real estrictas A y B se ejecutan a 100Hz. Una tarea de recolección de datos C a 33Hz, y el almacenamiento de datos en disco D a 5Hz. La asignación de prioridades rate-monotonic asigna las más altas prioridades a las tareas A y B, por ejemplo 90 y 100. C es la frecuencia que sigue, por lo tanto se le puede asignar 80. U recibe 70 y finalmente D recibe la prioridad más baja, por ejemplo 60. Utilizar números para las prioridades bastante separados entre sí, sirve para no tener que recalcular todas las prioridades cuando se desea agregar una tarea intermedia.

En la siguiente tabla se presentan las tareas periódicas, junto con su frecuencia y tiempo de ejecución, factor de utilización y prioridad asignada según el método rate-monotonic. Sumando los factores de utilización se obtiene 0.657. Esto indica que utilizando dicha asignación de prioridades, la aplicación cumplirá las especificaciones temporales.

Tarea	Frecuencia	Tiempo de ejecución	Factor de utilización	Prioridad
A	100	1 milisegundo	0.1	90
B	100	2 milisegundos	0.2	100
C	33	4 milisegundos	0.132	80
D	5	5 milisegundos	0.025	60
U	10	20 milisegundos	0.2	70

Debe aclararse que esta condición es pesimista, ya que es solo una condición suficiente. Podría existir un grupo de funciones que no cumpla el test y en cambio sí funcione en una implementación real. Lo que *siempre* es cierto es que si cumple el test, la implementación es factible. Un ejemplo clásico de que no es una condición necesaria es el siguiente grupo de tareas. Una tarea A con período 2 y tiempo de ejecución 1, más una tarea B con período 4 y tiempo de ejecución 1. La asignación de prioridades rate-monotonic indica mayor prioridad para la tarea A y menor para la B. El test no se satisface, ya que ½ + ¼ = 0.75 > 0.69. Sin embargo, sin mucho esfuerzo puede imaginarse la secuencia en el tiempo de la siguiente figura, que demuestra que es implementable.



La demostración del test de factibilidad se basa en el análisis del peor caso. Esto implica que todas las tareas comienzan al mismo tiempo y que, por lo tanto, las tareas de baja prioridad serán retardadas al máximo por las de alta prioridad (instancia crítica). La demostración debe encargarse de probar que la

[10] Una condición necesaria para la implementación es que la suma de los factores de utilización de todas las tareas sea menor que 1. Si no es así, no hay tiempo de CPU suficiente para ejecutar todas las tareas, y ningún algoritmo de scheduling proveerá la solución.

instancia crítica es realmente el peor caso (lo que es bastante complicado) y que todas las tareas cumplen los requisitos de tiempo hasta en el peor caso (tiempo de respuesta de la tarea de prioridad más baja menor que su tiempo límite de ejecución).

A pesar de que este test es solo una condición suficiente, es muy utilizado, aunque más no sea como análisis previo, por su simplicidad. Para el análisis exacto se presenta el método que sigue a continuación.

Análisis exacto

El análisis rate-monotonic es simple, pero pesimista, como se mostró anteriormente. El *análisis exacto*, introducido por Joseph y Pandya [Buttazzo, 1997], también está basado en el análisis de la instancia crítica. Introduce un test de factibilidad mucho más complejo y difícil de calcular. Se trata de una serie numérica, que si converge es una condición suficiente y necesaria para la implementación.

Análisis EDF

El análisis rate-monotonic tiene como una de las condiciones para las tareas que el tiempo límite para su ejecución sea igual a su período. Esto significa que para ejecutar una tarea de período T, se dispone de un tiempo T. Esto no es muy común en la realidad. Muchas veces es necesario que la tarea sea ejecutada en un tiempo menor, aunque su período sea T. Tal es el caso de una adquisición de datos, en la cual se desea que cada toma de datos se realice en forma precisa, a un ritmo T (frecuencia de muestreo constante).

El *análisis EDF* (earliest deadline first) hace las mismas suposiciones respecto de las tareas que el análisis rate-monotonic, con la excepción que los deadlines pueden ser menores que los períodos. En este método, la asignación de prioridades es diferente. Éstas se asignan dinámicamente en cada momento, tal que la tarea con el deadline más próximo recibe la más alta prioridad. En el momento de la ejecución, la tarea de más alta prioridad es ejecutada, lo que significa que constantemente se ejecuta la tarea con el deadline más cercano.

En este caso, luego de una extensa demostración, el test de factibilidad resulta muy simple. Es condición suficiente y necesaria que se cumpla la siguiente condición.

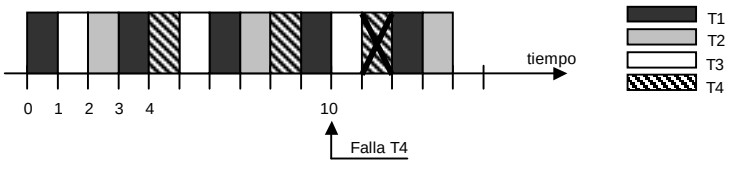
$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Esto significa que la asignación de prioridades EDF es óptima: si cualquier algoritmo puede encontrar una secuencia, EDF la encontrará también. Es suficiente que se cumpla que el factor de utilización de la CPU sea menor que uno, lo cual es evidente.

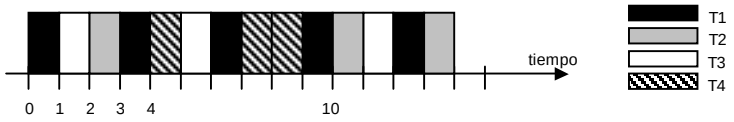
Se presenta a continuación un grupo de tareas que no satisface los deadlines si se utiliza la estrategia RM y sí lo hace con EDF. Sean las siguientes tareas:

T1 con período 3 y tiempo de ejecución 1;
T2 con período 6 y tiempo de ejecución 1;
T3 con período 5 y tiempo de ejecución 1; y
T4 con período 10 y tiempo de ejecución 3.

Intentando una secuencia de ejecución según el método RM, T1 recibe la prioridad más alta, T3 la siguiente, T2 la siguiente, y T4 la más baja. Si se suponen todas disponibles en t=0, luego de ejecutarse T1, T3 y T2, como se muestra a continuación, solo quedan dos rebanadas de tiempo para T4, mientras que necesita 3, y pierde su deadline de 10 (nótese que debido a una ejecución anticipada de la segunda instancia de T2 en t=7, dada por su mayor prioridad, T4 sólo consigue ejecutarse en las rebanadas de tiempo 4, 8 y 11, produciendo la falla de T4 en t=10).



Aplicando una estrategia EDF puede verse cómo la segunda instancia de T2 se retrasa, sin perder su deadline, hasta la rebanada de tiempo 11, permitiendo que T4 se ejecute en las rebanadas 4, 7 y 8. El momento crucial de decisión es t= 7. En ese momento el método RM opta por ejecutar T2 ya que tiene prioridad más alta que T4. En cambio EDF opta por T4, pues su deadline está más próximo.



Debe aclararse que si bien este método resulta más atractivo en lo que se refiere al análisis y la performance, es mucho más complejo en el momento de la implementación. Esto se debe a que las prioridades asignadas a las tareas deben ir modificándose a lo largo del tiempo. Este aspecto implica una mayor sobrecarga para el scheduler y el dispatcher, que deberán ser más elaborados que en el caso anterior. Debe recordarse que el análisis teórico se hace considerando que la sobrecarga al sistema debido al scheduler es despreciable, lo cual puede ser no siempre cierto, y más aun cuanto más complicada es la estrategia de scheduling.

Tareas dependientes. Inversión de prioridades, dead-locks y bloqueo encadenado

Hasta ahora se han presentado diferentes formas de análisis y asignación de prioridades para tareas periódicas e *independientes*. Esto último significa que si dos tareas comparten un recurso, o deben estar sincronizadas, los análisis anteriores no son aplicables. Pero en aplicaciones reales es siempre necesaria la utilización de semáforos, secciones críticas, etc.

Los tests de factibilidad no pueden utilizarse con tareas dependientes. Y aun peor, si existe al menos un recurso compartido, el problema de encontrar el algoritmo de scheduling se torna *NP hard* (NP significa No Polinomial), como lo demostraron Garey y Johnson [Buttazzo, 1997]. NP hard implica, básicamente, las siguientes consecuencias:

el esfuerzo para encontrar la solución óptima no es 'tratable' (*tractable*); y
el esfuerzo para resolver el problema crece más que polinomialmente (es exponencial con la dimensión del problema).

El análisis teórico es extremadamente complejo. Sin embargo debe tratarse de algún modo, ya que son los verdaderos casos de aplicación. Antes de presentar algunas estrategias utilizadas se presentarán los principales problemas a tener en cuenta. De un análisis de éstos surgen algunas soluciones posibles. Se presentará el primero de ellos utilizando un ejemplo.

Considérese el caso en el que exista un recurso *exclusivo*[11] (que puede ser accedido por una sola tarea simultáneamente) protegido por un semáforo S, más dos tareas que acceden a este recurso:

A con prioridad alta, y
Z con prioridad baja.

Supóngase que la tarea Z comienza a ejecutarse y entra en la sección crítica; incrementa el semáforo con P(S) y es aceptado. En ese momento, la tarea A está lista para ejecutar, y como tiene mayor prioridad, desplaza a Z. Cuando A intenta acceder al recurso encuentra el semáforo ocupado y P(S) es negado pues está en poder de Z. La tarea A es puesta en estado de espera. Z continúa ejecutando, y finalmente libera la sección crítica decrementando el semáforo con V(S). En ese momento A está en condiciones de ejecutarse nuevamente pues ahora sí puede acceder al recurso compartido.

En este caso pudo verse que la tarea Z de menor prioridad se ejecutó antes que la tarea A de prioridad mayor. Este fenómeno se denomina *inversión de prioridades*, y es un aspecto muy importante a tener en

[11] Tal puede ser el caso de una zona de memoria que es importante que la tarea que la escribe la complete totalmente, antes de permitir que otra tarea la lea.

cuenta en el diseño de sistemas de tiempo real. Puede ocurrir siempre que dos tareas de diferente prioridad accedan a un mismo recurso.

La situación puede ser aún peor. En el ejemplo anterior se presentó la tarea A como capaz de ser suspendida en el caso de encontrar un recurso ocupado, entregando la CPU a la tarea Z para que continúe su procesamiento y libere el semáforo. Pero supóngase que la tarea A no puede suspenderse y continúa acaparando la CPU hasta que el recurso sea liberado. Esto nunca ocurre, pues la tarea Z está suspendida. Dicha situación se denomina *dead-lock* y lleva a la paralización total de la aplicación, con las consecuentes catástrofes asociadas.

Además, la inversión puede darse entre más de dos tareas. Supóngase que en el panorama anterior existe otra tarea de prioridad intermedia M, que no accede al recurso compartido. Si una vez que Z entró en la zona crítica es desplazada por M, y ésta es a su vez desplazada por A, ocurre una doble inversión de prioridades. Cuando A es suspendida porque encuentra el recurso ocupado, M retoma la ejecución pues es la de prioridad siguiente. Cuando M termina, Z puede ejecutarse, luego de lo cual libera el semáforo. Es recién en ese momento cuando A (la tarea de mayor prioridad) puede retomar la ejecución: después que M y Z (de prioridad menor) terminaron.

Lo que agrava la situación es que dentro de un sistema operativo existen recursos que se comparten sin el conocimiento del usuario. Por ejemplo, considérese el caso de dos tareas que tienen archivos abiertos. Incluso cuando estos dos archivos no estén relacionados, ambas tareas necesitan acceder a los mismos bloques de directorio para tener acceso a los archivos. Dichos bloques deben ser compartidos. Esta es una fuente de inversión de prioridades sobre la cual no se tiene control.

Estas posibles condiciones de las tareas con recursos compartidos son el factor que hace que el análisis de este tipo de sistemas sea más complicado que para EDF o rate-monotonic.

Existen dos formas de evitar la inversión de prioridades, y ninguna de las dos suena muy atractiva. La primera es analizar la aplicación para determinar en qué momento puede ocurrir una inversión, calcular cuanto tiempo de demora implica ésta y cada cuanto sucede. Este tiempo debe ser considerado en el tiempo de respuesta de cada tarea afectada. El problema es que, además de que pueden existir inversiones ocultas, este cálculo puede tornarse muy complicado para un número grande de tareas.

La segunda forma de evitar la inversión de prioridades es el llamado *protocolo de herencia de prioridades* (priority inheritance protocol ó PIP), introducido por Sha, Lehozcky y Rajkumar en 1990 [Buttazzo, 1997]. Con la herencia de prioridades se evita la inversión utilizando la siguiente regla: "cuando una tarea de baja prioridad Z está bloqueando a una tarea de prioridad alta A, Z continúa ejecutándose con la prioridad de A; en cuanto Z abandona la zona crítica, A continúa su ejecución". De esta forma, en el ejemplo anterior, M no podría desplazar a Z. Esta regla puede enunciarse también como sigue: "si una tarea de alta prioridad está bloqueada por una de baja prioridad, evitar que la de baja prioridad sea desplazada".

Debe aclararse que no es suficiente que esta técnica sea utilizada en la aplicación. Debe estar implementada, además, en el sistema operativo, para evitar complicaciones ocultas como la presentada en el ejemplo de los archivos.

Existe un fenómeno más que debe ser tenido en cuenta en el caso de aplicarse la herencia de prioridades. Se denomina *bloqueo encadenado*, y se presentará también con un ejemplo.

Sean tres tareas y dos semáforos, tal que:

A es una tarea de alta prioridad
M es una tarea de prioridad intermedia,
Z es una tarea de baja prioridad,
A y Z acceden al semáforo S_1 y
A y M acceden al semáforo S_2.

Inicialmente, ambos semáforos están libres. Considérese la siguiente secuencia de eventos.

Z comienza y accede a S_1;
M comienza, desplaza a Z y accede a S_2;
A comienza, desplaza a M e intenta acceder a S_1;
como S_1 está bloqueado por Z, Z comienza su ejecución con la prioridad de A, desplazando a M;
Z continúa su ejecución hasta que libera el semáforo S_1 y se resetea la prioridad de Z;

A desplaza a Z y accede a S_1;
A intenta acceder a S_2, que está bloqueado por M;
M hereda la prioridad de A y ejecuta hasta que libera S_2 y se resetea la prioridad de M;
A desplaza a M y accede a S_2.

En este ejemplo, la tarea A ha sido demorada nuevamente por dos tareas de prioridad más baja, M y Z. Esta vez fueron necesarios dos semáforos. Este fenómeno es lo que se denomina bloqueo encadenado. Sha, Lehozcky y Rajkumar [Buttazzo, 1997] diseñaron una nueva técnica para combatirlo, llamada *protocolo de techo de prioridades* (priority ceiling protocol ó PCP).

Esta técnica consiste en no dejar que una tarea de baja prioridad acceda a un semáforo que puede ser accedido por una tarea de alta prioridad. Para ello, al semáforo se le asigna un *techo*, que es la prioridad de la tarea de mayor prioridad que lo accede. Solo las tareas de prioridad más alta que el techo pueden acceder al semáforo. La regla de ejecución es: "la tarea T puede acceder a un semáforo solo si su prioridad es mayor que los techos de todos los semáforos que están actualmente bloqueados". De esta forma, los semáforos que pueden ser accedidos por tareas de alta prioridad son entregados de a uno por vez.

En el ejemplo anterior, el techo de S_1 y de S_2 es la prioridad de A. La secuencia de eventos sería la siguiente:

Z comienza y accede a S_1;
M comienza, desplaza a Z e intenta acceder a S_2; el acceso es negado pues M<A;
M es bloqueada por Z y Z hereda la prioridad de M;
A comienza, desplaza a M e intenta acceder a S_1;
como S_1 está bloqueado por Z, Z comienza su ejecución con la prioridad de A;
Z continúa su ejecución hasta que libera el semáforo S_1 y se resetea la prioridad de Z;
A desplaza a Z y accede a S_1;
A accede a S_2 y finaliza;
M puede ahora acceder a S_2.

Como puede verse, se evitó con esta técnica el doble bloqueo de la tarea de alta prioridad.

Scheduling de tareas no periódicas

Las técnicas presentadas anteriormente son válidas para el análisis de aplicaciones cuyas tareas son *todas* periódicas. Como se mencionó, puede considerarse la máxima tasa de arribo de las tareas no periódicas (peor caso) y utilizar este tiempo como su período. Pueden así incluirse en el análisis rate-monotonic o EDF. Si se puede conseguir un algoritmo de scheduling para el peor caso, no hay problema. Pero este análisis es muy pesimista.

Existen diferentes técnicas para la asignación de prioridades y el análisis de factibilidad de tareas no periódicas. Algunas de ellas se presentan a continuación.

Servicios de background

Las tareas aperiódicas se ejecutan cuando no hay tareas periódicas presentes. En este caso no existen garantías para las tareas no periódicas, ya que se ejecutan con el tiempo de CPU que sobra de las tareas periódicas.

Polling server

Esta técnica se utiliza para garantizar que al menos una cierta cantidad de procesamiento sea destinado a las tareas aperiódicas, y fue presentada en 1989 por Lehozcky, Sha, Strosnider y Sprunt [Buttazzo, 1997]. Para ello existe una nueva tarea periódica llamada *tarea servidor*, que se encarga de ejecutar las tareas aperiódicas lo más rápido posible. Esta tarea tiene asignado un cierto tiempo de ejecución, llamado *capacidad del servidor* C_s. El servidor tiene asignado también un período de ejecución T_s. Así, la relación C_s/T_s puede incluirse en el análisis de factibilidad. Si no hay tareas aperiódicas pendientes, la tarea servidor no se ejecuta. Una tarea aperiódica que aparece inmediatamente después del comienzo de la tarea servidor, con la CPU libre, debe esperar al próximo período, aunque podría ser ejecutada inmediatamente. Esto se debe a que la tarea servidor no se ejecuta si no hay tareas aperiódicas pendientes cuando arranca. Este es el peor caso, sobre la base del cual se realiza el análisis de performance.

Intercambio de prioridades

La técnica de *Priority Exchange*, presentada en 1987 por Lehozcky, Sha y Strosnider [Buttazzo, 1997], preserva la capacidad no utilizada por la tarea servidor para el futuro. Para ello la tarea servidor intercambia prioridades con el resto de las tareas. Los mismos autores presentaron también en 1987 su *Deferrable Server*.

Otras técnicas para el scheduling de tareas no periódicas con prioridades estáticas que pueden mencionarse son *Sporadic Server* (Sprunt, Sha y Strosnider, 1989) y *Slack Stealing* (Lehozcky y Ramos-Thuel, 1992) [Buttazzo, 1997].

En 1995, Tia, Liu y Shankar [Buttazzo, 1997] demostraron que con asignación estática de prioridades no existe un algoritmo para minimizar el tiempo de respuesta.

Servidores de prioridades dinámicas

Las técnicas anteriores tienen asignación estática de prioridades a las tareas, tomando como base el rate-monotonic. Pueden mencionarse también otras técnicas que utilizan como base el análisis EDF con prioridades dinámicas. Por ejemplo *Dynamic Priority Exchange Server* (Spuri y Buttazzo, 1994, 1996), *Dynamic Sporadic Server* (Spuri y Buttazzo, 1994, 1996) y *Earliest Deadline Late Server* (Chetto, Chetto, 1989). Uno de los servidores dinámicos más recomendados actualmente por su mejor desempeño es el *Total Bandwidth Server* (Spuri y Buttazzo, 1994, 1996). Para mayor detalle acerca de estos métodos se recomienda el texto *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications* [Butazzo, 1997].

Pre run-time scheduling

Esta técnica es un enfoque diferente al problema de scheduling. La toma de decisiones durante la ejecución, tal cual fue presentada hasta el momento, puede llegar a complicarse bastante en el caso de aplicaciones reales, donde coexiste una mezcla de tareas periódicas, aperiódicas, dependientes e independientes. Como se mencionó anteriormente, las reglas de toma de decisiones en el momento de la ejecución deben mantenerse simples para no sobrecargar al sistema con el scheduling. Esto no siempre es posible.

Existe un punto de vista diferente, que presenta una solución a este problema. Se trata de realizar un análisis más intensivo de las situaciones que pueden presentarse. Este análisis puede ser realizado previo a la ejecución del programa, en el momento del diseño. Si se consideran todas las posibilidades y se resuelven todos los problemas de scheduling con antelación, puede construirse una tabla que reemplace al scheduler. Las decisiones se toman previamente.

Por lo tanto no hay un kernel tal como se presentó anteriormente. Solo es necesario un dispatcher que se encargue de ir ejecutando las tareas en la secuencia precalculada que la tabla indica[12].

Existen varios métodos para la construcción de dicha tabla, entre los cuales puede mencionarse la utilización de los gráficos de precedencia, junto con procedimientos de evaluación de dichos gráficos. Tal es el caso de las estrategias *IDA* (Fohler, 1991), *Branch-and-bound* (Ramamritham, 1991) y *Two stage branch-and-bound for pipelining* (Fohler y Ramamritham, 1997).

Este método tiene la ventaja de ser capaz de enfrentar situaciones muy complejas, como sistemas no polinomiales y estructuras distribuidas, y la desventaja de ser muy poco flexible e incapaz de manejar situaciones excepcionales. Existen algunas técnicas mixtas, que combinan *pre run-time* y *run-time* scheduling. Tal es el caso de *Mode changes* (Fohler, 1991) y *Slot shifting* (Fohler, 1994).

Para mayor detalle acerca de estos métodos se recomienda también el texto *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications* [Butazzo, 1997].

Éste es un campo actual de investigación que se encuentra en evolución permanente y es motivo de múltiples debates académicos. En general, los sistemas reales más complejos (como es el caso de los embedded systems de aplicación en la industria aeronáutica y automotriz) están evolucionando hacia una combinación de dichas técnicas.

---

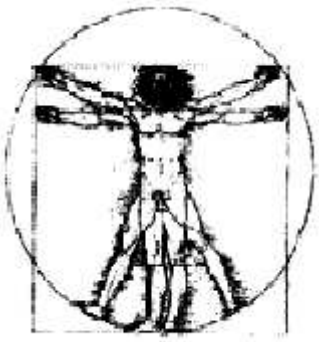
[12] Si, es cierto... el capítulo comenzó criticando el polled loop y terminó utilizándolo.

Resumiendo, en este capítulo se presentaron los aspectos básicos a tener en cuenta en el diseño de un sistema de tiempo real. Se trata de una introducción que intenta unificar los diferentes enfoques presentados en la escasa bibliografía disponible para este tema. Debe aclararse que algunos de los temas presentados se encuentran en evolución permanente, sobre todo el referido a las técnicas de scheduling y asignación de prioridades.

Estos conceptos se utilizarán en el *Capítulo IV* para el diseño de un pequeño kernel operativo para un microcontrolador de propósitos generales. La implementación se llevó a cabo utilizando una estrategia preemptiva basada en prioridades, con asignación de prioridades rate-monotonic.

El número de instalaciones Unix se ha elevado a 10 y se espera que dicho número aumente.

Unix Programmer's Manual, 2da. Edición, 1972.



Capítulo II
UNIX en tiempo real

En este capítulo se presenta una introducción a los principales aspectos de los sistemas operativos tipo UNIX (y de Linux en particular) relacionados con los conceptos de tiempo real. Si bien estos sistemas están diseñados como *sistemas operativos de propósitos generales*[13], la introducción de las normas POSIX abre un espectro muy amplio de aplicaciones en la rama de adquisición de datos y control.

Se analiza cada uno de los aspectos relevantes por separado, tales como técnicas de scheduling, memory locking, medición del tiempo y características de entrada/salida, tanto para los UNIX en general, como para los UNIX que cumplen la norma POSIX.4, y el Linux en particular.

Sobre el sistema operativo Linux se realizó una serie de mediciones que intenta caracterizar su comportamiento, tanto para la versión standard de éste como para la extensión Real-Time Linux.

Los sistemas UNIX

En 1969, Ken Thompson, del grupo de investigación de Bell Laboratories USA, comenzó el desarrollo de un sistema operativo multiusuario y multitarea, utilizando una computadora PDP-7. Al poco tiempo se sumó Dennis Ritchie al proyecto y ambos, junto con otros miembros del grupo de investigación, produjeron las primeras versiones de UNIX. Ritchie trabajó muy influenciado por un proyecto anterior llamado MULTICS[14], que utilizó como base [Leffler et al. 1990].

Las dos primeras versiones fueron escritas en código assembler, y la tercera de ellas fue reescrita en un nuevo lenguaje de programación llamado C. Este lenguaje fue específicamente diseñado e implementado por Ritchie como una herramienta para escribir sistemas operativos. Ritchie escribió el lenguaje C basándose en el trabajo previo de Thompson en el lenguaje llamado B. La reescritura en C permitió mudar UNIX a computadoras más potentes, como la PDP-11/45 y 11/70, producidas por DIGITAL.

El resto de la historia es conocida. UNIX se mudó fuera del laboratorio y en poco tiempo la mayoría de los fabricantes de computadoras estaban produciendo sus propias versiones: Sun Microsystems, Data General Hewlett Packard, IBM, DIGITAL, y otros.

---

[13] Recordemos que estos sistemas utilizan generalmente un esquema de scheduling del tipo *time-slice*.

[14] El nombre UNIX deriva de éste.

Actualmente existen dos tipos diferentes de UNIX, con diferentes filosofías.

BSD (Berkeley Software Distributions): el proyecto es liderado por la Universidad de California, en Berkeley. Fue el primer UNIX completo que se distribuyó. La mayoría de los sistemas comerciales tomaron el código de éste como punto de partida. Su filosofía es mantener un sistema operativo abierto con fines educativos.

SYSTEM V: sistema operativo con fines comerciales, que es manejado principalmente por la compañía AT&T.

En sus veinte años de vida, el sistema operativo UNIX se ha popularizado en diferentes aplicaciones científicas y de ingeniería. Su importancia, sin embargo, ha ido más allá de un impacto de mercado ya que UNIX fue el primer sistema operativo, de los ampliamente utilizados, en proporcionar una capacidad uniforme en diferentes computadoras, con diferentes arquitecturas.

Esta característica, junto con las implementaciones compatibles del lenguaje C, permite a los usuarios portar fácilmente sus aplicaciones de software entre diferentes computadoras. El deseo de preservar y extender estas ventajas llevó al enorme crecimiento de los *sistemas abiertos* (open systems).

Características de tiempo real de los sistemas UNIX

Los sistemas UNIX standard, ya sean de AT&T o Berkeley, no fueron diseñados con la problemática de tiempo real en mente. A continuación se enumeran las características más importantes a tener en cuenta en la implementación de una aplicación de tiempo real, junto con las herramientas de las que se dispone en UNIX para implementarlas.

i. Scheduling

La principal desventaja de los sistemas operativos UNIX para su utilización en aplicaciones de tiempo real es que están diseñados con schedulers que realizan tiempo compartido. Esto es así pues se desea realizar una justa distribución de recursos (en particular del tiempo de CPU) entre todas las tareas. Fundamentalmente, se busca un balance entre las necesidades de entrada/salida interactiva (buen tiempo de respuesta para los operarios en las terminales) y las necesidades de cómputo de los procesos de background (que requieren tanta CPU como puedan obtener). Aunque el comportamiento de los schedulers de tiempo compartido no está definido por ningún standard, estos hacen un muy buen trabajo en los sistemas operativos que llamamos *de propósitos generales*. Sin embargo, en el momento de diseñar una aplicación, debe tenerse en cuenta que dependiendo del tipo de UNIX, existen diferentes algoritmos de toma de decisiones [Bach, 1986].

Para obtener un funcionamiento de éste tipo, las prioridades de las tareas deben ser regularmente ajustadas (dependiendo de qué está haciendo esta tarea en cada momento). A las tareas que tienen largos períodos de espera se les aumenta la prioridad para que tengan mejor respuesta en el momento que termine dicha espera. Los procesos que monopolizan la CPU son penalizados por su comportamiento "antisociable".

Otra característica de estos sistemas es que las interrupciones son deshabilitadas periódicamente para poder hacer el tiempo compartido, para lo cual se utilizan los timers internos de la computadora. Este aspecto es crucial en el momento de diseñar una aplicación muy dependiente del hardware.

La única herramienta disponible, en los UNIX standard, para alterar el scheduling es la llamada al sistema (y el comando) *nice*. Éste define cuán *amigable* se desea que se comporte una cierta tarea, pudiendo tomar valores entre –20 y 20, siendo el valor por defecto 0. Pero esto define solamente las *intenciones* de la tarea. La última palabra la tiene el scheduler de tiempo compartido, que no está definido en ninguna norma.

Esta estrategia de scheduling no es, evidentemente, la más apropiada para aplicaciones de tiempo real, que deben tener alta prioridad permanentemente, y deben ejecutarse siempre que no haya otra tarea de mayor prioridad pendiente (esquema preemptivo basado en prioridades).

ii. Memory locking

Los sistemas UNIX modernos utilizan técnicas sofisticadas de memoria virtual, que hacen al sistema seguro y confiable, permitiendo la ejecución de programas que consumen mucha memoria, sin riesgos y sin deteriorar la performance del sistema. Además la memoria virtual elimina el problema de fragmentación que es normal en los sistemas con memoria física. Esta facilidad permite que sean

ejecutadas aplicaciones que consumen más memoria que la disponible, lo cual se logra trasladando páginas de memoria de tareas ociosas al disco rígido. Esta técnica se denomina *swapping*.

El problema es que el swapping consume mucho tiempo, y no hay forma de saber si un proceso ha sido desplazado al disco. Así, una aplicación que tiene un tiempo de respuesta del orden de los microsegundos puede llegar a emplear milisegundos en reinstalarse en la memoria antes de poder ser ejecutada.

Este problema se soluciona si el sistema operativo dispone de la facilidad de *trabar* una tarea en la memoria (o al menos la porción crítica de ésta). A esta facilidad se la denomina *memory locking*.

Los sistemas operativos UNIX no disponen, en general, de memory locking, excepto las últimas versiones de SYSTEM V.

iii. Relojes y timers

Los sistemas UNIX tienen solo un reloj utilizable por el usuario, con unas pocas herramientas para manejarlos. Por ejemplo, la llamada al sistema *time*, devuelve el número de segundos transcurridos desde las 00:00 horas del 1 de enero de 1970[15] (instante de tiempo conocido como "*the Epoch")*. La resolución es en segundos.

La llamada *gettimeofday* puede resolver hasta microsegundos. Pero la mayoría de los sistemas UNIX no pueden resolver menos que su tic de sistema operativo (tiempo que utiliza para hacer el tiempo compartido) que usualmente es de 100Hz[16], o sea que cada tic es de 10.000 microsegundos. Por lo tanto no puede obtenerse una granularidad más fina que esa. Esta granularidad se debe a que los relojes internos de la computadora están siendo utilizados para contar los intervalos de tiempo del time-slice. El sistema operativo recupera el control cada vez que finaliza dicha rebanada de tiempo, y puede utilizar las rebanadas para contar intervalos de tiempo. Pero nunca podrá medir intervalos menores que la rebanada.

La llamada *sleep* demora la ejecución de una tarea un número entero de segundos, pero no fracciones.

Los UNIX standard proveen a cada tarea la capacidad de medir intervalos de tiempo, asignando un timer de software a cada una de ellas (recordar la granularidad). Este timer es independiente para cada tarea. El manejo se hace a través de las llamadas *setitimer* y *getitimer*. El timer puede ser programado y al expirar dicho plazo el timer envía una señal a la tarea llamada SIGALARM. Para estos timers, las consideraciones de granularidad son las mismas que para *gettimeofday*.

iv. Entrada/salida

Cuando se habla de entrada/salida (I/O) del sistema de computadora, se está haciendo referencia al punto clave en el campo de las aplicaciones de tiempo real. No existen aplicaciones de tiempo real que no tengan que manejar algún tipo de interacción con el entorno. Sin el *mundo real* no existe el *tiempo real*.

Varios aspectos del UNIX pueden considerarse como problemas respecto de la entrada/salida en tiempo real.

En primer término, la entrada/salida en UNIX no es sincronizada. Esto significa que una operación de escritura, por ejemplo al disco rígido, no necesariamente retorna cuando los datos están realmente salvados en el medio físico. Este sistema operativo implementa un sistema de entrada/salida con buffers para mejorar la performance global, y realiza la escritura del medio en bloques de 4Kbytes, por ejemplo. Existe una llamada al sistema (*sync*) que realiza el volcado de todos los buffers, pero dicha operación no puede realizarse para cada proceso individualmente.

Otro aspecto importante es que la entrada/salida en UNIX es sincrónica[17], lo que significa que debe esperarse a que la lectura o escritura de un dispositivo termine para retornar de la función. Los accesos no pueden hacerse en paralelo con la aplicación, lo cual puede ser necesario a veces. Se dice que son *bloqueantes*. Existen algunas herramientas, como la opción O_NONBLOCK para la llamada *open*, que hacen que la lectura/escritura falle si no existen datos para leer/lugar para escribir. Pero la opción de suspender la lectura/escritura y proceder con el resto del programa no está disponible.

---

[15] *...in that time, there were no microwaves, compact discs, or ATM machines, and there was very little cable television. You can see why time doesn't go back any further than that.* [Gallmeister, 1995].

[16] Este número está definido en el kernel y puede cambiarse, dentro de un límite razonable.

[17] No debe confundirse sincrónico con sincronizado.

Por último, la geometría de los archivos no es controlable. En UNIX, un archivo no tiene forma, solo contenido. Un archivo es una secuencia de bytes no estructurada. No hay forma de saber cómo el sistema operativo distribuyó la información en el medio físico. Incluso, las diferentes implementaciones de UNIX utilizan distintas estrategias de distribución de los bloques. Para una aplicación de tiempo real sería deseable que los bloques correspondientes a un archivo de tiempo real fueran contiguos, para que el sistema no demore tiempo en encontrar los bloques adyacentes. Además sería deseable que una aplicación pudiera reservar un cierto espacio de disco para sus archivos. Estas facilidades no están disponibles en los UNIX standard.

Ante este panorama tan desalentador, ¿por qué intentar utilizar un sistema operativo UNIX para algo que no está específicamente diseñado?

La respuesta es simple. Los sistemas operativos UNIX son sistemas muy completos que incluyen sofisticadas interfaces gráficas y capacidades de networking, requisitos que no puede dejar de cumplir hoy en día ninguna aplicación de tiempo real. En cambio los sistemas operativos específicos de tiempo real generalmente no disponen de estas facilidades, o si disponen de algunas de ellas son limitadas y poco versátiles.

Pero no descartemos al UNIX todavía. Todavía existe una esperanza...

Los standard POSIX

Con la creciente popularidad del sistema operativo UNIX han aparecido varias versiones diferentes (algunas de ellas mencionadas anteriormente) que no son totalmente compatibles, lo que causa problemas al intentar portar aplicaciones. El IEEE, Institute of Electrical and Electronic Engineers creó, con el objetivo de solucionar este inconveniente, un comité de estandarización conocido técnicamente como P1003, para crear un standard que describa la interfaz de programación (*application program interface*) para los sistemas operativos tipo UNIX.

El resultado del trabajo de este comité son los documentos POSIX, *Portable Open Systems Interface,* que son estandarizados por ANSI e ISO. La importancia de POSIX radica en la portabilidad de código fuente de aplicaciones: permite llevar una aplicación de un sistema operativo POSIX a otro, simplemente recompilándolo.

El primero de dichos standard fue POSIX.1, y uno de los últimos el que es de mayor interés en el campo del tiempo real: POSIX.4.

POSIX.1

Esta primera versión del standard POSIX fue finalizada por la IEEE en 1988. Describe, básicamente, la interfaz del lenguaje C para llamadas a funciones del sistema operativo. Cubre solo un núcleo principal de funciones del sistema operativo, para que las aplicaciones sean portables entre sistemas. Pero son solo la estructura mínima de programación. Si bien no cubre todas las funciones necesarias para escribir una aplicación, proporcionan una base que facilita la portabilidad.

Actualmente casi todos los sistemas operativos tipo UNIX soportan esta primer versión de la interfaz POSIX. Hablar de UNIX y de POSIX.1 es casi lo mismo, por lo que no se entrará en detalle con este tema.

POSIX.4: Extensión de tiempo real

La creciente demanda de capacidad de procesamiento en tiempo real, y el desorden en el que las diferentes implementaciones atacaban el tema, llevó al comité a la producción de un nuevo standard que define las funciones básicas necesarias para la implementación de aplicaciones de tiempo real. En septiembre de 1993 se publicó la primer versión de éste.

Debe aclararse que POSIX.4 no es la solución a ningún problema de tiempo real. Solo se trata de un intento de organizar los esfuerzos que se están haciendo individualmente por agregar capacidad de tiempo real a los sistemas tipo UNIX. Además, la norma no dice cómo solucionar los problemas. Simplemente describe las interfaces para las funciones relativas al tiempo real [Gallmeister, 1995].

La funcionalidad POSIX.4 está siendo implementada en diferentes sistemas UNIX, pero en la actualidad no existe ninguno que cumpla las especificaciones en su totalidad.

Suponiendo que dicho sistema exista, mostraremos a continuación las mejoras que éste presentaría en lo que respecta a scheduling, memory locking, timers y entrada/salida, frente a un UNIX standard, que solo cumple la especificación POSIX.1 (los problemas de éstos fueron mencionados en el punto anterior).

i. Scheduling

El standard POSIX.4 incluye una serie de recursos que permiten el manejo preciso de la estrategia de scheduling utilizada por el sistema. Es posible establecer una política preemptiva, basada en prioridades y asignar prioridades a cada tarea. Además la norma define una extensión llamada round-robin, que es similar al esquema preemptivo, con la diferencia que las tareas de igual prioridad se ejecutan haciendo tiempo compartido.

ii. Memory locking

Al contrario que los UNIX standard, los UNIX que cumplen la especificación POSIX.4 tienen definido un set de llamadas al sistema que permiten trabar procesos en la memoria. En el caso que la memoria no sea suficiente para almacenar todos los procesos importantes, posee la facilidad de trabar solo las partes críticas de dichas tareas (si es que pueden ser localizadas). Las llamadas disponibles son *mlockall* y *munlockall* para procesos completos, y *mlock* y *munlock* para partes de ellos. A estas últimas debe especificársele la zona de direcciones a trabar.

iii. Relojes y timers

POSIX.4 define, además de los recursos standard, timers adicionales, mayor resolución (nanosegundos) y la posibilidad de utilizar mecanismos diferentes de SIGALARM para notificar que un timer ha expirado. Para obtener la hora absoluta, esta norma define además un reloj de tiempo real para cada tarea. Y presenta también una versión mejorada de *sleep* llamada *nanosleep*, con resolución de nanosegundos.

iv. Entrada/salida

En la normalización POSIX.4 se definen un nuevo tipo de archivo: el archivo sincronizado. Esta variante se obtiene simplemente declarando un flag al invocar un *open*. Se define también la opción de entrada/salida asincrónica, lo que permite invocar una lectura/escritura en paralelo y seguir con el procesamiento. El sistema operativo notifica al concluir la operación.

Debe recordarse que, en este aspecto, cada implementación de UNIX debe analizarse por separado, ya que cada una se encuentra en diferentes instancias de la normalización POSIX.4. Incluso algunos desarrolladores no se han mostrado interesados en hacer que sus sistemas evolucionen por esta vía.

LINUX

El sistema operativo Linux es el resultado de una colaboración internacional, encabezada por Linus Torvalds, el autor original. Es una reimplementación gratuita de la especificación POSIX.1, con extensiones SYSTEM V y BSD. Esto significa que se parece mucho a UNIX, pero no proviene del mismo código. Este sistema operativo está disponible tanto en formato binario, como en código fuente.

El único sistema operativo que Linus Torvalds tenía disponible en sus épocas de estudiante, era MINIX [Tanenbaum, 1988]. Éste es un sistema tipo UNIX, muy simple, ampliamente utilizado como herramienta de enseñanza. Impresionado por sus características, decidió escribir su propio sistema operativo, tomando a UNIX como modelo. Comenzó su trabajo con una PC con procesador Intel 386. En poco tiempo unió sus esfuerzos con otros estudiantes a través de las entonces modernas redes de computadoras, utilizadas en aquellos tiempos solo por la comunidad científica. En poco tiempo, Linux se convirtió en un sistema operativo.

Los derechos de autor pertenecen a Linus Torvalds, y es de libre distribución bajo la General Public License (GPL) de la fundación Free Software Foundation, Cambridge, Massachusetts, la cual entre otros puntos especifica que se puede copiar, cambiar y distribuir libremente, pero no se pueden imponer restricciones a futuras distribuciones y el código fuente debe estar siempre disponible. Una copia de la GPL se incluye en el *Apéndice I*. Esto significa que Linux no es de dominio público, ni *shareware*. Es software libre (*freeware*). Debe aclararse que *libre* se refiere a la disponibilidad de las fuentes y no al costo monetario. Es perfectamente legal vender una distribución de Linux, mientras se adjunten las fuentes de éste.

Las licencias de los utilitarios y programas que se distribuyen junto con Linux pueden variar. La mayoría proviene del proyecto GNU (*GNU's Not Unix*) de la Free Software Foundation, que produce software de muy alta calidad bajo la licencia GPL. Un ejemplo interesante es el compilador de lenguaje C conocido como *gcc*, que es considerado actualmente como uno de los mejores paquetes de software de este tipo, y que muchos fabricantes recomiendan para sus sistemas operativos. El software escrito específicamente para este trabajo de tesis adhiere también a la licencia GNU, cuyo texto completo se presenta en el *Apéndice I.*

Gracias a la compatibilidad POSIX.1, Linux ha sido mudado con éxito a múltiples plataformas, además de la PC x86 o Pentium. Actualmente se encuentran disponibles versiones para plataformas Motorola 680x0 (Amiga, Atari y máquinas VME), Alpha (DEC), SPARC (Sun), Power PC, etc.

En los últimos años, una gran cantidad de grupos de trabajo ha realizado aportes, obteniéndose un producto completo y confiable, que además tiene la ventaja de estar en permanente desarrollo. Desde la versión 1.0, el kernel de Linux ha superado la etapa de testeo *beta.*

Debe aclararse que cuando se menciona al Linux se hace referencia, generalmente, al *kernel* o núcleo del sistema operativo (a cargo de Torvalds y cía.) más un grupo de utilidades base para dicho sistema, proveniente de diversas fuentes. Algunas compañías han lanzado productos que incluyen el kernel junto con grandes cantidades de software adicional, lo que constituyen las diferentes *distribuciones* de Linux (Red Hat, Slackware, etc.)

Un aspecto importante a tener en cuenta, mas allá del económico, es que Linux está desarrollado utilizando un modelo abierto y distribuido, en lugar de uno cerrado y centralizado, en oposición a la gran mayoría del software comercial. Esto significa que la versión actual de desarrollo es siempre pública para que cualquiera pueda utilizarla. El resultado es que cada vez que una nueva versión es lanzada, con nuevos aspectos funcionales y, por supuesto, errores, gracias a esta filosofía abierta dichos errores son rápidamente localizados por los usuarios y autores (algunas veces en horas) e instantáneamente un gran número de desarrolladores trabaja para solucionarlos.

Por el contrario, el modelo cerrado y centralizado lleva generalmente a que solo una persona de un equipo esté trabajando en una parte del proyecto, y solo se pone disponible una nueva versión cuando se supone que está funcionando correctamente. Esto lleva en general a largos períodos de tiempo entre versiones y correcciones de errores, y por lo tanto la evolución es lenta.

Se mencionan a continuación algunas de las principales características de Linux, que lo ponen a la altura de los más modernos sistemas operativos.

Multitareas: varios programas pueden ejecutarse simultáneamente.

Multiusuario: varios usuarios pueden utilizar el mismo sistema de computadora simultáneamente.

Multiplataforma: puede ejecutarse en diferentes CPUs, no solo Intel.

Multiprocesador: disponible para plataformas Intel y SPARC.

Ejecuta en modo protegido en el 386: tiene protección de memoria, para que una aplicación no pueda inutilizar el sistema.

Carga de ejecutables por demanda: solo se leen del disco las partes del programa que son utilizadas.

Kernel modular: permite la modificación de éste mientras se encuentra en funcionamiento.

Páginas compartidas de memoria entre ejecutables: varios procesos pueden utilizar la misma memoria, lo que incrementa la velocidad y reduce el consumo. Paginado (no swapping) de memoria a una partición separada del filesystem. Capacidad de agregar área de paginado durante la ejecución, hasta 2Gbytes.

Memoria unificada para programas y cache de disco: puede utilizarse la memoria libre como cache, o el cache puede reducirse para ejecutar programas grandes.

Librerías linkeables dinámicamente (DLLs) o estáticamente.

Hace dumps de memoria para análisis post-mortem, permitiendo utilizar un debugger luego que un programa terminó anormalmente.

Emulación de coprocesador en el kernel.

Compatible con POSIX, System V y BSD al nivel de fuentes y con SCO, SVR3 y SVR4 al nivel de ejecutables.

Pseudo-terminales, con soporte de gran cantidad de teclados, y múltiples consolas virtuales

Soporte para tipos standard de filesystems, incluyendo MINIX, Xenix, System V, OS/2, MS-DOS, Windows NT y 95, hasta 4Tbytes, con nombres hasta 255 caracteres.

Múltiples protocolos de red, como TCP/IP, Netware, Appletalk, SMB, IPX, NetBEUI, DDP, AX.25, X.25, IPV6, etc.
Entorno gráfico X Windows completo.
Gran variedad de device drivers para dispositivos standard.
Gran variedad de programas de aplicación, sobre todo científicos.

En estas condiciones, Linux se ha transformado en una alternativa apropiada para realizar desarrollos en el ámbito de un laboratorio, ya que dispone de todas las ventajas de un UNIX más una gran variedad de herramientas de desarrollo y toneladas de información disponible[18]. Se dispone, además, de una comunidad de usuarios enorme que, con el desarrollo de las redes informáticas, utiliza los grupos de discusión para el intercambio de información. Si bien no se dispone de un soporte técnico, las soluciones a múltiples problemas y las respuestas a todo tipo de preguntas llegan generalmente más rápidamente por esta vía.

¿Linux es POSIX?

El sistema operativo Linux cumple el standard POSIX.1 (en realidad dicho standard fue tomado como el punto de partida para su diseño), pero una versión que cumpla completamente las especificaciones del POSIX.4 no está disponible.

Los sistemas POSIX.1 tienen características que los hacen muy atractivos para adquisición de datos y control (como por ejemplo que los dispositivos externos de hardware sean presentados al programa de usuario como *archivos secuenciales* normalizados –esta técnica se describirá en detalle en el capítulo siguiente). Sin embargo, una gran cantidad de los servicios descriptos en el capítulo anterior solo está disponible en los sistemas POSIX.4 (y son indispensables para la implementación de sistemas optimizados, como se vio anteriormente).

Algunas de las facilidades POSIX.4 están implementadas experimentalmente, como por ejemplo *nanosleep()*, y otras ni siquiera están siendo ensayadas, como la reserva de espacio de disco para archivos de tiempo real.

Existen varias modificaciones que pueden hacerse al kernel standard y paquetes de software que pueden agregarse para acercar al Linux a la especificación POSIX.4.

Scheduling: Existe actualmente un scheduler que cumple las especificaciones del POSIX.4, el cual requiere que el kernel sea modificado y recompilado.

Comunicaciones entre procesos: Los mecanismos de comunicación de los sistemas tipo System V (*System V IPC*) son parte del kernel standard del Linux. Agrega semáforos, queues y shared memory a los procesos de comunicación usuales del UNIX.

Memory locking: Existe un paquete que debe ser compilado con el kernel, que cumple las especificaciones del POSIX.4.

I/O sincronizada: Puede ser obtenida con los comandos *sync* y *fsync*. No existen los archivos sincronizados.

Reserva de espacio para archivos: No disponible.

Clocks y timers de tiempo real: Existe la posibilidad de agregar al kernel standard una mejora del sistema de timers, que aumenta la resolución en tiempo.

Debe aclararse que todos estos agregados al sistema operativo Linux son fruto de diferentes grupos que trabajan independientemente, sin que en la actualidad exista un proyecto formal que intente orientar el desarrollo por esta vía. Torvalds y su grupo no ha demostrado, hasta ahora, interés en el tema. Su objetivo principal es mantener un buen sistema operativo de tiempo compartido. Por lo tanto debe descartarse, por ahora, la posible existencia de un sistema Linux que cumpla la especificación POSIX.4.

[18] Toneladas, en caso que se desee imprimirla.

Real-Time Linux: un concepto diferente

Real-Time Linux (RTLinux) es un sistema operativo en el cual un pequeño kernel de tiempo real coexiste con el kernel POSIX.1 de Linux [Barabanov et al. 1997]. Esta estructura es deseable ya que permite utilizar los servicios sofisticados y altamente optimizados de un sistema standard de tiempo compartido como Linux, y al mismo tiempo permitir que tareas de tiempo real sean ejecutadas en un ambiente predecible y de baja latencia.

Esta es una variante que ataca el problema desde otro punto de vista. La solución está generalmente en la respuesta a la siguiente pregunta: ¿Cuáles son las tareas de mi aplicación que *realmente* necesitan prestaciones de tiempo real? ¿Necesito archivos de tiempo real? ¿Necesito redes en tiempo real?

La respuesta, en el caso de la implementación de sistemas de adquisición de datos y control para experimentos, es generalmente la misma: las tareas de tiempo real son las que se encargan de interactuar con el hardware del laboratorio. Las redes y archivos de tiempo real podrían utilizarse en algunos casos, pero no son indispensables. Las interfaces de usuario pueden soportar demoras de milisegundos sin problemas. Y generalmente los datos almacenados son procesados y analizados off-line. Lo que sí es indispensable es que la sincronización del experimento, las señales de alarma y el acceso al hardware se ejecuten en un entorno predecible.

Tiempo atrás, los sistemas operativos de tiempo real eran primitivos. Se trataba simplemente de un ejecutivo con pocas herramientas más que un juego de rutinas. Pero hoy en día las aplicaciones de tiempo real requieren acceso a redes TCP/IP, display gráfico y entorno windows, archivos, bases de datos, y otros servicios que no son ni primitivos ni simples. Una solución es agregar esos servicios que no son necesariamente de tiempo real al kernel básico de tiempo real, como se ha hecho en productos como VXworks o QNX. Una segunda solución es modificar un kernel standard y hacerlo completamente preemptivo. Esta es la filosofía encarada por los autores de RT-IX, entre otros.

RTLinux es una tercera opción, en la cual un ejecutivo de tiempo real ejecuta un kernel que no es de tiempo real como su tarea de más baja prioridad, utilizando una capa de máquina virtual[19], para lograr así que el kernel standard sea preemptivo. En RTLinux todas las interrupciones son manejadas inicialmente por el kernel de tiempo real y son pasadas a la tarea Linux sólo cuando no hay tareas de tiempo real para ejecutar. Para minimizar los cambios que deben hacerse al kernel de Linux, se le agrega una emulación del hardware de control de interrupciones. Así, cuando Linux *deshabilita* las interrupciones, el software de emulación encola las interrupciones que el kernel de tiempo real ha generado.

Las tareas de tiempo real se comunican con las de Linux a través de *FIFOs* y memoria compartida. Desde el punto de vista del programador, las colas son similares a los dispositivos de caracteres de UNIX, accedidos a través de los mecanismos POSIX read, write, open, close e ioctl (llamadas al sistema). La memoria compartida se accede utilizando las llamadas al sistema standard, como *mmap*.

RTLinux utiliza al Linux para el arranque, los device drivers, acceso a redes, filesystems, control de procesos y para la administración de los módulos del kernel, que son utilizados para hacer que el sistema de tiempo real sea fácilmente expansible y simple de modificar.

Las aplicaciones de tiempo real son tareas que se incorporan al kernel a través de módulos (este mecanismo se describe en detalle en el próximo capítulo), mientras que las tareas del Linux standard se encargan del almacenamiento de los datos, display, acceso a la red, y cualquier otra tarea que no sea de tiempo real. Un caso típico de aplicación es la adquisición de datos. Las aplicaciones de este tipo se componen generalmente de una tarea de tiempo real que atiende interrupciones o hace un polling del dispositivo adquisidor. Esta tarea necesita tener los tiempos de acceso acotados. Ésta envía los datos a través de una cola a una tarea Linux (que no es de tiempo real) que se encarga del almacenamiento y del display de éstos. En este caso el kernel de tiempo real se encarga de la tarea con estrictos requerimientos de tiempo, mientras que Linux, con su buena performance promedio y buffering de entrada/salida, se encarga del resto.

Desde el punto de vista del programador, una tarea de tiempo real se programa de la siguiente forma. Como primera medida se escribe un programa en C, utilizando un grupo de funciones disponibles en una *interfaz de programación* (API, Aplication Program Interface) de tiempo real. Entre las más significativas pueden citarse: crear una tarea de tiempo real, hacerla periódica, ejecutarla, suspenderla, obtener la hora de un reloj de tiempo real, esperar cierto tiempo. Este programa se compila como un módulo del kernel

[19] En inglés, *virtual machine layer*.

(para lo cual deben tenerse en cuenta ciertas consideraciones que se detallan en el próximo capítulo) y luego se instala. La comunicación desde el Linux con esta tarea, que está continuamente ejecutándose en el background, se hace por medio de FIFOs, que se pueden acceder desde un programa en C o desde el mismo sistema operativo, como archivos standard.

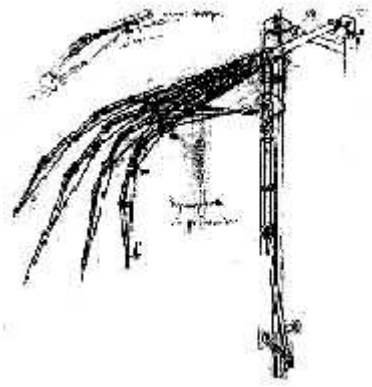
Como puede verse, la metodología es simple, aunque no muy versátil. Para mas detalles acerca de la implementación puede consultarse la bibliografía [Barabanov, 1997]. En el próximo capítulo se detalla con profundidad el mecanismo de programación de las tareas de tiempo real.

Resumiendo, en este capítulo se presentaron las características principales de los sistemas Unix, junto con un análisis de su inserción en el campo de los sistemas de tiempo real. Se presentó también el sistema Linux, tanto en su versión POSIX.1, como en su versión de tiempo real.

En el próximo capítulo se analiza con mayor detenimiento el kernel POSIX.1 de Linux, su estructura modular y sus device drivers, elementos que proporcionan una muy potente herramienta en un entorno de control y adquisición de datos. Además se profundiza en el funcionamiento de la extensión de tiempo real RTLinux, para finalmente proceder con un análisis comparativo entre las performances de ambos sistemas, sobre un caso real de aplicación.

We're back to the times when men were men and wrote their own device drivers.

Linus Torvalds



Capítulo III
Linux y el acceso al hardware

Al mencionarse el proyecto *Linux* se hace referencia, esencialmente, al equipo de personas encargadas del mantenimiento y desarrollo del *kernel* o *núcleo* del sistema operativo. Es de uso general el llamar Linux al conjunto de programas de aplicación usualmente distribuidos con dicho kernel, pero eso no es correcto. El Linux propiamente dicho es el kernel, resultado de una colaboración internacional dirigida por Linus Torvalds, el autor original. A este esfuerzo se han sumado muchos otros grupos de desarrollo que han incorporado nuevos paquetes de software, disponibles a través de las distribuciones comerciales del sistema operativo.

La intención de este capítulo es presentar una introducción a algunos aspectos internos del kernel del sistema operativo Linux, que son de interés en el desarrollo y utilización de nuevas interfaces de hardware. Tener claro el funcionamiento del kernel ayuda a desarrollar aplicaciones más eficientes. Este es el punto clave que hace al Linux preferible frente a otros sistemas operativos: la información está disponible.

Con esta intención en mente se procede con una presentación de los aspectos fundamentales del kernel o núcleo de Linux (que, como se mencionó en el capítulo anterior, cumple las especificaciones POSIX.1), junto con su funcionamiento básico y estructura. Se introduce también la utilización de *módulos* y la estructura de sus *device drivers*. A continuación se profundiza en el funcionamiento de la extensión RTLinux, que permite incorporar algunas facilidades de tiempo real al sistema. Finalmente se procede con un análisis comparativo de las performances de ambos sistemas en un caso real de aplicación que implica el contacto del sistema con hardware externo.

Este capítulo se trata solamente de una revisión general, acompañada de ejemplos prácticos de aplicación, que debería ser suficiente para comprender el funcionamiento de los drivers y módulos empleados en la implementación del experimento Mössbauer, que es presentada en el *Capítulo V*. Para obtener información más detallada, puede consultarse la bibliografía disponible [Rubini, 1998] [Egan et al. 1992] o la información on-line [Linux Lab Project].

El kernel

En un sistema UNIX coexisten varios procesos que se encargan de atender diferentes tareas simultáneamente. Cada proceso necesita disponer de algunos recursos del sistema, ya sea capacidad de cómputo, memoria, acceso a la red, etc. El *kernel* es la porción de código ejecutable que tiene a su cargo la administración de dichos recursos. Las tareas del kernel pueden ser clasificadas de la siguiente forma:

Administración de procesos: el kernel tiene a su cargo la creación y destrucción de procesos (*dispatcher*) y el manejo de sus conexiones con el exterior (entrada/salida). Además se encarga de los diferentes métodos de comunicación entre dichos procesos (señales, pipes) y de la rutina más crítica del sistema, el *scheduler*. O sea que el kernel es el encargado de implementar la coexistencia de varios procesos sobre una CPU única.

Administración de memoria: otra de las tareas del kernel es la de construir un espacio de direccionamiento virtual para cada uno de los procesos, dentro de los recursos limitados de memoria.

Sistema de archivos: el sistema operativo UNIX tiene una muy fuerte interdependencia con el concepto de *filesystem*; casi todos los elementos del sistema pueden ser tratados como archivos. El kernel está encargado de crear un sistema de archivos estructurado sobre dispositivos de hardware no estructurado. La abstracción resultante es intensamente utilizada a lo largo de todo el sistema. Linux soporta varios tipos de sistemas de archivos, o sea que dispone de diferentes formas de organizar los datos en el medio físico.

Control de dispositivos: casi todas las operaciones de un sistema acceden a un dispositivo físico. Con excepción del procesador y la memoria, todas las operaciones de control de dispositivos son efectuadas por una porción de código que es específica de dicho dispositivo. Dicha porción de código es llamada *device driver*[20]. El kernel debe incluir un device driver para cada periférico presente en el sistema, desde el disco duro hasta el teclado. Este aspecto del kernel es el punto fundamental de este capítulo.

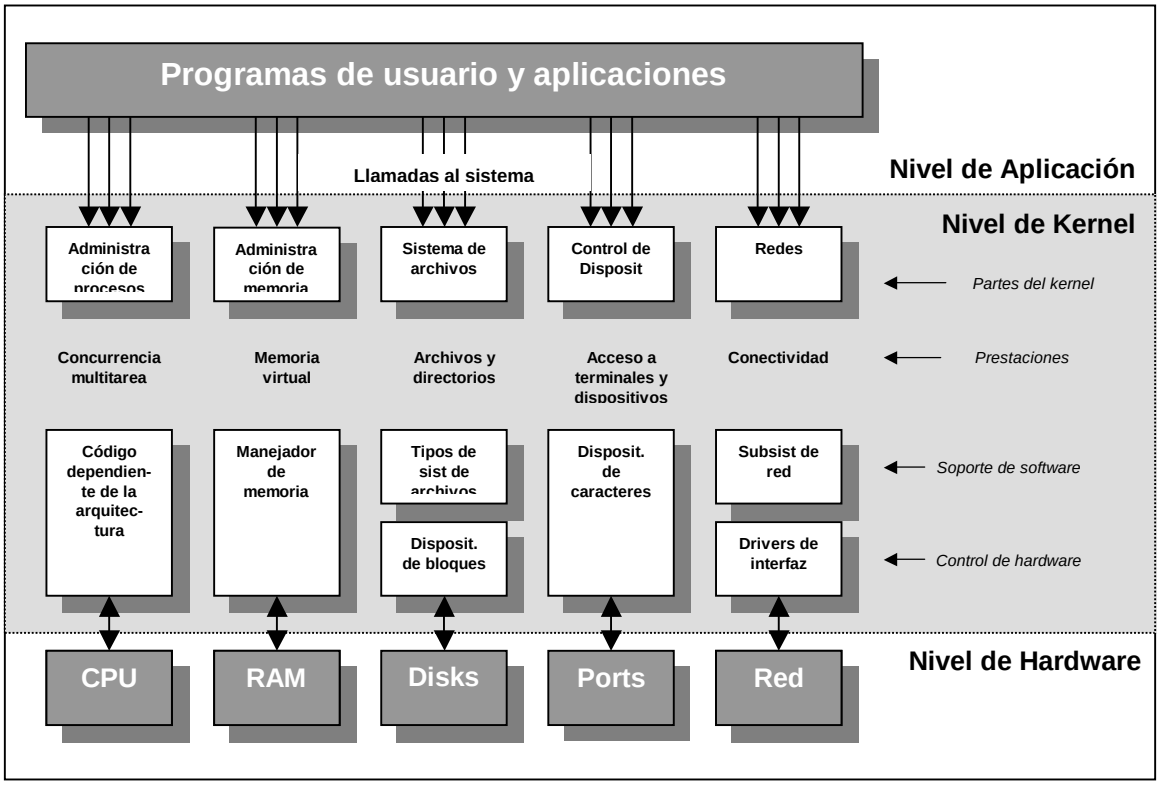


Figura III.1: Estructura interna del kernel de Linux

[20] El término *device driver* podría ser traducido como *manejador de dispositivo*, pero la amplia difusión del primero puede llevar a confusiones en el caso de utilizarse esta última denominación.

Networking: el acceso a redes debe ser manejado por el sistema operativo pues la mayoría de las operaciones relacionadas con la red no son específicas de un proceso. Los paquetes deben ser recolectados, identificados y redireccionados antes que un proceso pueda ocuparse de ellos. El kernel se encarga del transporte de paquetes de datos entre programas e interfaces de red. Adicionalmente, la resolución de nombres y el redireccionamiento son implementados por el kernel.

El kernel es el primer código ejecutable que se instala en la memoria de la PC al iniciarse el sistema. Cualquier falla de éste se reflejará como una salida de operación del equipo. En la *figura III.1* puede verse un diagrama en bloques de los diferentes componentes de un kernel de Linux.

El código fuente completo del kernel de Linux está disponible para los usuarios, y es parte de la distribución standard de este sistema operativo. Esta es la principal característica que diferencia al Linux de los sistemas comerciales. Un usuario de Linux puede hacer modificaciones directamente en el kernel del sistema operativo.

Módulos

Otra de las ventajas del Linux es su capacidad de incorporar líneas de código al kernel mientras éste se está ejecutando. Se le puede agregar funcionalidad al sistema operativo mientas éste está en funcionamiento. Cada parte de código que puede ser agregado al kernel se denomina *módulo*. Cada módulo es una porción de código objeto (sin linkear) que se linkea dinámicamente al kernel que está ejecutando utilizando el comando *insmod* provisto por el sistema. De la misma manera, el módulo puede ser descargado con el comando *rmmod*.

Debe recordarse que los sistemas operativos tipo UNIX hacen una diferencia muy importante entre la memoria del sistema y la memoria de usuario. Este es uno de los principales motivos de su robustez. El kernel se ejecuta en el espacio de memoria reservado para el sistema. Los programas de aplicación residen en un área de memoria diferente, y el código del kernel es accedido por estas aplicaciones a través de *llamadas al sistema* o *system calls*. De esta forma las aplicaciones pueden acceder a los recursos del sistema, ejecutando sólo el código seguro del kernel.

Cada módulo es una porción de código que se incluye en el kernel, y por lo tanto ejecuta en la memoria del sistema, sin ningún tipo de restricciones. Por lo tanto debe procederse con cuidado en el momento de escribir y depurar un módulo, ya que éste puede ocasionar graves daños al sistema.

A modo de ejemplo, se muestra a continuación la carga y descarga de un módulo que solo contiene las dos rutinas indispensables: la de inicialización (que se procesa al ejecutar *insmod*) y la de limpieza (que se procesa al ejecutar *rmmod*).

Nótese la utilización del comando *printk*. Debe tenerse en cuenta que el código de kernel no puede utilizar las librerías standard de Linux, ya que estas no están disponibles en su espacio de memoria, y por lo tanto no pueden ser linkeadas al kernel. No podría, por ejemplo, utilizarse *printf* ya que pertenece a la librería `libc` del sistema. Sólo puede utilizarse el C básico y algunas pocas funciones disponibles en el kernel, tales como *printk, outb, inb*, etc.

Sea el siguiente archivo hello.c:

```
#define MODULE
#include<linux/module.h>

int init_module(void)         { printk("Hello, world\n"); return 0; }
void cleanup_module(void)     { printk("Goodbye, world\n"); }
```

Para la compilación, instalación y eliminación del módulo se prosigue como se muestra a continuación:

```
root# gcc -c hello.c
root# insmod hello.o
Hello, world
root# rmmod hello.o
Goodbye, world
root#
```

Nótese que, por razones de seguridad en el sistema, es necesario tener privilegios de *superusuario* para poder instalar un módulo en el kernel.

Con estos simples pasos hemos insertado código dentro del kernel de Linux mientras éste estaba en funcionamiento y luego lo hemos eliminado, sin necesidad de recompilar el kernel, detener el sistema y reiniciarlo para que el nuevo kernel se sitúe en la memoria.

Linux POSIX.1: Device drivers

Un driver puede ser visto como la capa de software que se encuentra entre la aplicación y el dispositivo de hardware. Éste provee una interfaz normalizada de software para dicho dispositivo. En Linux, un device driver puede ser linkeado dinámicamente como un módulo, lo que facilita mucho la etapa de desarrollo del mismo.

¿Pero por qué hacer que el código de manejo de un dispositivo se encuentre dentro del kernel y no directamente en la aplicación de usuario que debe acceder a él? El motivo principal está dado por las características multiusuario y multitarea del sistema operativo. No puede permitirse a los usuarios el acceso directo al hardware, pues pueden presentarse problemas de concurrencia. Además el acceso a las interrupciones de hardware sólo puede hacerse a través del kernel.

En realidad, sí es posible escribir código de aplicación que acceda directamente al hardware[21], pero debe existir alguna señalización en el sistema que indique que ese dispositivo está siendo utilizado, para evitar el acceso por parte de otra aplicación. Es el kernel quien puede proveer este mecanismo. Además, si se utiliza el código de acceso directo en los programas de aplicación, existe una porción muy importante de este código que debe reescribirse idénticamente cada vez que se desea tener acceso al dispositivo. Incluso, si el dispositivo fuera modificado, todos los programas de aplicación que lo utilizan deberían modificarse.

Si se utiliza un driver linkeado con el kernel para acceder al dispositivo, la concurrencia es manejada por éste y el código es único para todas las aplicaciones. Además, una modificación del hardware implica la modificación de sólo su device driver, ya que los programas de aplicación sólo acceden a él a través de su interfaz de software standard (read, write, open y close). Además pueden utilizarse las interrupciones externas de la PC, que de otra forma sería imposible.

Existen varios tipos de device drivers, siendo de interés en este caso sólo los correspondientes a la primera clasificación:

Device drivers de caracteres (char device drivers): un dispositivo de caracteres es un dispositivo que puede ser accedido como un archivo, y su driver debe encargarse de implementar ese comportamiento. Este driver incluye generalmente las llamadas de sistema *open, close, read* y *write*. La consola y las puertas paralelo son ejemplos de char devices, que pueden ser imaginados como dispositivos *secuenciales*. Los devices de caracteres son accedidos a través de nodos del sistema de archivos (filesystem nodes), como */dev/tty1* o */dev/lp1*. Estos nodos son archivos especiales que representan el punto de entrada al código de kernel correspondiente al dispositivo: al device driver. De esta forma, los dispositivos de caracteres son percibidos desde el punto de vista del usuario como archivos ordinarios, que pueden ser abiertos, leídos, escritos y cerrados. La única diferencia es que en un archivo ordinario se puede volver atrás, a buscar el dato anterior, cosa que no puede hacerse en una puerta serie, por ejemplo. Ésta sólo puede ser accedida secuencialmente. Éste es el tipo de devices que interesan a este proyecto. Con un driver de caracteres puede representarse tanto una puerta paralelo, como una puerta serie, una interfaz GPIB, una placa con conversor A/D, etc. En el resto del capítulo se procederá con la confección de un device driver de caracteres que permita utilizar la puerta paralelo como entrada/salida de 8 bits a la PC, con fines de reutilizar este módulo dentro del sistema de control a implementar. Debe aclararse que las distribuciones standard de Linux contienen un device driver para la puerta paralelo (accesible a través de /dev/lp*) pero su funcionalidad está orientada fundamentalmente a la utilización de impresoras, con el handshake correspondiente, y utilizando el port como sólo de salida.

Device drivers de bloques (block device drivers): la principal diferencia entre un dispositivo de bloques y uno de caracteres es que un dispositivo de bloques puede contener un filesystem, como por ejemplo un disco duro. En la mayoría de los casos sólo puede accederse a éste de a múltiplos de una unidad mínima llamada bloque, que es

[21] Esta técnica puede ser una buena herramienta como primera etapa en el desarrollo.

usualmente un kilobyte de datos. Al igual que los dispositivos de caracteres, son accedidos a través de nodos en el filesystem. Los drivers para este tipo de dispositivos son complejos y no serán tratados en éste capítulo.

Interfaces para red: este tipo de dispositivos es un caso especial de dispositivos de caracteres que no tienen un nodo asociado en el filesystem, y cuya comunicación con el kernel es completamente diferente a las anteriores. En lugar de implementarse un *read* o *write*, el kernel utiliza funciones relacionadas con la transmisión de paquetes.

A continuación se presenta una introducción a los device drivers de caracteres, utilizando como ejemplo la utilización de la puerta paralelo como entradas/salidas digitales para la PC.

Estructura de los device drivers de caracteres

Este tipo de device drivers es el de mayor interés en el entorno de desarrollo planteado. Para ejemplificar los puntos importantes del diseño y la estructura de éstos, se procederá con la implementación de un driver mínimo que permita utilizar la puerta paralelo standard de la PC como entradas/salidas digitales. Es deseable la disponibilidad por parte del usuario de la interrupción externa de hardware asociada a dicha puerta, asignando una tarea que será ejecutada ante la existencia de una señal externa a la PC.

El primer punto a tener en cuenta en el diseño de un driver es la funcionalidad deseada. Debe especificarse perfectamente la forma en que se desea que el dispositivo sea visto a través del driver. En el caso del driver para la puerta paralelo, como primera aproximación al driver completo que se presenta al final del capítulo, sería deseable que la puerta pueda ser manejada como un archivo, con su nodo de entrada en el directorio /dev. Una forma adecuada de trabajar con la puerta paralelo sería que ésta pueda ser abierta (*open*), escrita (*write*) y cerrada (*close*), en una primera instancia. Este comportamiento se asemeja al de un archivo secuencial.

El código asociado a la *apertura* del dispositivo debe encargarse (en forma transparente para el usuario) de inicializar el dispositivo en un estado deseado y de comunicar al sistema que dicho dispositivo esta en uso. Esto puede implicar un grupo de comandos que se ejecutarán conjuntamente en el momento que el usuario invoque la llamada al sistema *open()*.

El código de *escritura* debería ser capaz de recibir del usuario una palabra de ocho bits y enviarla al port de salida de la puerta paralelo.

El código de *cerrado* debe resetear el dispositivo a un estado deseado y comunicar al sistema que éste ha sido liberado por la aplicación, para que otra aplicación pueda utilizarlo.

En otras palabras, sería deseable que el dispositivo paralelo fuera manejado como un archivo secuencial, por ahora de sólo escritura. La lectura del dispositivo y el manejo de la interrupción externa se posponen para más adelante, cuando el funcionamiento del driver sea aclarado.

En conclusión, si llamamos a nuestro nuevo driver *Daq*, es deseable que un usuario pueda escribir un programa en C que incluya las siguientes líneas de código:

```
int fd;
fd = open(/dev/Daq, O_RDWR);
write(fd,0x00);
close(fd);
```

con el objeto de llevar a "0" lógico todas las salidas del port paralelo, tratando al dispositivo como si fuera un archivo secuencial ordinario, utilizando un entero como descriptor de archivo o *file descriptor*.

Para proceder con la implementación debemos adentrarnos primero en la estructura genérica de un driver de caracteres. El funcionamiento básico esta esquematizado en la *figura III.2*. La primer diferencia que se encuentra con un programa en C standard es la ausencia de la función *main()*. Un driver es sólo un conjunto de funciones que son invocadas al hacerse la llamada al sistema correspondiente. Por ejemplo, en nuestro driver Daq, es necesario que la función *Daq_open()* residente en el kernel, sea llamada al ejecutarse *open(/dev/Daq)* desde un programa de usuario. Para que el programa de usuario mostrado anteriormente pueda correrse, el driver instalado en el kernel debe contener además las funciones *Daq_close()* y *Daq_write()* correspondientes.

La asignación de funciones a las llamadas a sistema correspondientes se hace por medio de una tabla que para el driver debe presentarse como una estructura del tipo *file_operations* (ver listado). Esta tabla asigna una función para cada llamada al sistema, debiendo ser llenada por NULL en el caso de funciones no implementadas. En nuestro caso implementaremos sólo la función de escritura (posición número 3 de la tabla, a la que asignaremos *Daq_write()*), apertura (posición 8 *Daq_open()*) y cerrado (posición 9 *Daq_close()*). Las posiciones en la tabla son fijas y deben respetarse. Los nombres de las funciones asignadas pueden ser cualquiera, pero es recomendable utilizar un prefijo propio (en este caso Daq) para que no interfiera con símbolos preexistentes del kernel.

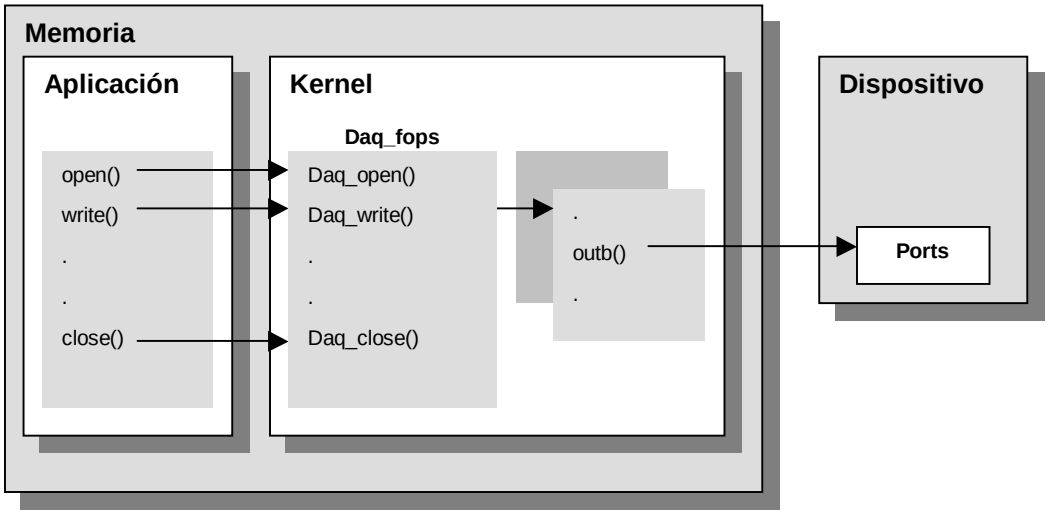


Figura III.2: Llamadas al driver desde una aplicación

Por supuesto que no deberán olvidarse las funciones *init_module()* y *cleanup_module()* (estos nombres deben respetarse) para la instalación y desinstalación del módulo en la memoria de kernel, tal como se describió anteriormente. La primera tiene la misión de registrar el driver frente al kernel con el nombre *"Daq"* y con la tabla *Daq_fops,* para que éste pueda ubicar las funciones en la memoria de sistema. Para ello utiliza la función *register_chrdev()*. La segunda realiza la operación inversa utilizando la función *unregister_chrdev()*. El significado de DAQ_MAYOR se comenta en la sección siguiente. Esta constante, junto con otras como DAQ_BUSY, DAQ_FREE y DAQ_OUT, se definen en el archivo *Daq.h*, que es incluido en la primera línea. Este es un procedimiento standard, que facilita la organización del driver. Además, el archivo *Daq.h* puede ser incluido en la aplicación que utiliza el driver, para poder utilizar dichas definiciones. Dicho archivo se denomina de encabezado o *header file*.

```
#include<Daq.h>
Daq_busy=LIBRE;

static struct file_operations Daq_fops={
       NULL,
       NULL,
       Daq_write,
       NULL,
       NULL,
       NULL,
       NULL,
       Daq_open,
       Daq_close
       }

static int Daq_open() {
       if(Daq_busy==DAQ_BUSY) return -EBUSY;
       Daq_busy=DAQ_BUSY;
       printk("DAQ: abierto.");
       outb(0x00,DAQ_OUT);   /* Inicialización del hardware */
       return 0;
       }

static void Daq_close() {
       outb(0x00,DAQ_OUT);   /* Reset del hardware */
```

```
        Daq_busy=LIBRE;
        printk("DAQ: cerrado.");
        }

static int Daq_write() {
        char c;
        const char *buff;
        c=get_user(buff);
        outb(c,DAQ_OUT);
        return 0;
        }

int init_module() {
        register_chrdev(DAQ_MAYOR,"Daq",&Daq_fops);
        return 0;
        }

void cleanup_module() {
        unregister_chrdev(DAQ_MAYOR,"Daq");
        }
```

La función *Daq_write()* utiliza *get_user()* para tomar el byte que envía el programa de aplicación y *outb()* para enviar dicho byte al port correspondiente de la puerta paralelo.

Este listado es una versión muy resumida del driver, no operativa. La versión completa de *Daq.c*, junto con el archivo *Daq.h* y el *Makefile* correspondiente se presentarán al final del capítulo, una vez que se le haya agregado la capacidad de lectura y de atención de la interrupción.

Resumiendo, este módulo se compila y se instala en el kernel, registrándose debidamente. Este queda dormido hasta que alguna de las funciones es invocada. Para utilizar el dispositivo, un programa de usuario primero invoca *open()* sobre el device, lo cual produce la inicialización e inhibición del control de acceso o *lock* del dispositivo. Luego puede escribirlo con la función *write()* cuantas veces desee, para finalmente liberarlo con la llamada al sistema *close()*. Exactamente como si se tratara de un archivo.

Este nuevo device driver ya puede ser instalado, pero no puede ser utilizado aún. Falta el archivo especial o *nodo* que lo incluya en el filesystem y que nos permita acceder a dicha porción de kernel.

Nodos

Para que un device driver pueda ser accedido por un programa de aplicación de usuario, el driver necesita tener un punto de entrada dado por un archivo especial llamado *filesystem node*. En Linux estos archivos, por una cuestión de orden, se encuentran ubicados en el directorio */dev*. Allí puede encontrarse una gran cantidad de estos archivos especiales, relacionado cada uno de ellos con un driver standard del sistema.

Estos nodos son archivos que contienen solamente dos números enteros, llamados el número MAYOR y el número MENOR, que pueden verse en las columnas 4 y 5 de un listado. Por ejemplo:

```
MiLinux:~# ls -l /dev/lp*
crw-rw----    root     daemon     6,   0 Apr 27  1995 lp0
crw-rw----    root     daemon     6,   1 Apr 27  1995 lp1
crw-rw----    root     daemon     6,   2 Apr 27  1995 lp2
```

En este caso se listan los nodos correspondientes a las interfaces paralelo disponibles en Linux. Las tres primeras columnas son los atributos del nodo (por quien puede ser utilizado y quien es el 'dueño'). La cuarta columna indica el número MAYOR (6) y la siguiente el número MENOR (1 al 3). La última columna es el nombre. Nótese que en la primer columna, además de los permisos usuales, aparece como primer letra una 'c' indicando que se trata de un nodo para un device driver de caracteres.

El número MAYOR es un entero único en el sistema que identifica unívocamente un device driver. El número MENOR identifica una subclase dentro del mismo driver. En este caso existe un único driver de puerta paralelo, accedido con MAYOR. Con leves modificaciones internas dentro del driver, observando qué nodo fue accedido, se hace referencia al primero, segundo o tercer puerto de la PC y se accede a un código diferente, utilizando MENOR.

Para poder utilizar nuestro driver *Daq*, debemos crear un nodo para éste. Para ello necesitamos un número MAYOR que no este siendo utilizado por el sistema. En el archivo /usr/include/linux/major.h están los números MAYOR reservados por el sistema (allí puede verse que el 6 está reservado para la puerta paralelo standard como vimos anteriormente). Desde el número 30 en adelante están libres para ser utilizados por el usuario. Podemos tomar el 31. En nuestro caso no es necesario utilizar un número

MENOR pues este driver tiene una sola versión para la única interfaz paralelo disponible. En la elaboración del driver no contemplaremos la posibilidad de múltiples puertas paralelo por una cuestión de claridad. Se adoptará MENOR igual a 0.

Una vez seleccionados los números MAYOR y MENOR que serán utilizados, el nodo debe ser creado en el directorio */dev*. Linux provee un programa que se encarga de ello: *mknod*. Este comando debe ser ejecutado como *superusuario*, por razones obvias.

```
MiLinux:~# mknod /dev/Daq c 31 0
```

Este comando se encarga de crear el nodo Daq en el directorio /dev, con número MAYOR 31 y MENOR 0, para acceder a un char device driver, indicado por c.

Al producir un listado puede verse el nuevo nodo:

```
MiLinux:~# ls -l /dev/Daq
crw-r--r--   root     root      31,   0 Aug  5 07:42 Daq
```

El número MAYOR debe ser informado al kernel en el momento del registro del driver. Por ello se utilizó la variable DAQ_MAYOR en el procedimiento de registro del driver listado anteriormente. Esta constante tiene asignado el valor entero 31 en el archivo Daq.h.

Luego de ser creado el nodo de acceso, el programa en C que accede a *Daq* puede ser utilizado, si previamente el driver fue linkeado con el kernel utilizando *insmod*.

Un device driver completo

Quedaron pendientes dos puntos en la implementación del device driver. El primero es el agregado de la capacidad de lectura de la puerta paralelo, y el segundo la facilidad de capturar la interrupción de dicha puerta para la utilización como interrupción externa a la PC. Esta interrupción está disponible en el conector de la puerta paralelo standard.

Con respecto a la lectura de la puerta paralelo el procedimiento es similar al de escritura. Resta solamente incluir una nueva función en *Daq_fops* en la posición 2 llamada *Daq_write()*, y proveer el código para dicha función en el cuerpo del driver. Esta función debe utilizar la función *inb()* para leer el port correspondiente a la puerta paralelo y la función *put_user()* para enviar el byte al programa de aplicación que solicitó la lectura.

El procedimiento de lectura de la puerta paralelo está condicionado a que dicho modo de funcionamiento esté disponible en la versión presente de hardware, lo cual es casi siempre cierto en los nuevos modelos de PC. Para mayor información respecto de la programación de la puerta paralelo se recomienda la consulta de la bibliografía [Eggenbretch, 1991] [Stewart, 1994].

Para la utilización de la interrupción correspondiente a la puerta paralelo (en general la interrupción 7 para la primera de ellas) se procede de la siguiente forma. En primera instancia, debe solicitarse dicha interrupción al kernel en la rutina de apertura del dispositivo *Daq_open()*. Para ello se utiliza la función del kernel *request_irq()* como se presenta en el listado siguiente, indicando cuál es la función que debe ejecutarse cuando la interrupción se haga presente. En nuestro ejemplo la llamaremos *Daq_irq()*. El segundo paso es proveer el código para la nueva función *Daq_irq()*.

En el driver completo que se lista a continuación se utilizo la interrupción externa para sincronizar la lectura del port. Para ello se utilizó la facilidad de *colas de espera* disponible en el kernel. Cuando el programa de aplicación ejecuta una lectura con *read()* de la puerta paralelo, el driver manda este proceso a 'dormir' con *interruptible_sleep_on()*, hasta que llega una interrupción que lo 'despierta' ejecutando *wake_up()*. Este es un procedimiento muy potente para implementar la sincronización de la PC con dispositivos externos. La implementación puede verse en el listado que se presenta a continuación.

Código del driver

```
// Daq.c
// Device Driver para utilizar la puerta paralelo como in/out
// Crear el nodo con: mknod /dev/Daq c 31 0

#include <linux/module.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <linux/kernel.h>
```

```
#include <linux/major.h>
#include <linux/sched.h>
#include <linux/malloc.h>
#include <linux/ioport.h>
#include <linux/fcntl.h>
#include <linux/delay.h>
#include <time.h>
#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>

#include "Daq.h"

int Daq_busy = DAQ_FREE;                    // Semáforo
struct wait_queue *Daq_wait_q;              // Declaración de la cola de espera
```

```
static void Daq_irq(void)
{
       wake_up(&Daq_wait_q);                // Despierto a read
}
```

```
static int
Daq_open(struct inode * inode, struct file * file)
{
       int codigo;

// Verifico que el MINOR sea 0
unsigned int minor = MINOR(inode->i_rdev);

// Verifico que el semáforo este en verde y lo pongo en rojo
if (minor != 0) return -ENODEV;
       if (Daq_busy == DAQ_BUSY) return -EBUSY;
       Daq_busy = DAQ_BUSY;

// Solicito la interrupción
       codigo = request_irq(DAQ_IRQ, (void *)Daq_irq, “Daq”);
       if(codigo){
              printk(“ERROR: no se puede utilizar la interrupción 7\n”);
              return codigo;
       } else {
              outb(inb(0x21)&(~DAQ_IRQ),0x21);
              outb(0x20,0x20);              // Habilito máscara de interrupciones
       }
return 0;
}
```

```
static int
Daq_read(struct inode * inode, struct file * file, char * buf, int count)
{
       unsigned char testvalue;

       if(count!=1) return -EINVAL;                // Leer de a uno
       interruptible_sleep_on(&Daq_wait_q);        // Espero la interrupción
       testvalue = inb(DAQ_IN);                    // Toma el byte del port
       put_user(testvalue, buf);            // Lo envía a la aplicación
       return 0;
}

static int
Daq_write(struct inode * inode, struct file * file, const char * buf, int count)
{
       char c;
       const char *temp;
       temp=buf;

       while(count>0){
              c=get_user(temp);             // Toma el próximo byte
              outb(c,DAQ_OUT);              // Lo envía al port
              count--;
              temp++;
       }

       return temp-buf;
}
```

```
static void
Daq_release(struct inode * inode, struct file * file)
{
       Daq_busy  = DAQ_FREE;                // Semáforo en verde
}
```

```
static struct file_operations Daq_fops = {
              // Solo open, read, write, y close implementadas
```

```
NULL,          // seek
Daq_read,      // read
Daq_write,     // write
NULL,          // readdir
NULL,          // select
NULL,          // control
NULL,          // mmap
Daq_open,      // open
Daq_release    // release
};
```

```
int
init_module( void)
{
       outb(0, DAQ_OUT);                                          // Reset
       if (register_chrdev(DAQ_MAYOR, "Daq", &Daq_fops)) {
         printk("ERROR: init_module ha fallado\n");               // Error
         return -EIO;
       }
return 0;
}
```

```
void
cleanup_module( void)
{
if (Daq_busy) printk("WARNING: Daq ocupado, eliminación postergada\n");
if (unregister_chrdev(DAQ_MAYOR, "Daq") != 0)
              printk("ERROR: cleanup_module ha fallado\n");
}
```

Archivo de encabezado

Se lista a continuación el archivo de definiciones e *includes* (header file) para el driver de puerta paralelo. Este archivo debe incluirse en el programa de aplicación si se desean utilizar las constantes, como por ejemplo DAQ_OUT.

```
// Daq.h
// Definiciones para Daq.c

#include <sys/ioctl.h>

#define DAQ_MAYOR     31
#define DAQ_OUT       0x378
#define DAQ_IN        0x378
#define DAQ_BUSY      1
#define DAQ_FREE      0
#define DAQ_IRQ       7
```

Makefile

Se presenta a continuación la configuración necesaria para emplear el utilitario *make* en la compilación del driver. Incluye varias opciones de optimización correspondientes al compilado de módulos. Para más detalle puede consultarse la página de manual del compilador *gcc*. Esta configuración se presenta aquí solo con fines prácticos, para quienes estén familiarizados con el utilitario *make*.

```
# Makefile para el driver Daq

CC=gcc
CFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer \
         -fno-strength-reduce -pipe -m486
MODCFLAGS = -DMODULE -DMODVERSIONS -DCPU=486 -DEXPORT_SYMTAB -D__KERNEL__ -DLINUX

all:  Daq.o
Daq.o: Daq.c Daq.h
       $(CC) $(MODCFLAGS) $(CFLAGS) -c -o Daq.o Daq.c
```

Programa de aplicación

Se presenta el siguiente programa en C para la utilización de Daq, que genera una onda cuadrada en el pin 0 de la puerta paralelo, utilizando para la generación de la base de tiempo la función nanosleep().

```
// Generación de una onda cuadrada con la puerta paralelo

#include <stdio.h>
#include <fcntl.h>
#include <time.h>
#include "Daq.h"

void main(){

        int fd;
        short number = 0x1111;
        unsigned char buffer;
        struct timespec tiempo;             // Estructura definida en time.h.

        tiempo.tv_sec = 0;                  // Tiempo en segundos
        tiempo.tv_nsec = 100 * 1000000;     // + nanosegundos

        fd = open("/dev/Daq",O_RDWR); // Abro la puerta paralelo
        if (fd<0) {
                perror("ERROR: la puerta paralelo no pudo ser abierta");
                exit(-1);
        }
        for(i=0;i++;i<1000){
                write(fd,"1",1);
                nanosleep(&tiempo,NULL);
                write(fd,"0",1);
                nanosleep(&tiempo,NULL);
        }
        close(fd);                          // Cierro la puerta paralelo
}
```

Este conjunto de programas se utilizará al final del capítulo para medir la performance de Linux al generar tareas con base de tiempo, sin la extensión de tiempo real.

A continuación se prosigue con algunos aspectos más avanzados del diseño de drivers, como son las técnicas de debugging y algunas funciones más disponibles para su confección.

Depurado del código

El depurado del código del driver es un paso fundamental en el proceso de elaboración. Dicho código debe estar perfectamente optimizado y libre de errores, ya que pasará a formar parte del kernel.

Al ubicarse el driver en la memoria reservada para el sistema, no es posible accederlo utilizando utilitarios de debugging ordinarios, tal como se haría con una aplicación en la memoria de usuario.

Un procedimiento utilizado para el depurado es la utilización de la función *printk* dentro del cuerpo del driver, tal como se procede utilizando *printf* para depurar programas standard y detectar errores. Estos invocaciones adicionales a printk deben eliminarse en la versión definitiva del driver, para que no signifiquen una sobrecarga inútil en el momento de la puesta en funcionamiento.

Una técnica utilizada para eliminar dichas llamadas es condicionarlas a la existencia de una constante previamente declarada, como por ejemplo DAQ_DEBUG. Así puede procederse de la siguiente forma:

```
...
outb(ox20,ox20);
#ifdef DAQ_DEBUG
printk(“Byte enviado.\n”);
#endif
...
```

La sentencia *#ifdef* es para el compilador. Le indica que si DAQ_DEBUG está definida, que incluya la línea siguiente, hasta la sentencia *#endif*. Si no, es ignorada en la compilación.

Pueden así compilarse dos versiones diferentes del driver: una explícita para debugging y una silenciosa para su utilización, una vez que fue depurado el código. Esto evita tener dos versiones de código diferentes, una con y otra sin *printk*, con los errores que ello podría implicar.

Si bien esta es una herramienta primitiva de depurado, es la única disponible en este caso. Se recomienda su utilización intensiva, ya que el óptimo funcionamiento de los drivers es crucial para la performance del sistema.

La función de control ioctl()

Con la misma jerarquía que las funciones *read* y *write* para el driver, existe una función adicional (posición sexta en la tabla) llamada *ioctl()*. Su objetivo es servir como la 'navaja suiza' del driver. Se utiliza para realizar todas las tareas que no pueden ser realizadas por el resto de las llamadas al sistema, tales como configuración durante la ejecución, cambio de estado del dispositivo, reset, etc.

Desde la aplicación puede invocarse esta función con un entero como parámetro. Este procedimiento generará una llamada al sistema que invocara, en nuestro caso a *Daq_ioctl()*, previamente declarada en *Daq_fops*. Dentro de *Daq_ioctl()* pueden ejecutarse diferentes acciones, dependiendo del entero que se pasó como parámetro. Esta selección se hace generalmente con un *switch*.

Para una descripción más amplia de la utilización de esta función, como así también para un más extenso análisis de las técnicas de diseño, se recomiendan varios textos en el capítulo de bibliografía consultada [Rubbini, 1998].

Como intentó mostrarse a lo largo de éste punto, la confección e instalación de device drivers en el entorno POSIX.1 del sistema operativo Linux es una tarea relativamente simple, dependiendo, por supuesto, de la funcionalidad y versatilidad deseada para dicho driver.

El entorno presentado, con herramientas standard de simple manejo y distribución gratuita, se presenta como muy adecuado para el desarrollo de aplicaciones y la automatización de experimentos. Un usuario con nivel medio de conocimientos puede introducirse en el kernel y modificar su funcionamiento. Esta es la gran diferencia entre Linux y el resto de los sistemas operativos: las vías para modificar el kernel se le *facilitan* al usuario, ya que se lo considera algo normal, dentro de la filosofía de Linux. No es necesario ser un *hacker* ni disponer de oscuros secretos para adentrarse en el sistema operativo. Esto es lo que ha disparado la evolución de Linux a límites insospechados en su momento.

Debe aclararse que la introducción presentada hasta aquí sólo hace referencia a las herramientas utilizadas en el *Capítulo V* en la elaboración del driver para la placa *μDAQ*. En el capítulo de bibliografía consultada pueden encontrarse algunas recomendaciones para extenderse en el tema.

A continuación se comentará otra filosofía de diseño, fuera del espectro de los standard POSIX, que permite incorporar funcionalidad de tiempo real al sistema operativo Linux. Debe aclararse que no se trata de una implementación de los standard POSIX.4 de tiempo real, sino una implementación diferente, basada en el concepto de máquina virtual.

RTLinux: Linux en tiempo Real

En el capítulo anterior se describió brevemente el funcionamiento del sistema RTLinux. A continuación se procede con un análisis más detallado del mismo.

RTLinux es una modificación hecha al kernel POSIX.1 de Linux, que permite el manejo de tareas con estrictos requerimientos de tiempo. Fue implementado por Victor Yodaiken y Michael Barabanov [Barabanov et al. 1997], y el código fuente esta disponible a los usuarios, siguiendo la filosofía de Linux.

RTLinux introduce un pequeño ejecutivo de tiempo real, que tiene al Linux POSIX.1 como su tarea de más baja prioridad. La intención es hacer uso de los servicios sofisticados y altamente optimizados para comportamientos promedio del sistema Linux, que utiliza una estrategia de tiempo compartido, mientras que permite la existencia de funciones de tiempo real, que operan en un ambiente predecible y de baja latencia.

Este ejecutivo garantiza un servicio no interrumpido a sus tareas de tiempo real, que disponen de las siguientes facilidades:

Asignación de prioridades relativas a las tareas.
Incorporación de tareas de tiempo real como módulos cargables.
Periodicidad de tiempo real para las tareas.
Comunicación entre procesos (IPC).
Funciones asociadas a IPC definidas por el usuario.
Técnica de scheduling definida por el usuario.

Debido a la existencia del ejecutivo de tiempo real, la funcionalidad del sistema Linux no se ve alterada. Los servicios de networking, interfaz gráfica, herramientas de desarrollo y entorno de programación, mantienen su funcionalidad.

Las tareas de tiempo real no pueden hacer uso de las llamadas al sistema para requerir servicios del Linux POSIX.1. En caso de ser necesario, deben comunicarse con tareas de Linux, que hacen las llamadas por ellos, a través de dos mecanismos disponibles:

Memoria compartida: las tareas de RTLinux y de Linux pueden compartir la memoria de kernel, sin gozar de los beneficios de sincronización o estructuras de datos de alto nivel.

FIFOs de tiempo real: son un tipo de pipes con características especiales. La sincronización de lectura/escritura es invisible, pueden asociársele funciones (handlers) y la administración de los FIFOs no interfiere con el acceso en tiempo real de éstos.

Debido a su prioridad frente a las tareas de Linux, un proceso RTLinux que requiera mucha CPU podría producir una sobrecarga del sistema, hasta el punto de congelar el funcionamiento de Linux. Este comportamiento es necesario, ya que debe garantizársele a las tareas de tiempo real la capacidad de cumplir con sus restricciones de tiempo. Por ello debe hacerse una medición cuidadosa de la CPU requerida por cada tarea.

Las tareas RTLinux son tareas propias del kernel. Esto tiene sus beneficios.

Tienen privilegios de kernel, y por lo tanto acceso directo al hardware.

Pueden ser módulos cargables, lo que facilita su instalación mientras el sistema está en funcionamiento.

Comparten el espacio de kernel, lo que ahorra tiempo en el *context-switch*.

No son víctimas del paginado de memoria que consume cantidades indeterminadas de tiempo.

Considerando las características citadas, es lógico suponer que una aplicación que hace uso de las facilidades de RTLinux se compone de dos partes. La primera es un grupo de tareas de tiempo real, instaladas como módulos. La segunda es un grupo de tareas de Linux que se encargan del registro de datos, acceso a la red y todas aquellas tareas que puedan dificultar el cumplimiento de los *deadlines* de las anteriores. La comunicación entre ambas se realiza a través de FIFOs o memoria compartida. Los FIFOs se presentan al usuario Linux como device drivers de caracteres standard, que pueden ser accedidos utilizando las llamadas al sistema de POSIX (read/write/open/close/ioctl). La memoria compartida es accedida utilizando la llamada *mmap* de POSIX.

Para que esta estrategia pueda ser utilizada, el kernel POSIX.1 de Linux debe ser modificado, tal que todas las interrupciones sean inicialmente capturadas por el kernel RTLinux. Estas interrupciones son pasadas al Linux POSIX.1 solamente cuando no hay tareas de tiempo real para ejecutar. Para minimizar los cambios que deben hacerse al kernel, se implementa una emulación por software del hardware de control de interrupciones. Los autores proveen un "patch" para el kernel de Linux, que debe ser recompilado e instalado, previo a la carga de los módulos.

La programación de las tareas de tiempo real (que luego serán instaladas como módulos) se hace utilizando una API (Application Programming Interface) suministrada a tal efecto. Esta incluye funciones para la inicialización y eliminación de tareas de tiempo real, sincronización y control de las tareas, declaración y utilización de los FIFOs, sincronización del scheduler, y operación en punto flotante. Estas funciones se describen brevemente a continuación.

Inicialización y eliminación de tareas

rt_task_init	crea una tarea de tiempo real y su stack.
rt_task_delete	elimina una tarea de tiempo real y su stack.

Sincronización y control de las tareas

rt_task_make_periodic	señala una tarea de tiempo real para la ejecución, y la hace periódica.
rt_task_wait	detiene la ejecución hasta el comienzo de un nuevo período.
rt_task_suspend	suspende una tarea periódica; rt_task_make_periodic debe ser invocada para reiniciarla.
rt_task_wakeup	señala una tarea no periódica para la ejecución, o reinicia una tarea suspendida en forma no periódica.

Declaración y utilización de los FIFOs

rtf_create_fifo	crea un FIFO de tiempo real
rtf_create_handler	asocia una función con un FIFO; dicha función es llamada cada vez que se lee o escribe dicho FIFO.
rtf_destroy	elimina un FIFO.
rtf_get	lee de un FIFO.
rtf_put	escribe a un FIFO.
rtf_resize	modifica el tamaño de un FIFO

Sincronización del scheduler

request_RTirq	asocia una función con una interrupción determinada.
free_RTirq	libera dicha interrupción.

Operación en punto flotante

rt_use_fp	habilita el uso de operaciones de punto flotante dentro de la tarea.

Además de estas facilidades, se dispone también de diferentes estrategias de scheduling (algunas de ellas aportadas por la comunidad de usuarios) que permiten optar entre una estrategia *rate monotonic* o EDF. También existe una implementación de semáforos y varias implementaciones de servicios del sistema. Estos paquetes se presentan también como módulos que pueden ser cargados durante el funcionamiento del sistema, lo que permite, por ejemplo instalar el scheduler EDF, correr una aplicación bajo test, y luego reemplazar dicho scheduler por el *rate monotonic*. Este tipo de modularidad es una herramienta invaluable durante la etapa de desarrollo de una aplicación.

Ejemplo simple de aplicación

A los efectos de resumir lo detallado anteriormente, se presenta a continuación una aplicación muy simple, en la cual se desea conmutar uno de los bits de salida de la puerta paralelo a intervalos constantes, a fin de obtener una onda cuadrada de niveles TTL y frecuencia conocida. Este mismo ejemplo se utilizará a continuación en este capítulo para comparar la performance de RTLinux frente al Linux POSIX.1.

Básicamente, este módulo instala dos tareas periódicas de tiempo real: una setea el bit y la otra lo resetea. Ambas tareas tienen el mismo período (40.000 μseg), para obtener una onda cuadrada en la salida del port paralelo (en este caso direccionado en la dirección de entrada/salida 378 hexadecimal de la PC).

La secuencia *init_module* es ejecutada una vez en la instalación del módulo (hecha por medio del comando *insmod*, como se vio al comienzo del capítulo) y *cleanup_module* al ser removido (con *rmmod*).

Init_module inicializa las dos tareas de tiempo real, asociándoles una función que setea o resetea el bit según el parámetro que se le pase. A continuación instruye al scheduler que, periódicamente despierte a dicha tarea, que estará esperando debido a la utilización de la función *rt_task_wait*. *Cleanup_module* elimina dichas tareas del scheduler.

Nótese la utilización de *rt_get_time* para componer el instante exacto en el cual debe comenzar la ejecución periódica de las tareas. Tanto el período, como el instante de comienzo, tienen una resolución de 1 μseg.

El Linux POSIX.1, en este caso, es utilizado para editar el archivo fuente, compilarlo con las herramientas standard e instalarlo como un módulo dentro del kernel. Luego de ello, el Linux POSIX.1 no realiza función alguna en esta aplicación. Las dos tareas de tiempo real estarán ejecutando fuera de su rango de visión, y las tareas de Linux no pueden acceder a las de RTLinux, ya que no se ha definido ningún mecanismo de comunicación (como FIFOs o memoria compartida). La única interacción posible desde Linux es la remoción del módulo, lo que interrumpe la ejecución de las tareas de tiempo real.

```
// rt_sqr.c
// Genera una onda cuadrada en la salida 0 del port paralelo

#include <./rt.h>     // Includes necesarios para compilar un módulo en rt.h
#define LPT 0x378     // Dirección del byte de salida del port paralelo standard
de la PC

RT_TASK tarea1;       // Declaración de dos tareas de tiempo real
RT_TASK tarea2;
```

```
int periodo = 40000;  // Período de la onda cuadrada: 40mseg.

// Esta función se le asigna a ambas tareas, una envía t=1 y la otra t=0
void funcion(int t) {
       while (1) {
              outb(t, LPT);          // Escribe al port paralelo
              rt_task_wait();        // Espera su próximo período
       }
}

// Inicialización del módulo: define las dos tareas de tiempo real e indica al
// scheduler que las ejecute periódicamente
int init_module(void)
{
       RTIME ahora = rt_get_time();                // Obtiene la hora.

       rt_task_init(&mytask, funcion, 1, 3000, 4); // Esta tarea setea el bit,
                                                   // con prioridad 4
       rt_task_init(&mytask2, funcion, 0, 3000, 5);// Esta tarea resetea el
                                                   // bit, con prioridad 5

       // Ambas tareas se ejecutan periódicamente con un offset de periodo/2
       rt_task_make_periodic(&tarea1, ahora + 3000, periodo);
       rt_task_make_periodic(&tarea2, ahora + 3000 + ( periodo/2 ), periodo);

       return 0;
}

// Limpieza del módulo: elimina las dos tareas de tiempo real
void cleanup_module(void)
{
       rt_task_delete(&tarea1);
       rt_task_delete(&tarea2);
}
```

Linux POSIX.1 vs. RTLinux

Hasta ahora se han analizado dos posibilidades bien diferentes para interaccionar con el hardware en Linux, cada una con sus ventajas y desventajas.

Linux POSIX.1 provee un mecanismo standard que permite presentar el hardware al usuario de una forma normalizada (a través de sus device drivers). La ventaja de utilizar esta estrategia es que el programador de una aplicación puede desconocer completamente el funcionamiento interno del hardware (handshake, timing, forma de transferencia de los datos) siempre que disponga del device driver correspondiente a dicha pieza de hardware. Esto permite escribir aplicaciones normalizadas y portables, que solo utilizan los mecanismos standard de POSIX para acceder a los recursos. En caso de no disponerse del device driver adecuado, se mostró que el mecanismo de elaboración de éste no es complicado, dependiendo siempre de la complejidad del hardware asociado. La desventaja más notable de esta forma de trabajo es que no se dispone de capacidad de procesamiento en tiempo real. Las tareas que acceden al hardware estarán condicionadas por el funcionamiento del scheduler de tiempo compartido del Linux. No por ello debe descartarse completamente la utilización de esta metodología de trabajo; la capacidad de procesamiento de los equipos actuales hace que en la gran mayoría de los casos pueda encontrarse una solución por esta vía. Un ejemplo de esto es el reciente lanzamiento por parte de National Instruments de una versión de LabView para Linux POSIX.1, en la cual la empresa provee el entorno de trabajo, mientras que recomienda la utilización de los múltiples device drivers de dominio público disponibles para casi todo el hardware de instrumentación y control standard.

Por otro lado, RTLinux posee la capacidad de tiempo real requerida en algunos casos, pero como pudo verse, los programas de aplicación no son tan simples como una secuencia de read y write. Se requiere un conocimiento por parte del programador del hardware utilizado, ya que no se dispone del equivalente a un device driver. El programa de aplicación del lado de Linux POSIX.1 trabaja con read y write sobre los FIFOs, pero la verdadera aplicación de adquisición y control está del lado de RTLinux, donde no se dispone de las facilidades POSIX.

Por lo tanto, antes de decidir qué entorno de trabajo utilizar deben conocerse en detalle los requerimientos de tiempo de la aplicación a desarrollar, como así también la capacidad de respuesta de cada uno de los sistemas.

A efectos de caracterizar el comportamiento del Linux en el campo del tiempo real, se diseñaron dos experimentos. El primero evalúa los tiempos de respuesta a eventos sincrónicos (generación de intervalos de tiempo a partir del timer interno de la PC), y el segundo a eventos asincrónicos (atención de

interrupciones externas). Las tareas asociadas a estos eventos consistieron en activar y desactivar bits del port paralelo standard.

Se ensayaron el Linux POSIX.1 con sus device drivers vs. la extensión RTLinux, instalados en una PC con un procesador Pentium operando a 120MHz. y para distintas condiciones de carga del sistema. Los diferentes tipos de carga ensayados en ambos experimentos fueron:

Sin carga: se eliminaron todos los procesos de background de la inicialización. Solamente se mantuvo el kernel en la memoria. Para conseguir este estado se utilizó el comando *kill* manualmente sobre todos los procesos no indispensables para el funcionamiento (sendmail, cron, httpd, etc.).

Carga de background: se mantuvieron los procesos de inicialización correspondientes al estado init 3, incluyendo httpd, lpd, smbd, nmbd, sendmail, inetd, crond, syslogd y otros. Se eliminaron todas las tareas de usuario.

Carga de disco rígido: además de los procesos de background de ejecutó el comando *ls –lR* repetitivamente durante el tiempo de la medida. Este comando hace un listado recursivo de la estructura completa de directorios del filesystem, y requiere un acceso intensivo al disco.

Carga de red: además de los procesos de background de ejecutó el comando *ping –f,* dirigido hacia otro hosts de la red local. Esta tarea envía paquetes TCP/IP intensivamente, solicitando al máximo los recursos de red.

Carga completa: corresponde a la ejecución simultanea de las tareas de disco rígido y de actividad de red.

Carga de cálculo: además de los procesos de background de ejecutó un programa de usuario que ejecuta cálculos intensivos en punto flotante.

Experimento 1: Generación de una onda cuadrada de 20 milisegundos con el port paralelo

La intención de este experimento es comparar el comportamiento del sistema operativo RTLinux frente al Linux POSIX.1, cuando se desea sincronizar tareas utilizando una base de tiempo generada con los relojes internos de la PC. Para ello se tomó como referencia una onda cuadrada obtenida seteando y reseteando uno de los bits de salida de la puerta paralelo de la PC. El cálculo del período esta a cargo de Linux POSIX.1 y RTLinux, respectivamente.

Para generar la onda cuadrada con el Linux POSIX.1 se utilizó el driver de puerta paralelo, cuyo funcionamiento se describió en detalle al comienzo del capítulo. Éste permite abrir la puerta paralelo como si se tratara de un archivo, utilizando el comando *open*, y enviar una palabra de 8 bits a la salida con el comando *write*, desde un programa en C escrito por el usuario. Este mecanismo de interacción con los dispositivos es típica de los sistemas POSIX.1.

De esta forma, el siguiente programa de usuario (compilado con el comando *gcc –o file file.c*) genera una onda cuadrada en el bit 1 de la puerta paralelo standard de la PC, si previamente fue instalado el driver correspondiente, accedido a través de /dev/lpv.

```
#include<...>
fd = open("/dev/lpv",O_RDWR;
while(1){
write(fd,"1",1);
nanosleep(&tiempo,NULL);
write(fd,"0",1);
nanosleep(&tiempo,NULL);
}
```

Nótese la utilización de la función *nanosleep* para la medición de tiempos, función que corresponde al standard POSIX.4, y que ya está implementada en Linux. El período de la onda cuadrada se eligió en 40 milisegundos, ya que dicha función tiene una resolución mínima de 20 milisegundos (ver granularidad, en el punto anterior).

Para obtener la onda cuadrada con RTLinux se utilizó el módulo descripto anteriormente con dos tareas sincronizadas, que se repite a continuación por comodidad.

```
//
// rt_sqr.c
// Genera una onda cuadrada en la salida 0 del port paralelo
//
```

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/cons.h>
#include <asm/io.h>
#include <linux/rt_sched.h>

// Dirección del port paralelo
#define LPT 0x378

RT_TASK tarea1;
RT_TASK tarea2;

// Período de la onda cuadrada
int periodo = 40000;

// Esta función se le asigna a ambas tareas, una envía t=1
// y la otra t=0
void funcion(int t) {
   while (1) {
          outb(t, LPT);          // Escribe al port paralelo
          rt_task_wait();        // Espera su próximo período
   }
}

// Inicialización del módulo: define las dos tareas
// y las pone a correr
int init_module(void)
{
   RTIME ahora = rt_get_time();         // Obtiene la hora

   rt_task_init(&mytask, funcion, 1, 3000, 4);
                // Esta tarea setea el bit, con prioridad 4
   rt_task_init(&mytask2, funcion, 0, 3000, 5);
                // Esta tarea resetea el bit, con prioridad 5

   // Ambas tareas se ejecutan periódicamente con
   // un offset de periodo/2

   rt_task_make_periodic(&tarea1, ahora + 3000, periodo);
   rt_task_make_periodic(&tarea2, ahora + 3000 + ( periodo/2 ), \
                                                periodo);
   return 0;
}

void cleanup_module(void)
{
   rt_task_delete(&tarea1);
   rt_task_delete(&tarea2);
}
```

Nótese la utilización de la función *rt_get_time()* para obtener una referencia de tiempo, *rt_task_init()* para definir una tarea de tiempo real y *rt_task_make_periodic()* para ejecutarla. Estas funciones son propias de la extensión RTLinux. Para más detalles acerca de la utilización ver la bibliografía citada. Este programa

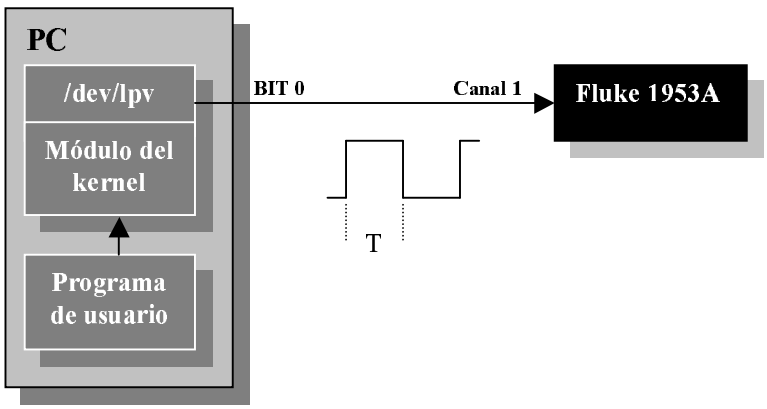


Figura III.3: Diagrama en bloques del primer experimento

se compila como un módulo de kernel y se instala con *insmod*, como de detalló anteriormente.

Para medir la dispersión de dicha onda cuadrada se utilizó un *Digital Counter Timer* marca Fluke, modelo 1953A, con rango 0-125MHz, con interfaz a PC, conectado como se muestra en la *figura III.3*.

Este instrumento tiene dos entradas, nueve dígitos y rangos desde 0.1 microsegundos hasta 10000000 segundos. Se programó el contador para medir el tiempo T transcurrido entre el flanco ascendente y el flanco descendente de la onda cuadrada, cuyo valor teórico es 20 milisegundos. Las mediciones se efectuaron durante largos períodos de tiempo.

Resultados obtenidos

En la *figura III.4* se muestra la dispersión de los 20 milisegundos de la onda cuadrada, como eventos versus período. Para el caso del Linux POSIX.1 se observaron desviaciones de -2/+1 milisegundos en las mediciones realizando únicamente tareas de background y -3/+10 milisegundos sometiendo al sistema a una carga intensiva de red y disco. En la *figura III.4.a* puede verse claramente cómo el aumento de la carga aumenta la varianza de la gaussiana. Los distintos tipos de carga influyen en forma diferente, pero hay un evento (medido a plena carga) que demoró 10 milisegundos más de lo esperado. En esta figura se hacen evidentes las variaciones producidas por los accesos al disco rígido.

Utilizando la extensión RTLinux para las mismas condiciones de carga la precisión se mantuvo en -10/+16 microsegundos. La especificación de este sistema operativo (por parte de los autores) declara que en una PC 486 este tiempo debería mantenerse acotado en ±30 microsegundos. En la *figura III.4.b* puede verse cómo la distribución de los tiempos es prácticamente insensible del tipo de carga que experimenta el Linux.

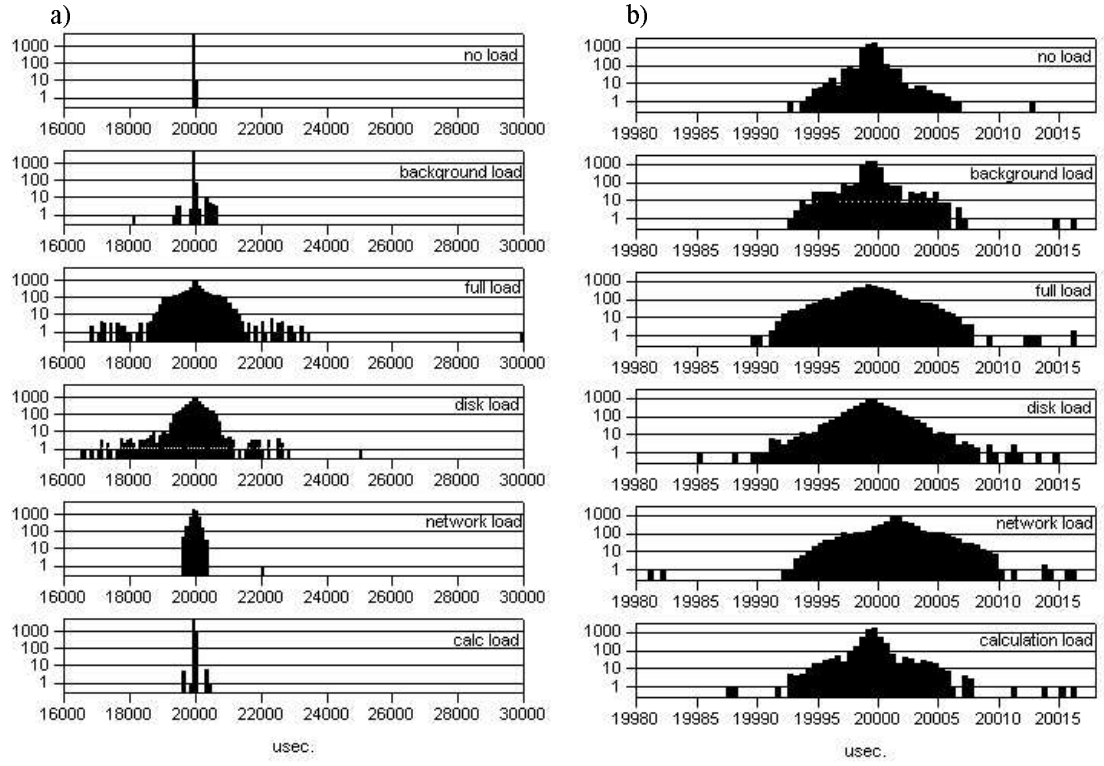


Figura III.4 a y b: Dispersión de los 20 milisegundos teóricos para Linux POSIX.1 y RTLinux, respectivamente

En las *tablas III.1 y III.2* se presentan los cálculos del valor medio, desviación standard, máximo, mínimo y kurtosis[22], para cada una de las mediciones realizadas, tanto para el Linux standard como para el RTLinux. Nótese el buen comportamiento promedio del Linux POSIX.1, reflejado en los valores medios de la *tabla III.1*. Esta es una característica de los sistemas de tiempo compartido, si bien las grandes

[22] La *kurtosis* es una medida del apartamiento del histograma de una gaussiana perfecta. Kurtosis igual a cero significa gaussiana.

variaciones en la varianza, y sobre todo el alejamiento de los máximos y mínimos, lo descartan para su utilización en aplicaciones estrictas de tiempo real.

Tipo de carga	Media	Desv Std	Min	Max	Kurtosis
Sin carga	19996	1.45	19989	20054	490.46
Background	19999	45.61	18186	20676	586.67
Carga completa	20009	538.84	16804	29908	27.38
Sólo disco duro	19991	391.14	16594	25054	18.63
Sólo red	19999	114.35	19653	22007	19.21
Sólo cálculo	19999	16.79	19643	20405	389.28

Tabla III.1: Generación de una onda cuadrada de 20 milisegundos con Linux POSIX.1, en microsegundos

Tipo de carga	Media	Desv Std	Min	Max	Kurtosis
Sin carga	19999	0.89	19993	20013	18.85
Background	19999	1.31	19993	20016	15.80
Carga completa	19999	2.45	19990	20016	2.10
Sólo disco duro	20000	1.92	19985	20014	5.92
Sólo red	20001	2.53	19981	20019	3.31
Sólo cálculo	19999	1.39	19988	20016	18.62

Tabla III.2: Generación de una onda cuadrada de 20 milisegundos con RTLinux, en microsegundos

En la *figura III.5* pueden verse en escala ampliada los dos casos extremos. Nótense las diferentes escalas de los histogramas. De la experiencia se desprende, como conclusión, que el sistema operativo RTLinux cumple con las especificaciones previstas (mantener la dispersión máxima acotada en 20 microsegundos), mientras que Linux POSIX.1 mostró un comportamiento que puede ser catastrófico en aplicaciones con requerimientos estrictos de tiempo, si el objetivo es sincronizar eventos con los relojes del sistema operativo. Sin embargo debe destacarse el buen comportamiento promedio de éste, que lo hace apropiado para un gran número de aplicaciones.

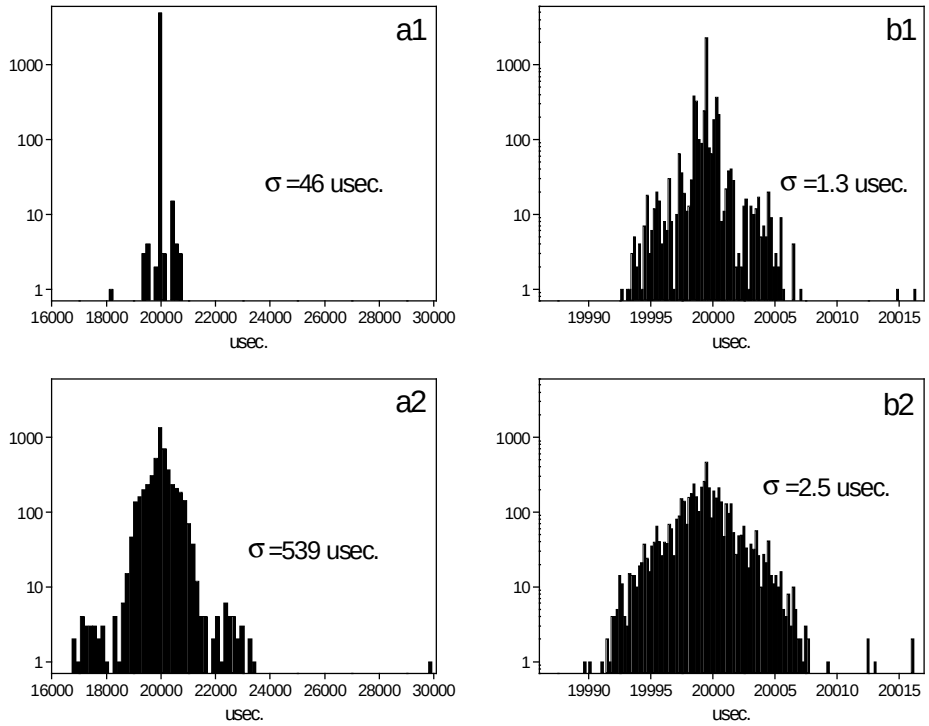


Figura III.5: Generación de intervalos. a1) Linux POSIX.1 sin carga; a2) Linux POSIX.1 con carga; b1) RTLinux sin carga; b2) RTLinux con carga

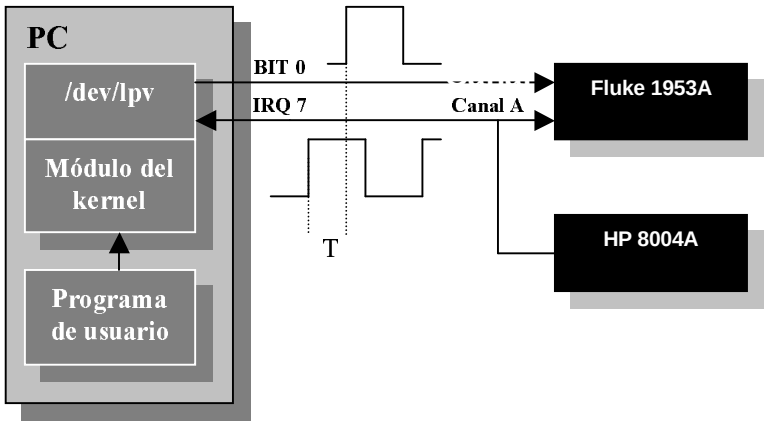


Figura III.6: Diagrama en bloques del segundo experimento

Experimento 2: Medición del tiempo de respuesta a la interrupción del port paralelo

El objetivo de este experimento es comparar los comportamientos de los sistemas operativos RTLinux y Linux POSIX.1 cuando tienen como tarea responder a una interrupción de hardware externa. Para ello se utilizó la interrupción disponible en el port paralelo de la PC (generalmente la IRQ 7), asignándole a dicha interrupción la tarea de generar un pulso corto en uno de los bits de salida del mismo port[23].

La dispersión del tiempo transcurrido entre ambas señales da una idea de las variaciones que pueden existir en el tiempo de latencia, o tiempo de respuesta a la interrupción.

Para asignar dicha tarea a la interrupción en el Linux POSIX.1 se utilizó el mismo driver que en el punto anterior, modificando la rutina de atención de la IRQ 7. A continuación se muestra sólo dicha modificación.

```
// Interrupt Handler
void lpv_irq7(int irq7)
{
   // Test directo:
   outb(inb(LPV)|0x02,LPV);
   outb(inb(LPV)&(~0x02),LPV);
}

//... y en la función open
request_irq(irq7,(void *)lpv_irq7,SA_INTERRUPT,"lpv",NULL);
```

Para asignar la misma tarea como módulo de tiempo real, se procede como se muestra a continuación.

```
//
// rt_irq.c
// Test para mediciones.
//
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <asm/io.h>
#include <asm/rt_irq.h>
#define LPT 0x380

int i;
char c;
```

[23] Para obtener la interrupción 7, debe habilitarse previamente ésta seteando la mascara correspondiente, y luego poniendo en 1 el quinto bit del byte de control de la puerta paralelo. Así, el seteo del bit ACK de la puerta paralelo genera una interrupción numero 7 al sistema [Stewart, 1994].

```
// Interrupt handler: Al llegar la interrupcion 7 genera un pulso.
void intr_handler(void)
{
   outb(0x01, LPT);                    // Escribe al p. paralelo.
   outb(0x00, LPT);
}

int init_module(void)
{
   request_RTirq(7, intr_handler);     // Solicita IRQ 7,
                                       // asignando la función.
   return 0;
}

void cleanup_module(void)
{
   free_RTirq(7);                      // Libera IRQ 7.
}
```

Al igual que el ejemplo anterior, este programa se compila como módulo y se inserta en el kernel de Linux. Nótese la utilización de las funciones *request_RTirq()* y *free_RTirq()* para la asignación de la tarea de tiempo real a la interrupción.

El tiempo transcurrido entre la señal externa que genera la interrupción y el pulso de respuesta es medido con el contador Fluke utilizado en el experimento anterior, que tiene la capacidad de medir el tiempo transcurrido entre dos flancos ascendentes en sus dos canales

Debe aclararse que el medido no es el *tiempo de latencia*[24] del sistema, sino que es la suma de éste con el tiempo de ejecución de la tarea de generar el pulso de respuesta. Sin embargo, las variaciones del tiempo medido son representativas de las variaciones de la latencia del sistema.

Para la generación de los pulsos externos que excitan la interrupción de la PC de utilizó el generador de pulsos Hewlett Packard 8004A. Éste tiene rangos entre 100 y 10MHz, salida máxima 5V, y rise time < 1.5nseg. Los instrumentos fueron conectados como muestra la *figura III.6*.

Esta experiencia se ejecutó también durante períodos largos de tiempo.

Resultados obtenidos

En la *figura III.7.a*, se observa para el Linux POSIX.1 una indeterminación en el tiempo máximo de respuesta a interrupciones, que *varía en función de la carga* entre 20μs y 30ms. Puede verse cómo, al igual que en el experimento anterior, el acceso a disco desmejora los tiempos de respuesta. Pero fundamentalmente se evidencia el efecto nocivo de la operación en punto flotante.

En el caso de RTLinux (*figura III.7.b*) puede verse cómo el tiempo de respuesta a la interrupción externa es prácticamente independiente de la carga y se mantiene acotado por debajo de 25μseg, si bien la media y la varianza dependen levemente del tipo de carga.

[24] El tiempo de latencia se define como el tiempo transcurrido entre la generación de la interrupción y la ejecución de la primera instrucción de la rutina asociada.

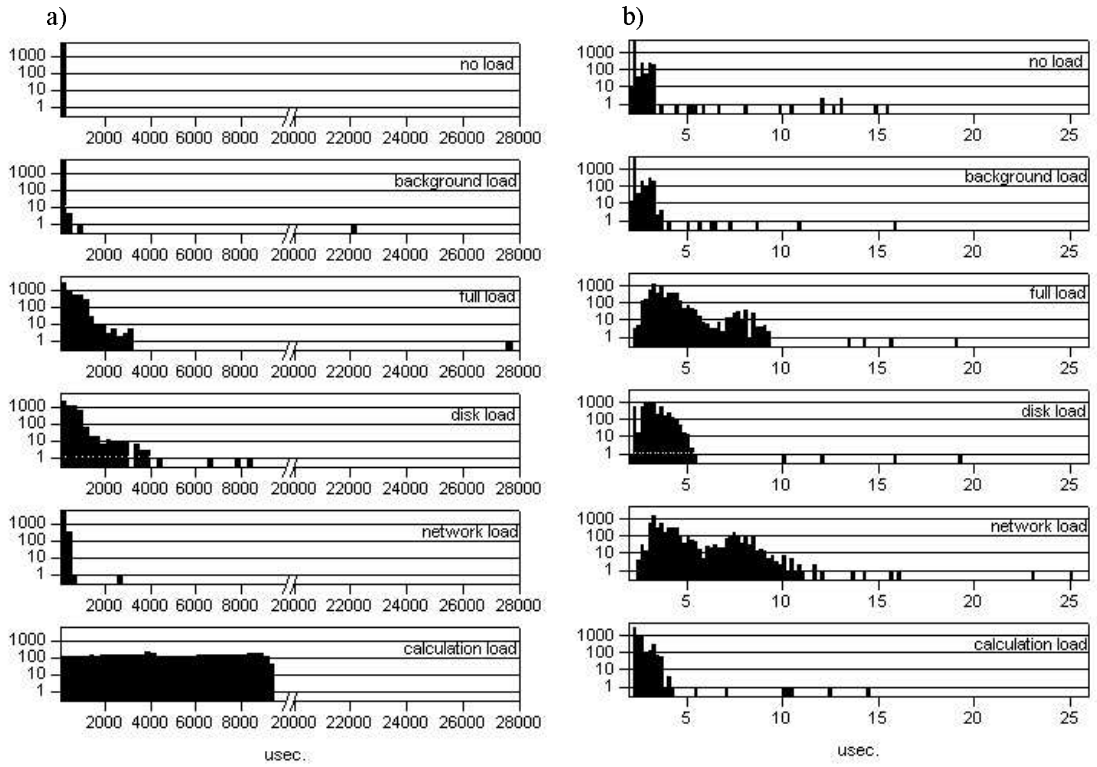


Figura III.7 a y b: Variación de los tiempos de respuesta a las interrupciones para Linux POSIX.1 y RTLinux, respectivamente.

En las *tablas III.3 y III.4* se presentan los cálculos del valor medio, desviación standard, máximo, mínimo y kurtosis, para cada una de las mediciones realizadas, tanto para el Linux POSIX.1 como para el RTLinux. En la *figura III.8* pueden verse en escala ampliada los casos extremos.

Tipo de carga	Media	Desv Std	Min	Max	Kurtosis
Sin carga	21.62	1.55	20.80	48.70	139.41
Background	28.16	312.70	21.30	22103.00	4963.00
Carga completa	369.31	528.20	29.10	27546	1408.00
Sólo disco duro	440.61	420.82	26.60	8292.00	63.35
Sólo red	114.33	91.59	27.50	2593.80	107.06
Sólo cálculo	4790.10	2684.80	26.20	9381.10	-1.21

Tabla III.3: Tiempo de respuesta a interrupciones externas para Linux POSIX.1, en microsegundos

Tipo de carga	Media	Desv Std	Min	Max	Kurtosis
Sin carga	2.36	0.55	2.10	15.50	266.11
Background	2.34	0.35	2.10	10.80	106.58
Carga completa	3.84	1.04	2.30	19.00	22.10
Sólo disco duro	3.20	0.65	2.10	19.40	112.96
Sólo red	4.38	1.71	2.50	25.00	8.90
Sólo cálculo	2.56	0.43	2.20	14.50	243.16

Tabla III.4: Tiempo de respuesta a interrupciones externas para RTLinux, en microsegundos

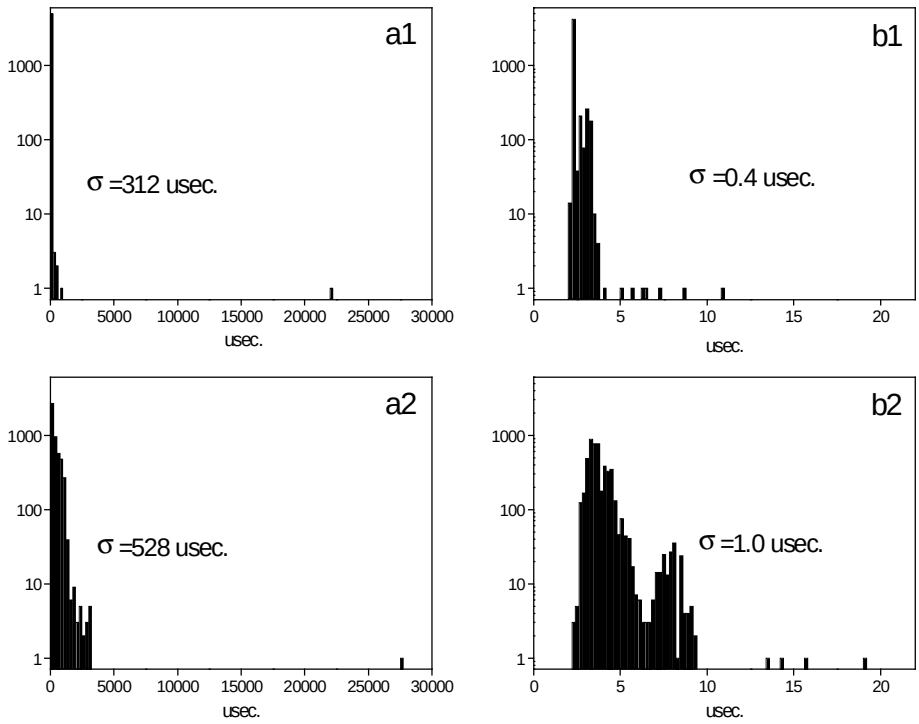


Figura III.8: Tiempo de respuesta a interrupciones externas. a1) Linux POSIX.1 sin carga; a2) Linux POSIX.1 con carga; b1) RTLinux sin carga; b2) RTLinux con carga

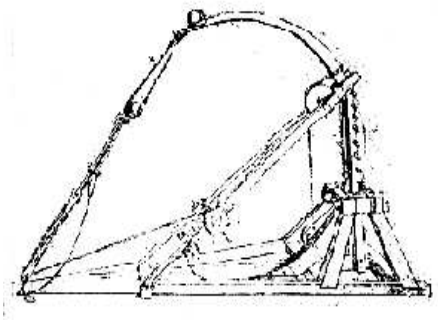
Resumiendo, estas experiencias han demostrado que el sistema operativo RTLinux puede ser utilizado en el caso de ser necesario agregar tareas de tiempo real a las aplicaciones, siempre que éstas se limiten al acceso a los recursos de hardware que no requieran de drivers complicados (esto descarta los discos rígidos y las interfaces de red, pero es muy útil para el acceso a adaptadores de instrumentación, como placas A/D y D/A, entradas y salidas digitales). Utilizando este sistema operativo, se dispone de archivos y acceso a las redes sólo a través de tareas POSIX.1, sin facilidades de tiempo real.

Este comportamiento es suficiente en el caso de la implementación de sistemas de adquisición de datos y control. En el *Capítulo V* se muestra cómo esta técnica es utilizada en varios casos en que fue necesario garantizar tiempos de respuesta.

Previamente, en el *Capítulo IV* se presenta una alternativa de hardware/software que mejora la performance del RTLinux utilizando una estructura distribuida, con varios procesadores.

Everything should be as simple as it is, but not simpler.

Albert Einstein



Capítulo IV
La placa μDAQ

En el *Capítulo III* se analizó la capacidad de los sistemas operativos Linux POSIX.1 y RTLinux para el manejo de tareas con estrictos requerimientos de tiempo, y se encontraron algunas limitaciones. Por ejemplo, en caso de presentarse una tarea que necesita ser atendida en un tiempo menor que 25 microsegundos, la administración de ésta no podrá hacerse con ninguno de estos sistemas operativos.

Para solucionar este problema, y para aliviar al Linux de algunas de las tareas de tiempo real del experimento, se presenta en este capítulo un nuevo elemento de hardware diseñado a tal efecto, compuesto por una placa para bus AT con un microcontrolador de 8 bits de Intel (bautizado *placa μDAQ*). Se presenta también un pequeño ejecutivo de tiempo real escrito a medida para dicho microcontrolador, que se encarga de la administración de tareas específicas, previamente asignadas a éste, con requerimientos mayores de tiempo (bautizado *sistema operativo μKeox*).

Se propone utilizar varias de estas placas en forma de sistemas autónomos que son programados sólo en la inicialización del experimento por el sistema operativo. Los microcontroladores se encargan de ejecutar un grupo reducido de tareas de tiempo real, y de reportar al sistema operativo Linux los resultados. Las tareas de tiempo real ejecutadas por Linux POSIX.1 o RTLinux se verán entonces limitadas a la comunicación de datos y señales entre las placas *μDAQ* presentes, y las alarmas del experimento. Fuera del campo del tiempo real, Linux deberá encargarse de todo el resto de las tareas, ya sea del almacenamiento de datos, acceso a las redes, display e interfaz con el usuario, etc.

En este capítulo se presenta primero la estructura de hardware de dicha placa, y a continuación una estrategia para la programación de microcontroladores de propósitos generales en la implementación de sistemas de tiempo real utilizando los conceptos básicos de un sistema multitarea basado en prioridades, previamente presentados en el *Capítulo I*. Esta estrategia, usada al escribir los programas de aplicación, puede proveer la solución a diferentes casos con estrictos requerimientos en tiempo, sin necesidad de utilizar software adicional ni microcontroladores sofisticados. Provee tiempos de respuesta predecibles, lo cual es altamente deseable en sistemas que se encuentran en contacto con el mundo real.

Se analizan dos implementaciones de software de fácil programación, que permiten organizar las tareas que lleva a cabo el microcontrolador. Ambas implementaciones se basan en asociar las diferentes tareas que debe realizar el microcontrolador con interrupciones de hardware. De esta manera se puede utilizar el manejo de interrupciones del microcontrolador como parte del scheduler, resultando un ejecutivo compacto y simple de programar.

De esta manera pueden asignarse varias tareas a cada placa *μDAQ* (una por cada interrupción disponible), las cuales serán ejecutadas por ésta dentro de un esquema preemptivo. Una tarea adicional

debe reservarse para establecer la comunicación con Linux, para la recepción de la configuración y el envío de datos.

A continuación se muestra la implementación de un control de temperatura para un calefactor de muestras, utilizado en experimentos de física nuclear. Esta placa de control de temperatura fue posteriormente utilizada en la automatización del experimento Mössbauer, tal cual se presenta en el *Capítulo V*.

Se presenta luego el driver POSIX.1 necesario para utilizar la placa en Linux. Este driver fue elaborado tratando de optimizar los recursos que el sistema operativo debe dedicar a la atención de la placa, utilizando las técnicas presentadas en el *Capítulo III*.

Finalmente se muestra un ejemplo de cómo podría utilizarse dicha placa dentro del entorno RTLinux. El ejemplo presentado utiliza tareas de tiempo real que, a través de FIFOs envía los datos a una aplicación gráfica POSIX.1, escrita en TCL/TK.

El hardware

La amplia difusión de los microcontroladores (dispositivos que integran un microprocesador, RAM, ROM, timers y periféricos de entrada/salida en un único chip), su versatilidad y bajo costo, han llevado a que, en una gran cantidad de casos, sean la solución más viable para la implementación de sistemas que interaccionan con el mundo real. Su capacidad de entrada/salida hace que sean una opción rápida y efectiva para el manejo de dispositivos externos, los cuales, en general, tienen restricciones estrictas en tiempo. En la *figura IV.1* se presenta la estructura general de un microcontrolador de 8 bits.

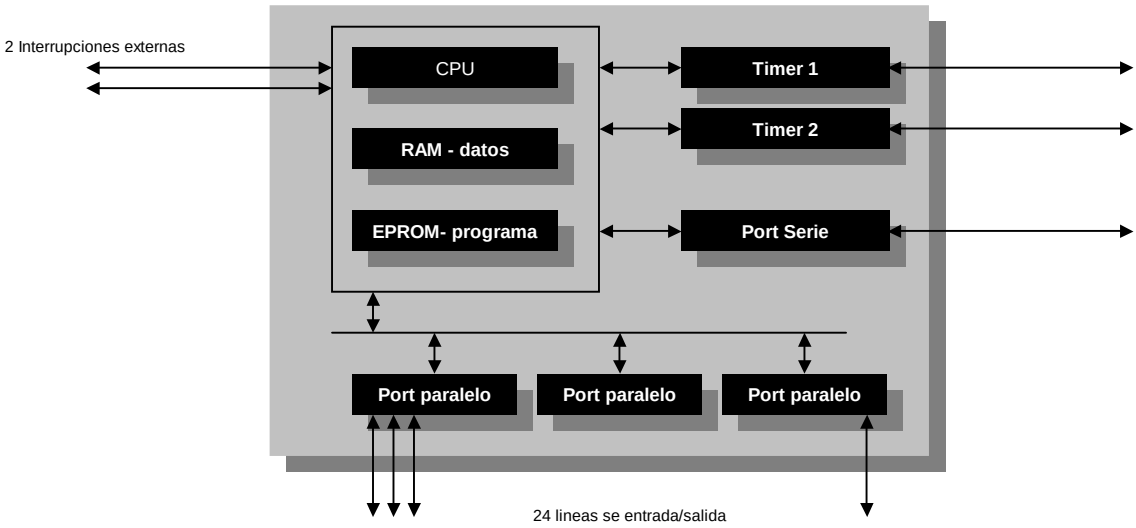


Figura IV.1: Esquema general de un microcontrolador de 8 bits

Para la construcción de la placa *µDAQ* se utilizó el microcontrolador 8051 de Intel montado en una placa con conector tipo AT, junto con la electrónica necesaria para implementar una comunicación de 16 bits con dicho bus. Esta placa fue construida utilizando los mecanismos clásicos de interfaz, donde dos de los ports de 8 bits del microcontrolador son accedidos desde un port de 16 bits de la PC. La dirección de dicho port puede ser seleccionada utilizando un juego de *jumpers* presente en la placa, entre varias opciones del mapa de la PC asignado al usuario.

Se utilizó la técnica de cablear el bit WR del bus AT con una de las interrupciones externas del microcontrolador. Así, cada vez que se escribe a la dirección asignada a la placa, se genera una interrupción en el microcontrolador. La rutina de atención de dicha interrupción debe encargarse de leer del bus la palabra de 16 bits que Linux está enviando al microcontrolador. Ésta será, en general, un comando o instrucción.

De la misma forma, el microcontrolador puede generar una interrupción externa de la PC con una de las líneas de salida de sus ports. Cada vez que el microcontrolador desea enviar un dato al Linux, coloca una palabra de 16 bits en sus dos ports de comunicación y genera la interrupción correspondiente. Para obtener este funcionamiento se cableó una de las líneas libres de los ports a una las interrupciones disponibles en el bus AT. En número de dicha interrupción puede ser seleccionado entre cuatro opciones a través de un juego de *jumpers* disponible en la placa. En la rutina de atención de Linux a dicha interrupción debe leerse la palabra que la placa *µDAQ* está enviando a través de bus de 16 bits. Esta palabra será, en general, un dato.

Al trabajarse en ambos sentidos utilizando interrupciones, se minimiza la cantidad de tiempos muertos que los microprocesadores emplean en la verificación de los dispositivos externos. Esta técnica disminuye la carga que el Linux debe afrontar, agilizando, como consecuencia, el resto de las tareas.

Nótese que para la implementación de las comunicaciones con el bus de la PC se utilizaron dos ports de 8 bits y una interrupción externa del microcontrolador. El resto de los recursos quedan disponibles para la implementación de diferentes configuraciones de hardware en la placa *µDAQ*.

Un diagrama en bloques de la placa *µDAQ* se muestra en la *figura IV.2*. En el apéndice correspondiente se presenta el esquemático completo de ésta.

Dos aspectos de la lógica deben ser tenidos en cuenta:

La señal proveniente de la decodificación del bus es un pulso que se utiliza tanto para activar el latch, como para interrumpir al microcontrolador. Si bien este pulso tiene los niveles lógicos correctos, el ancho de éste no es suficiente para interrumpir al 8051, por lo que debe intercalarse un monoestable.

El bit C16 del bus debe ser activado para indicar que la lectura/escritura es en 16 bits, pero dicha línea debe quedar libre cuando no se utiliza. Por lo tanto debe intercalarse un tri-state.

Para más detalles acerca de ambas implementaciones, ver el *Apéndice II*.

Con este hardware como punto de partida, y considerando los recursos del microcontrolador que quedaron libres, se detallan a continuación los aspectos fundamentales tenidos en cuenta para la elaboración del sistema operativo *µKeox*, encargado de la administración de tareas dentro del microcontrolador.

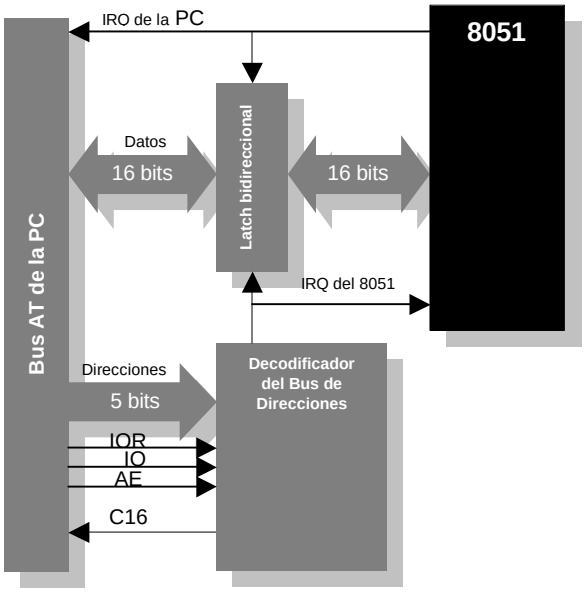


Figura IV.2: Diagrama en bloques de la placa µDAQ

µKeox, el ejecutivo de tiempo real

La técnica más simple y rápida para escribir programas de aplicación para diferentes tipos de microcontroladores es hacerlo en forma tal que las tareas que debe realizar se ejecuten secuencialmente,

siempre en el mismo orden (polled loop). Para solucionar los problemas de sincronización y de tiempos de respuesta que puedan presentarse, la técnica más común es hacer que el microcontrolador ejecute dichas tareas lo más rápido posible. En la mayoría de los casos puede encontrarse una solución por esta vía, y en caso de no ser posible se busca trasladar el desarrollo hacia un modelo de microcontrolador más potente.

Pero si se analiza en detalle el funcionamiento de un microcontrolador mientras ejecuta dicha secuencia fija de tareas, podrá observarse, en general, que éste emplea la mayor parte del tiempo esperando la finalización de eventos externos y señales de sincronización, sin realizar ningún procesamiento útil. Por ello, en la mayor parte de los casos en que no se consigue que el microcontrolador cumpla con requisitos de tiempo, la solución no es hacerlo trabajar más rápido, sino hacer que realice las tareas en forma más ordenada.

Como se mencionó anteriormente, una solución es asociar las tareas con interrupciones y administrarlas adecuadamente. Una de las opciones es utilizar prioridades y asignar las más altas a las tareas que tiene restricciones de tiempo más estrictas. Debe permitirse que estas tareas interrumpan la ejecución de tareas menos cruciales para el sistema (sistema preemptivo basado en prioridades). En general, este tipo de programación es adecuado para sistemas que tienen restricciones en tiempo, cuyo incumplimiento resulta en la falla total de dicho sistema (hard real-time systems).[25]

A continuación se analizan dos técnicas diferentes de scheduling que permiten organizar las tareas que lleva a cabo el microcontrolador, resultando un sistema con tiempos de respuesta acotados, lo cual es altamente deseable en sistemas que se encuentran en contacto con el mundo real.

En este caso se utilizó el microcontrolador 8051 de Intel, programado en assembler, pero lo expuesto no pierde generalidad al ser trasladado a otras plataformas, ya que los recursos disponibles son similares para las diferentes familias de microcontroladores de propósitos generales.

Principios básicos de funcionamiento

El kernel de un sistema operativo multitarea, tal como ha sido descripto en el *Capítulo I*, debe proveer al menos tres funciones específicas para la administración de tareas:

Scheduler de tareas: determina en qué orden y bajo qué condiciones se ejecutarán dichas tareas.

Dispatcher de tareas: toma los recaudos necesarios para iniciar un nuevo proceso en el entorno multitarea (*context switch*).

Comunicación: mecanismos de comunicación y sincronización entre los diferentes procesos que se ejecutan simultáneamente en el microcontrolador.

Recordemos que para su aplicación en el campo del tiempo real, es deseable que la estrategia de scheduling sea preemptiva y basada en prioridades. El principio de funcionamiento de esta estrategia se basa en que cada proceso tiene un nivel de prioridad fijo. Las tareas de más alto nivel de prioridad deben ser capaces de interrumpir la ejecución de las tareas de nivel más bajo. Con esta estrategia se consiguen los mejores resultados en la implementación de sistemas que tienen una estricta dependencia del tiempo, y es en general la solución más adecuada al planteo de un sistema multitarea que debe estar en contacto con el *mundo real*.

Una forma simple de implementación de ésta técnica en un microcontrolador es asociar cada una de las tareas que éste debe realizar con las interrupciones de hardware disponibles en dicho microcontrolador. Esto permite utilizar el manejo de interrupciones que provee el microcontrolador para la implementación del dispatcher y del scheduler.

Como primer paso, debe procederse a realizar una detallada clasificación de las tareas que debe realizar el microcontrolador, para cada aplicación en particular. Para ello es útil disponer del tiempo de ejecución de las tareas, junto con un análisis de la frecuencia de ocurrencia (para funciones periódicas) o tasa de arribo (para funciones no periódicas).

---

[25] La estrategia preemptiva basada en prioridades es la utilizada por algunas aplicaciones comerciales que se presentan como sistemas operativos de tiempo real para embedded systems. Estas aplicaciones tienen la desventaja de ser programas voluminosos que deben colocarse en memoria. Son voluminosos debido a que están escritos para cubrir diferentes casos de aplicación, lo cual los lleva irremediablemente a tener más funciones que las necesarias para un caso particular.

En general, las tareas pueden ser divididas en tareas que tienen restricciones estrictas en tiempo y tareas que pueden ser interrumpidas sin consecuencias graves para el conjunto.

El grupo de tareas con restricciones de tiempo puede subdividirse en tareas sincrónicas y asincrónicas. Las sincrónicas serán asociadas, en general, con las interrupciones provenientes de los timers internos del microcontrolador, los cuales se utilizarán para proveer la base de tiempo. Las tareas asincrónicas están generalmente asociadas a eventos externos que pueden utilizarse para disparar las interrupciones externas del microcontrolador.

Se propone la implementación de un sistema del tipo foreground/background (ver *Capítulo I*), donde las tareas con restricciones de tiempo (tareas de tiempo real) estén asociadas a interrupciones de hardware (foreground) y las tareas sin restricciones de tiempo constituyan el loop de background del programa, que es constantemente interrumpido por las tareas de foreground. Esto permite utilizar el manejo de interrupciones del microcontrolador para administrar las tareas a ejecutar simultáneamente.

En el grupo de tareas de background se incluyen las tareas que puedan tolerar interrupciones de larga duración. Incluso podría no haber ninguna tarea en el background.

En el grupo de tareas de foreground deben considerarse las tareas que se realizarán dentro de un esquema preemptivo basado en prioridades, lo cual significa que dichas tareas deben estar asociadas a interrupciones de hardware, ya sean externas o provenientes de los timers del microcontrolador, y que deben asignarse prioridades a estas tareas.

Debe tenerse en cuenta que la llegada de gran cantidad de interrupciones puede llegar a retardar excesivamente los procesos de background, incluso puede suceder que estos se suspendan por completo. De la misma manera debe chequearse la tasa de arribo de las interrupciones de más alto nivel de prioridad, pues retardan las de prioridad más baja. Para un análisis más detallado acerca de la asignación de prioridades y análisis de factibilidad, ver el final del *Capítulo I*.

En caso que se desee incluir en el esquema preemptivo una tarea asincrónica que no esté directamente asociada a una interrupción de hardware, sino a una condición de software, puede procederse de la siguiente manera: se asocia esta tarea a una interrupción externa y se activa dicha interrupción a través de un contacto eléctrico entre la entrada de dicha interrupción al microcontrolador y un bit cualquiera de uno de los ports de salida. Así, activando un bit por software, se produce una interrupción de hardware que ejecuta la tarea asociada.

En el punto siguiente se describen los recursos que generalmente están disponibles en los microcontroladores, y que nos permiten proceder con esta implementación.

Recursos disponibles

Para la implementación se dispone en este caso de un microcontrolador de propósitos generales que en general incluye los recursos enumerados a continuación, relacionados con el manejo de interrupciones. Estos recursos son similares para distintas familias de microcontroladores de distintos fabricantes. En este caso fue utilizado el microcontrolador 8051 de Intel.

Cinco fuentes de interrupción: dos interrupciones externas, dos interrupciones internas provenientes de timers, una interrupción de puerta serie. Las interrupciones son atendidas por medio de un llamado a una subrutina, que a su vez puede llamar a otras rutinas, las cuales pueden ser implementadas reentrantes.

Un registro de manejo de prioridad de dichas interrupciones (IP): permite solamente dos niveles de prioridad, como muestra la *figura IV.3*. Cuando llega una interrupción de menor o igual prioridad que la que está siendo atendida, ésta no es descartada, sino que queda pendiente su atención. Dentro de un mismo nivel, las diferentes interrupciones tienen diferentes prioridades, con el objeto de resolver interrupciones simultáneas. En orden decreciente, estas prioridades son: Interrupción Externa 0, Interrupción del Timer 0, Interrupción Externa 1, Interrupción del Timer 1.

Un registro de habilitación/deshabilitación de interrupciones (IE): Incluye un bit (EA) que deshabilita todas las interrupciones simultáneamente.

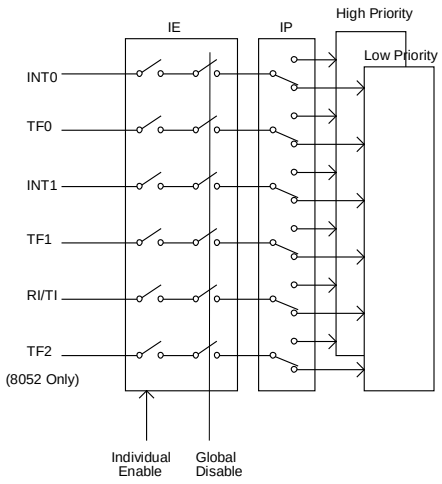


Figura IV.3: Manejo de interrupciones del 8051 de Intel

Estos recursos no son suficientes para implementar un scheduler y un dispatcher, ya que se desean varios niveles de prioridad y no sólo dos como provee el microcontrolador. Por lo tanto debe escribirse una parte del programa que se encargue de la administración.

Implementación del dispatcher de tareas

Se utiliza como dispatcher el mecanismo propio del que disponen los microcontroladores para asociar la ejecución de una subrutina a la llegada de una interrupción. Para ello no debe escribirse programa adicional. El microcontrolador, ante la llegada de una interrupción, produce un salto a una dirección prefijada y conocida para cada interrupción. En esa dirección puede incluirse un nuevo salto a la dirección de comienzo de la rutina de atención, que es la tarea que se desea realizar. El context switch que realiza el microcontrolador antes de producir el salto a la rutina de atención de una interrupción es únicamente del program counter (*PC*). El resto de los registros que sean modificados deben ser salvados en el stack por la propia subrutina. Esta estrategia presenta un tiempo de context switch variable para las diferentes rutinas, pero que puede ser calculado dentro del tiempo de ejecución éstas.

El context switch podría implementarse como una nueva subrutina que automáticamente envíe a un stack todas las variables utilizada y mantenga un registro de las tareas que están en ejecución. Pero guardar automáticamente el contexto implicaría guardar más variables de las necesarias y por lo tanto desmejorar el tiempo de atención de las interrupciones. Por ello se optó por continuar con un context switch ajustado para cada tarea, perdiéndose generalidad pero ganando en velocidad. Esto implica que cada tarea, al comenzar, debe guardar en el stack los registros del microcontrolador que va a utilizar. Esto requiere de una programación cuidadosa de cada tarea, pero redunda en menores tiempos de latencia.

Implementación de la comunicación entre procesos

La implementación de la comunicación entre tareas se realiza por medio de la utilización de variables globales, que pueden ser accedidas por todas las tareas. Es importante proteger el acceso a estas variables si son de más de 8 bits. Aunque el acceso a un registro o una posición de memoria de 8 bits es atómico, si una variable global es de 16 bits, el acceso a sus dos bytes puede ser interrumpido luego de la lectura del primero, por otra tarea que modifique esta variable, obteniéndose un dato incorrecto. Para ello puede optarse por la utilización de un semáforo o simplemente por la inhibición de las interrupciones durante el doble acceso (lo cual produce un retardo de tres ciclos de instrucción para las tareas de más alta prioridad, en caso que lleguen en ese instante). Las variables globales no son el método más elegante de comunicación entre procesos y deben utilizarse con precaución. Lo que es cierto es que son el método más rápido y sencillo, el cual acompañado con una programación cautelosa, suele ser la solución más adecuada para una gran parte de las aplicaciones que puedan presentarse.

Implementación del scheduler

A continuación se describen dos técnicas diferentes utilizadas para la implementación del scheduler de tiempo real del ejecutivo *μKeox*. La primera es una técnica simple, que necesita de un conocimiento grande de las características de las tareas a realizar, mientras que la segunda es más versátil, aunque requiere una programación adicional.

i. Implementación de un scheduler cooperativo

Esta implementación no utiliza un kernel. Se trata simplemente de varias tareas que se ejecutan en un esquema preemptivo basado en prioridades, pero sin un ente administrador principal. Las mismas tareas, en una base cooperativa, realizan la administración del scheduler de la forma que se describe a continuación.

Como se mencionó anteriormente, es deseable que la estrategia de scheduling sea preemptiva y basada en prioridades. El microcontrolador dispone del registro IP para el manejo de las prioridades de las interrupciones. Este registro solo permite asignar prioridad 0 o 1 a éstas. Utilizar solo dos niveles de prioridades puede ser útil en los casos que una de las tareas sea crucial (se le asigna prioridad 1), un grupo de tareas tenga restricciones de tiempo menos estrictas (se les asigna prioridad 0), y un grupo de tareas no tenga restricciones en tiempo (background). En el caso que se necesiten más niveles de prioridad, una solución es que las propias tareas manipulen constantemente el registro IP en una forma cooperativa, habilitando o deshabilitando la posibilidad de ser interrumpidas por otras tareas. Lo ideal es que se disponga de tantos niveles de prioridad como tareas deban ejecutarse.

El inconveniente de esta implementación es que requiere una cuidadosa programación, ya que cualquier tarea tiene el control total del sistema. La ventaja es que no utiliza un kernel ejecutivo adicional para la administración de las tareas, lo cual nos libra de retardos adicionales en el cambio de una tarea a la otra. Además el context switch no es automático, ya que debe realizarse manualmente al ejecutarse cada tarea, pero tiene la ventaja de no ser necesario el almacenamiento de todo el contexto, sino solo de los registros que serán utilizados, lo cual acelera el tiempo de respuesta.

Este tipo de configuración no siempre es aplicable, ya que se requiere un alto grado de conocimiento acerca de la frecuencia de ejecución de las tareas, como así también una importante sincronización entre éstas, ya que las propias tareas habilitan la ejecución de las demás a través de un manejo de prioridades. Para ello el nivel de cooperación entre las tareas debe ser alto. Al final del capítulo se muestra la aplicación de esta técnica a un caso específico.

ii. Implementación de un ejecutivo simple de tiempo real

Este ejecutivo puede ser considerado como una tarea de alta prioridad que tiene la capacidad de administrar la ejecución del resto de las tareas del sistema. Es un sistema operativo reducido. Debe proveer tres funciones fundamentales, tal cual se describió anteriormente: scheduling, dispatching y comunicación entre procesos.

Existen productos comerciales muy versátiles de carácter general que permiten implementar kernels de tiempo real en microcontroladores. Pero es esa generalidad y versatilidad lo que los hace voluminosos, pues proveen funciones que muchas veces no son utilizadas. En este caso, una programación que utilice los conceptos básicos de funcionamiento de un kernel ayuda a solucionar los problemas de tiempo sin necesidad de incluir un sistema operativo completo al sistema.

Se propone una *estrategia* de programación que utilice los conceptos básicos del funcionamiento de un kernel, con el objeto de cumplir los requerimientos de tiempo del sistema. Con una programación ordenada y un buen uso de las interrupciones y prioridades puede lograrse, sin la adición de una gran cantidad de código, un verdadero funcionamiento multitarea con un esquema preemptivo basado en prioridades. Este esquema es, como ya se mencionó, el más adecuado para la mayoría de las aplicaciones con restricciones de tiempo importantes (*hard real-time systems*).

Para ello, se escribió una nueva tarea que es la encargada de decidir cuál será la próxima tarea que debe ser ejecutada. Debe encargarse además de mantener un registro del estado de las distintas rutinas. Para ello se direccionaron todas las interrupciones a la tarea Kernel.

Al llegar una interrupción, este ejecutivo debe verificar si la tarea que está corriendo tiene mayor o menor prioridad que la tarea asociada a la interrupción. De ser menor, debe interrumpir la tarea que está ejecutando y debe iniciar la nueva, para luego retomar la suspendida. En el caso que llegue una de menor

prioridad debe marcar como pendiente dicha interrupción, para que sea ejecutada al finalizar las de prioridad mayor. Finalmente debe retornar de la interrupción.

Cada vez que se termina de ejecutar una tarea, el scheduler debe verificar la existencia de tareas pendientes en orden de prioridad, antes de devolver el control al loop de background. La ejecución de esta tarea adicional implica un aumento del tiempo de respuesta a las interrupciones, ya que todas las interrupciones deben pasar antes por el ejecutivo. Debe procurarse que este tiempo sea lo más pequeño posible y, fundamentalmente, que sea *constante* para todas las tareas, a fin de poder acotar los tiempos de respuesta.

Esta estrategia se implementó de la siguiente manera: la atención de todas las interrupciones se hace llamando a la rutina Kernel, utilizando un arreglo de flags para indicar cuál es la interrupción que ha llegado. La tarea Kernel hace un poll por los flags, de mayor a menor prioridad. Cuando detecta que ha llegado una interrupción verifica que no haya una tarea de mayor prioridad ejecutándose, y ejecuta la tarea asociada. Si está siendo ejecutada una de mayor prioridad la marca como pendiente y retorna de la interrupción. Cuando la tarea de mayor prioridad sea terminada, el Kernel se encontrará con la de menor prioridad pendiente y la ejecutará antes de retornar de la interrupción anterior.

Para que todas las tareas puedan ser interrumpidas por la tarea Kernel, ésta debe utilizar el registro de prioridades de interrupciones del microcontrolador para asignarle mínima prioridad a la tarea que está siendo ejecutada.

En el siguiente listado se presenta una implementación en pseudo-código para demostrar su funcionamiento.

```
Int1:Setear_Ready1       ; Llega Interrupción 1 (máxima prioridad)
   JUMP Kernel

Int2:Setear_Ready2       ; Llega Interrupción 2 (intermedia)
   JUMP Kernel

Int3:Setear_Ready3       ; Llega Interrupción 1 (mínima prioridad)
   JUMP Kernel


Kernel:                  ; NOTA: El orden de este polling es el de
                         ; prioridades de las rutinas.
                         ; Utilizo los flags Ready* y Running*
                         ; como Taks Status.

poll1:

   IFNOT Ready1 JUMP poll2       ; Miro si llego la interrupcion
                                 ; 1, si no miro la próxima.
   IF Running1 JUMP scherror     ; Si llego y esta todavía
                                 ; ejecutando,salgo con error.
   Limpiar_Ready1                ; Paso la tarea a ejecutando.
   Setear_Running1
   Bajar_Prioridad_1             ; Mientras esta corriendo le bajo
                                 ; la prioridad.
   EXEC Tarea_1                  ; Ejecuto la tarea asociada a la
                                 ; interrupción 1.
   Subir_Prioridad_1             ; Repongo la prioridad alta.
   Limpiar_Running1              ; Aviso que termino de correr la
                                 ; tarea 1.
   JUMP Kernel                   ; Termino. No debo salir hasta
                                 ; que no queden interrupciones
                                 ; pendientes.

poll2:

   IFNOT Ready2 JUMP poll3       ; Miro si llego la interrupcion2,
                                 ; si no miro la próxima.
   IF Running2 JUMP pollerror    ; Si llego y esta todavía
                                 ; ejecutando, salgo con error.
   IF Running1 JUMP pollexit     ; Verifico si está corriendo 1.
                                 ; De ser así marco la tarea como
                                 ; pendiente y salgo.
   Limpiar_Ready2                ; Idem anterior.
   Setear_Running2
   Bajar_Prioridad_2
   EXEC Tarea_2
   Subir_Prioridad_2
   Limpiar_Running2
   JUMP Kernel
```

```
poll3:

   IFNOT Ready3 JUMP pollexit    ; Miro si llego la interrupcion3,
                                 ; si no termino.
   IF Running3 JUMP pollerror    ; Si llego y esta ejecutando,
                                 ; salgo con error.
   IF Running1 JUMP pollexit     ; Verifico si están corriendo la
                                 ; 2 o la 1. De ser así
   IF Running2 JUMP pollexit     ; marco la tarea como pendiente
                                 ; y salgo.

   Limpiar_Ready3                ; Idem anterior.
   Setear_Running3
   Bajar_Prioridad_3
   EXEC Tarea_3
   Subir_Prioridad_3
   Limpiar_Running3
   JUMP Kernel

pollerror:
   EXEC Tarea_Error
pollexit:
   Return_from_Interrupt               ; Devuelvo el control
                                       ; al background.
```

En la *figura IV.4* se presentan los cuatro estados en los que pueden encontrarse cada una de las tareas, junto con las condiciones que deben cumplirse para que se modifiquen dichos estados.

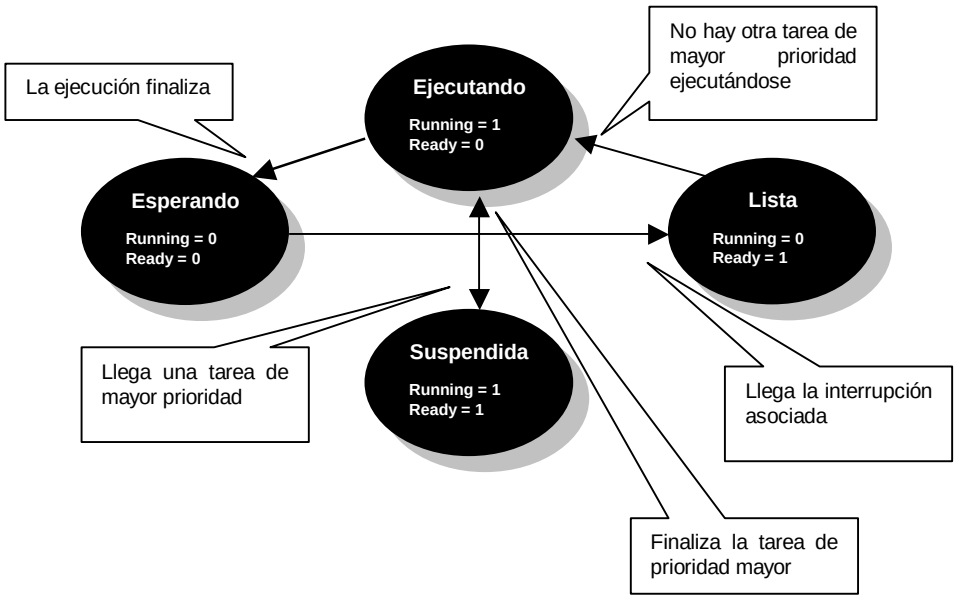


Figura IV.4: Estados posibles de las tareas

Performance y limitaciones

Haciendo un recuento manual de las instrucciones involucradas, para una implementación en assembler, puede asegurarse que la demora en la atención a interrupciones externas será menor que 20 ciclos de instrucción. En el caso de utilizarse un microcontrolador 8051 de Intel trabajando a 24MHz, éste tiempo corresponde a 10 microsegundos.

Ésta no intenta ser una implementación general de un kernel multitarea de tiempo real, sino que propone una estrategia de programación basada en los conceptos básicos presentados en el *Capítulo I*. Conociendo los principios de funcionamiento de un kernel de tiempo real, pueden aplicarse a una programación ordenada a fin de explotar al máximo los recursos disponibles en los microcontroladores de propósitos generales. Esta estrategia permite acotar los tiempos de respuesta de un sistema, lo cual es altamente beneficioso para las aplicaciones de tiempo real.

La principal desventaja es que el número de tareas está limitado por el número de interrupciones disponibles en el microcontrolador. Dicho número no puede ser modificado durante la ejecución del programa. Esto limita el campo de aplicación de lo anteriormente expuesto.

Debe aclararse que no se justifica extenderse demasiado en el perfeccionamiento y la generalización del scheduler y el dispatcher de tareas, ya que se estaría reescribiendo un sistema operativo de tiempo real. El campo de aplicación de esta estrategia de programación está limitado a aplicaciones en las que el microcontrolador efectúa tareas simples que tienen alta interrelación con dispositivos externos. En el caso que la programación se complique, es recomendable utilizar un sistema operativo de los que se encuentran disponibles en el mercado para las diferentes arquitecturas de microcontroladores.

Las experiencias realizadas aplicando esta estrategia a diferentes casos demostraron que con pocas líneas adicionales de programa es posible implementar una estrategia preemptiva basada en prioridades en un microcontrolador de propósitos generales. Sólo es necesario un conocimiento a fondo de los recursos disponibles y una cuidadosa programación.

Esta estrategia permite optimizar la utilización de la CPU, evitando tiempos muertos indeseables y, por lo tanto, permitiendo la resolución de problemas complejos con dispositivos standard, que hubieran requerido microcontroladores más potentes en el caso de utilizarse un polled loop.

La asignación de prioridades a las tareas debe realizarse cautelosamente, en el marco de lo expuesto en el *Capítulo I*, ya que de ello depende la performance total del sistema.

Desde el punto de vista del sistema operativo Linux, disponer de un sistema como el descripto, montado en una placa que pueda ser accedida a través del bus AT, le permite desligarse de un cierto grupo de tareas de tiempo real. Es evidente que las tareas asignadas a una misma placa deben estar relacionadas. Un punto clave en el diseño es realizar una adecuada distribución de las tareas. Esta distribución debe hacerse entre las diferentes placas *μDAQ* presentes en el sistema, y el sistema operativo Linux.

Para dicha asignación de tareas se debe tratar de minimizar el número de canales de comunicación necesaria entre los dispositivos presentes en el sistema. Es deseable que cada una de las placas *μDAQ* trabajen autónomamente, y que no sean necesarias excesivas comunicaciones entre ellas y el sistema Linux, ya que esto desmejoraría la performance del sistema. Para ello es necesario pensar en cada una de las placas como distintos *embedded systems* que tienen a su cargo un grupo altamente relacionado de tareas.

A continuación se muestra un caso de aplicación, utilizado luego en la implementación del experimento Mössbauer, en el cual todas las tareas de tiempo real relacionadas con la adquisición y control de la temperatura de la muestra fueron agrupadas en una placa *μDAQ*.

Caso de aplicación

Se utilizó la estructura presentada hasta ahora para construir un control de temperatura para el experimento Mössbauer. Para el sensado de la temperatura se utiliza una termocupla standard, asociada a un convertidor A/D doble rampa discreto, diseñado específicamente para su utilización con microcontroladores [Spinelli et al. 1996]. Para la modificación de la temperatura de la muestra se utiliza una resistencia calefactora accionada por un modulador de ancho de pulso, que es alimentado por una fuente regulada de tensión [Martínez et al. 1995].

Para la implementación de este conjunto de tareas en una única placa *μDAQ*, el microcontrolador debe realizar varias tareas simultáneamente: adquisición de la temperatura actual implementando el convertidor doble rampa discreto, cálculo de la acción de control, implementación del PWM por software, y por último manejo de las comunicaciones con Linux.

El hardware

Se utilizó el conversor A/D doble rampa que se muestra en la *figura IV.5*. Su característica principal es que fue implementado en forma discreta para su utilización con un microcontrolador, de forma tal de ocupar la mínima cantidad de recursos de éste: utiliza como base de tiempo uno de los timers internos del microcontrolador (Timer0), una de las interrupciones externas para detectar la finalización de la conversión (INT0), uno de los bits de cualquiera de los ports para comandar el cambio de sentido de la rampa (PA.1) y otro para el multiplexor de entrada (PA.3, no se presenta en la figura). El objetivo de

dicho multiplexor es adquirir la señal proveniente de una termocupla o del sensor de temperatura para la compensación de junta fría de la termocupla.

Sólo utiliza los siguientes componentes externos: un integrado con cuatro llaves analógicas, un amplificador operacional cuádruple, una referencia de tensión, resistencias y capacitores.

Tiene implementado un procedimiento de autocero con pocos dispositivos adicionales y utilizando un bit más de un port (PA.0), lo que permite trabajar con la precisión requerida (15 bits), independizándose de los corrimientos de los amplificadores utilizados para la termocupla.

Para implementar el modulador de ancho de pulso (PWM) se utiliza como base de tiempo un timer (Timer 1) y un bit de salida de uno de los ports (PA.2) activa la etapa de potencia.

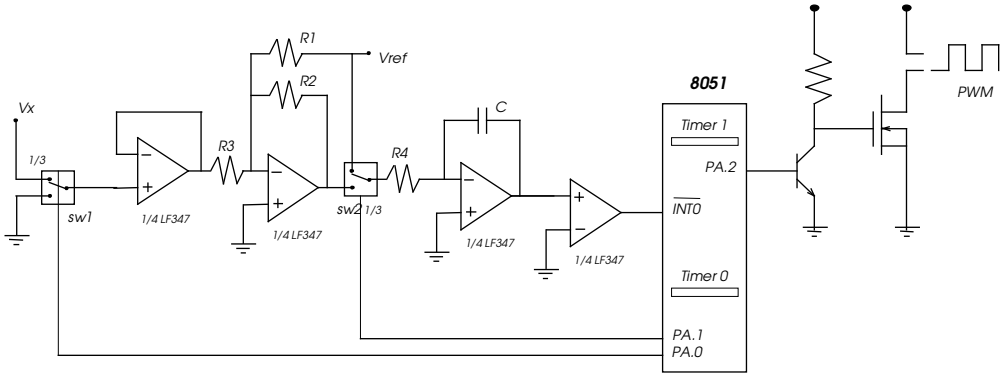
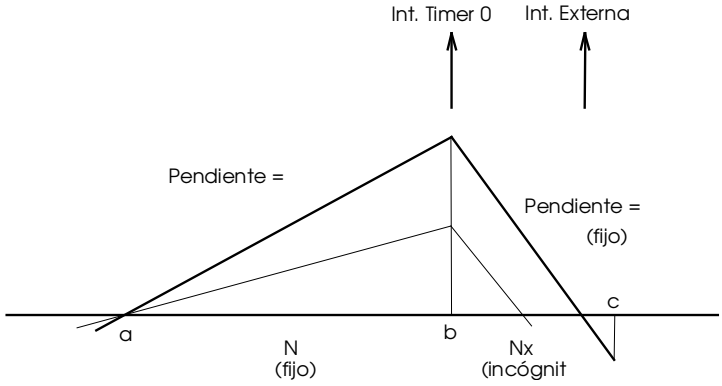


Figura IV.5: Esquemático del conversor A/D

La interrupción externa INT1 del microcontrolador se reserva para la comunicación con el bus AT de la PC.

El funcionamiento del conversor A/D se describe brevemente a continuación. Se carga el Timer 0 con un número fijo N elegido de antemano, que es el fondo de escala del conversor A/D (por ejemplo 4096 para conversión a 12 bits)[26] y se conmuta PA.1 para que aplique la tensión Vx de entrada a la rampa, generada por el capacitor. Cuando han transcurrido los N ciclos de clock, llega la interrupción del timer (la cual se genera al llegar la cuenta a cero). En la rutina de atención del timer se conmuta PA.1 para que aplique -*Vref* a la rampa. La interrupción externa 0 aparece cuando la rampa ha llegado a cero. Esta llegada a cero funciona además como gate del timer 0, en uno de los diferentes modos de programación de éste. Un esquema de la evolución de la rampa con el tiempo puede verse en la *figura IV.6*.



$$\frac{Vx}{R.C}.N = \frac{Vref}{RC}Nx \qquad Vx = Vref\frac{Nx}{N}$$

Figura IV.6: Ciclo de conversión

[26] El número de bits de resolución de este conversor A/D puede ser modificado en forma simple, teniendo en cuenta de conservar al capacitor de integración dentro de la zona lineal. Incluso puede utilizarse un número no entero de bits de resolución, cargando el timer con un valor que no sea un exponente de 2 (por ejemplo 6000, que equivaldría a 12 bits y medio aproximadamente).

Para aclarar el funcionamiento de este sistema, se presenta en la *figura IV.7* la red de Petri correspondiente, utilizando la extensión en tiempo de ésta. Se incluyen todas las tareas. Los tokens se encuentran en un estado tal que se está aguardando la finalización del conteo de ambos timers (este es el estado más frecuente en que se encuentra el sistema) y se están ejecutando las tareas de background.

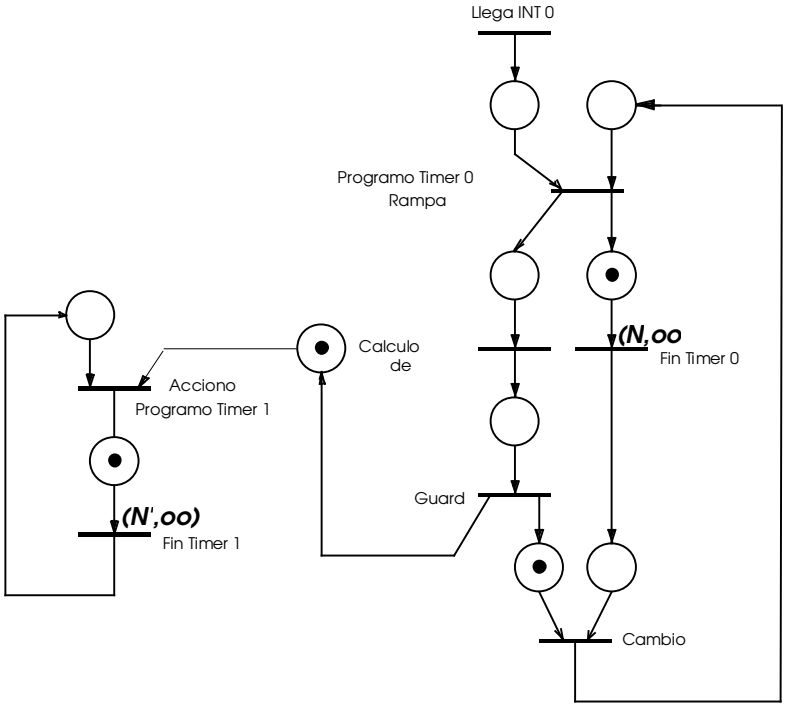


Figura IV.7: Red de Petri para el conversor A/D

El objetivo es implementar un sistema de software que sea capaz de manejar todas estas tareas simultáneamente, con tiempos de respuestas acotados.

Un primer intento fue la implementación de un sistema secuencial, del tipo polled loop, pero no se lograron resultados satisfactorios por dos motivos fundamentalmente. En primer lugar, el ciclo de conversión del A/D es muy diferente del correspondiente al PWM. En segundo lugar, el tiempo de conversión de un A/D doble rampa es variable, y por lo tanto la sincronización del sistema presentó inconvenientes.

A continuación se muestra la aplicación de las dos estrategias basadas en interrupciones, descriptas anteriormente.

i. Implementación del sistema cooperativo

En primer término, se asociaron las tareas con las interrupciones de la siguiente forma:

Interrupción del timer 0: cambio del sentido de la rampa del conversor A/D discreto utilizando un bit de uno de los ports de salida. Esta tarea es la que requiere mayor prioridad y debe ser capaz de interrumpir a las demás. Esto se debe a que un retardo en la ejecución del cambio de rampa produce un error en la adquisición de la temperatura actual. Esta tarea tiene la ventaja de ser la que menos instrucciones tiene (esto es muy común en sistemas de tiempo real)

Interrupción externa 0: detección del fin de conversión. La tarea asociada a ésta interrupción debe incluir el cálculo de la acción de control, y debe poner a disposición de la tarea que se encarga del PWM el valor de ésta, utilizando variables globales. Además debe presentar al proceso de background los valores de temperatura para enviar al display. No es necesario que esta tarea tenga una prioridad tan alta como la anterior, ya que al

ponerse en cero la línea INT0, el timer se detiene y no produce error. Debe permitir ser interrumpida por la interrupción del timer 0 y del timer 1.

Interrupción del timer 1: manejo del ciclo del PWM. Tiene una prioridad intermedia entre las dos anteriores. Tiene que ser capaz de interrumpir a INT_0.

Aquí estamos ante el caso en que se necesitan tres prioridades diferentes, además del background, lo cual se logró manipulando el registro IP como se muestra en el siguiente listado. En negrita pueden verse dichos cambios. Este programa incluye el redireccionamiento de las rutinas de atención de interrupciones y el procedimiento de inicialización de los recursos utilizados del microcontrolador. Algunas de las tareas no se describe en detalle y se incluyen solamente como llamados a subrutinas.

```
                        ;      REDIRECCIONAMIENTO      DE      LAS
INTERRUPCIONES:
   ORG 0000h
   LJMP Init            ; Reset.
   ORG 0003h
   LJMP INT_0           ; Interrupción externa 0.
   ORG 000Bh
   LJMP INT_T0          ; Interrupción del Timer 0.
   ORG 001Bh
   LJMP INT_T1          ; Interrupción del Timer 1.

   ORG 0100h

Init:
   LCALL Init_All

Loop_Bkgnd:             ; LOOP DE BACKGROUND:
                        ; Aquí ejecuto el background.
   LCALL Display        ; Refresco display tomando el dato de
                        ; las variables globales.
   LCALL Teclado        ; Teclado no manejado por interrupciones.
   AJMP Loop_Bkgnd

INT_O:                  ; FIN DE CONVERSION:
                        ; Interrupción 0: rutina de atención.
   LCALL Context_Switch ; Salvo los registros que voy a utilizar.
   LCALL Multiplex      ; Preparo el multiplexor para la
                        ; próxima conversión.
   LCALL Set_Timer_0    ; Reprogramo y arranco el timer.
   SETB  PA.1           ; Cambio a Vx.
   LCALL Calculo_PWM    ; Calculo la acción de control.
   LCALL Out_Data       ; Almaceno los cálculos en
                        ; variables globales.
   MOV IP,#00000010B    ; Doy prioridad al timer 0 hasta
                        ; que se produzca la interrupcion INT_T0.
   LCALL Context_Switch ; Recupero los registros que guardé.
   RETI


INT_TO:                 ; CAMBIO DE PENDIENTE:
                        ;  Interrupción  Timer  0:  rutina  de
atención.
   CLR PA.1             ; Cambio a Vref.
   MOV IP,#00001000B    ; Doy prioridad al timer 1 hasta que
                        ; se produzca la interrupcion INT_0.
   RETI


INT_T1:                 ; MANEJO DEL PWM:
                        ;
   LCALL Context_Switch ; Salvo los registros que voy a utilizar.
   CLR TR1              ; Paro el timer.
   LCALL On_Off         ; Comando la salida del PWM en On u Off
                        ; según corresponda.
   LCALL Context_Switch ; Recupero los registros que utilice.
   RETI
```

Nótese que las tres tareas conviven simultáneamente con tres niveles diferentes de prioridad. El máximo nivel de prioridad es para INT_T0. La INT_0 se deja interrumpir por INT_T1, manipulando IP en conjunto con INT_T0. El context switch es diferente en cada caso, y en INT_T0 no se realiza context switch en absoluto, pues no es necesario. Esto acelera el tiempo de respuesta de dicha interrupción.

Aquí puede verse el alto grado de conocimiento acerca del funcionamiento del sistema que es necesario, y algunas condiciones que deben cumplirse para lograr un adecuado funcionamiento.

En la *figura IV.8* se muestra la secuencia de eventos y la manipulación del registro de prioridades para el caso cooperativo.

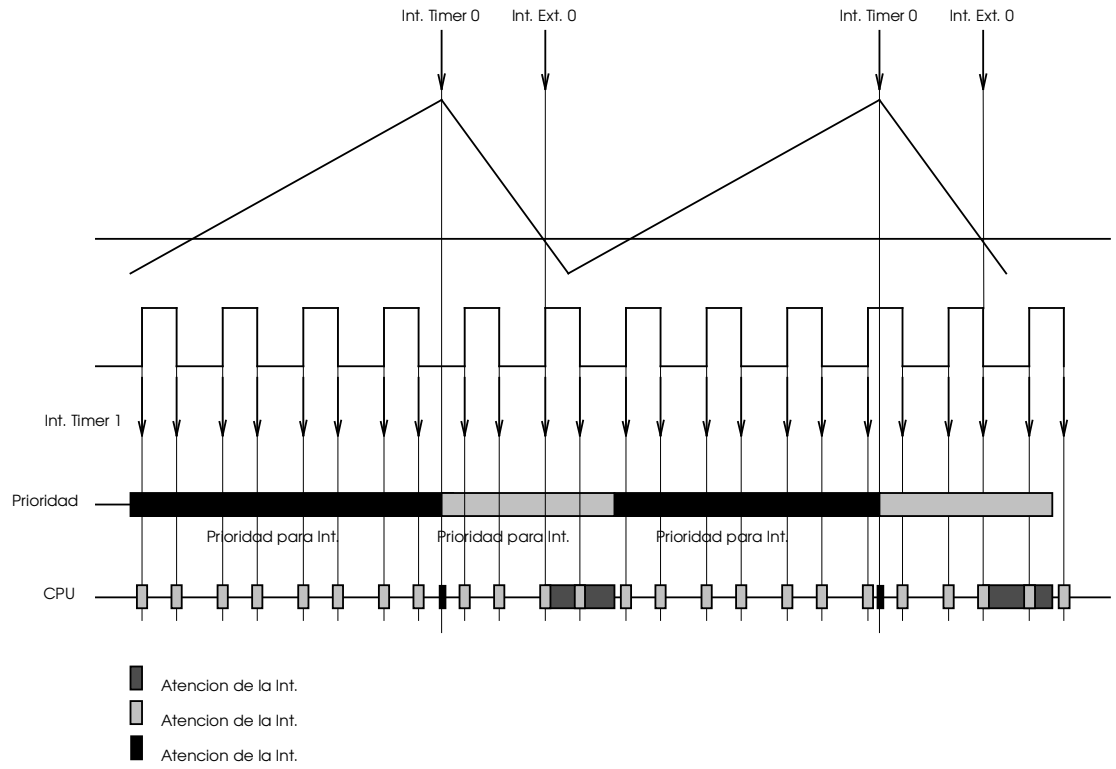


Figura IV.8: Secuencia de eventos en el conversor A/D

En primer lugar es necesario que las interrupciones se manipulen con cuidado y en forma cooperativa, pues si, por ejemplo, INT_T0 no realiza el cambio de prioridades adecuadamente, INT_T1 no podrá interrumpir la ejecución de INT_0, lo cual se evidenciará en una modificación del período del PWM.

También es indispensable un alto grado de sincronización entre INT_T0 e INT_0 para poder manipular IP en conjunto.

Por estos motivos, es posible encontrar casos en los que una implementación de este tipo sea imposible, dado que no se cumplen las condiciones para la cooperación. En ese caso es necesario agregar una nueva tarea con control sobre las demás, que se encarga de manipular las prioridades en función de los requisitos de tiempo establecidos.

Esta tarea se agregó en la implementación que se describe a continuación.

ii. Implementación del kernel

El software para el ejemplo anterior de aplicación se replanteó con el objeto de utilizar este pequeño ejecutivo de tiempo real. Se utilizó la misma asignación de tareas a interrupciones que en el punto anterior. Nótese que se eliminaron los manejos del registro IP dentro de las rutinas de atención de las interrupciones, y se agregó la tarea Kernel.

Antes de ejecutar una tarea debe bajarse el nivel de prioridad de ésta, para que pueda ser interrumpida por las demás. Para ello, debe cambiarse el nivel de prioridad de la interrupción durante su ejecución. Los cambios en las prioridades no son validados hasta el fin de una interrupción, por lo cual se ejecuta una instrucción RETI con un CALL, para forzar esta actualización de prioridades.

El siguiente listado muestra la implementación para el 8051, en la cual, por claridad, algunas de las tareas fueron resumidas a solo una llamada a subrutina, que no se presenta.

```
   ORG 0000h             ;      REDIRECCIONAMIENTO      DE      LAS
INTERRUPCIONES
   LJMP Reset            ; Reset.

                         ;  Todas  las  interrupciones  saltan  al
kernel,
                         ; previo aviso de cuál llego con Ready.*
                         ; (asignar Ready.1 a la interrupción de
                         ;  máxima  prioridad  y  Ready.4  a  la  de
mínima)
   ORG 0003h
   SETB Ready.3          ; Mínima prioridad para la
   LJMP Kernel           ; interrupción externa 0.

   ORG 000Bh
   SETB Ready.1          ; Máxima prioridad para la
   LJMP Kernel           ; interrupción del Timer 0.

   ORG 001Bh
   SETB Ready.2          ; Prioridad intermedia para la
   LJMP Kernel           ; interrupción del Timer 1.

   ORG 0100h

Kernel:
                         ; NOTA: El orden de este polling es el
orden
                         ; de prioridades de las rutinas.

poll1:
   JNB Ready.1,poll2     ; Miro si llego la interrupción del timer
0,
                         ; si no, miro la próxima.
   JB Run.1,pollerr      ;  Si  llego  y  esta  todavía  ejecutando,
salgo.
   CLR Ready.1           ; Atiendo la tarea asociada al timer 0.
   SETB Run.1            ; Aviso que esta corriendo.
   CLR IP.1              ; Mientras esta corriendo le bajo la
                         ; prioridad.
   CALL pollex           ; Para que se active el cambio de IP debo
                         ; ejecutar un RETI.
   LCALL INT_T0          ; Mando a ejecutar.
   SETB IP.1             ; Repongo la prioridad alta.
   CALL pollex           ; Ejecuto RETI.
   CLR Run.1             ; Aviso que termino de correr.
   AJMP Kernel           ;  Termino.  No  debo  salir  hasta  que  no
queden
                         ; interrupciones pendientes.

poll2:
   JNB Ready.2,poll3     ; Miro si llego la interrupcion del timer
1,
                         ; si no miro la próxima.
   JB Run.2,pollerr      ;  Si  llego  y  esta  todavía  ejecutando,
salgo.
   JB Run.1,pollex       ; Si esta Run.2 vuelvo luego.
   CLR Ready.2           ; Atiendo la tarea asociada al timer 1.
   SETB Run.2            ; Aviso que esta corriendo.
   CLR IP.3              ; Mientras esta corriendo le bajo la
                         ; prioridad.
   CALL pollex           ; Para que se active el cambio de IP debo
                         ; ejecutar un RETI.
   LCALL INT_T1          ; Mando a ejecutar.
   SETB IP.3             ; Repongo la prioridad alta.
   CALL pollex           ; Ejecuto RETI.
   CLR Run.2             ; Aviso que termino de correr.
   AJMP Kernel           ;  Termino.  No  debo  salir  hasta  que  no
queden
                         ; interrupciones pendientes.

poll3:
   JNB Ready.3,pollex    ; Miro si llego la interrupcion externa
0, si
                         ; no termino.
   JB Run.3,pollerr      ;  Si  llego  y  esta  todavía  ejecutando,
salgo.
   JB Run.1,pollex       ; Verifico si esta Run.2 o Run.3. De ser
así
   JB Run.2,pollex       ; vuelvo luego.
   CLR Ready.3           ;  Atiendo  la  tarea  asociada  a  la
interrupcion
```

```
                        ; externa 0.
   SETB Run.3           ; Aviso que esta corriendo.
   CLR IP.0             ; Mientras esta corriendo le bajo la
                        ; prioridad.
   CALL pollex          ; Para que se active el cambio de IP debo
                        ; ejecutar un RETI.
   LCALL INT_0          ; Mando a ejecutar.
   SETB IP.0            ; Repongo la prioridad alta.
   CALL pollex          ; Ejecuto RETI.
   CLR Run.3            ; Aviso que termino de correr.
   AJMP Kernel          ;  Termino.  No  debo  salir  hasta  que  no
queden
                        ; interrupciones pendientes.

pollerr:
   LCALL  Error         ; Hacer algo que indique overflow

pollex:
   RETI                 ; Regreso de la interrupcion.
                        ; También uso esta instrucción como
                        ; habilitador del
                        ;  cambio  de  prioridades,  llamándola  con
CALL.

PROGRAMA:

Reset:
   LCALL Init_All

Loop_Bkgnd:             ; LOOP DE BACKGROUND:
                        ; Aquí ejecuto el background, si existe.
   AJMP Loop_Bkgnd



INT_O:                  ; FIN DE CONVERSION:
                        ; Interrupción 0: rutina de atención.
   LCALL Context_Switch ; Salvo los registros que voy a utilizar.
   LCALL Multiplex      ; Preparo el multiplexor para la próxima
                        ; conversión.
   LCALL Set_Timer_0    ; Reprogramo y arranco el timer.
   SETB  PA.1           ; Cambio a Vx.
   LCALL Calculo_PWM    ; Calculo la acción de control.
   LCALL Out_Data       ; Almaceno los cálculos en variables
                        ; globales.
   LCALL Context_Switch ; Recupero los registros que guardé.
   RETI

INT_TO:                 ; CAMBIO DE PENDIENTE:
                        ;  Interrupción  Timer  0:  rutina  de
atención.
   CLR PA.1             ; Cambio a Vref.
   RETI

INT_T1:                 ; MANEJO DEL PWM:
                        ;
   LCALL Context_Switch ; Salvo los registros que voy a utilizar.
   CLR TR1              ; Paro el timer.
   LCALL On_Off         ; Comando la salida del PWM en On u Off
según
                        ; corresponda.
   LCALL Context_Switch ; Recupero los registros que utilice.
   RETI
```

Nótese que disponiéndose del listado completo del kernel, es posible calcular el tiempo de respuesta a las diferentes interrupciones, simplemente haciendo un recuento de los ciclos de instrucción necesarios para la atención.

La tarea de control

Tal como se comentó al comienzo del capítulo, la interrupción externa 1 del microcontrolador, junto con dos de los port de 8 bits, se reservaron para la comunicación de la placa con la PC, a través del bus AT. Para utilizar esta facilidad se escribió una rutina genérica de comunicación, a la cual puede asignársele la prioridad deseada en el scheduler. Esta rutina está asociada a la interrupción externa 1, y ante la presencia de ésta, toma una lectura del los ports de 8 bits para componer una palabra de 16 bits, que puede contener tanto datos como comandos.

Así, en el caso del control de temperatura mostrado, se utilizó esta palabra para pasar a través del bus la temperatura de referencia, en formato de 15 bits (con el bit más significativo en 0). Las palabras cuyo bit más significativo es 1 se reservaron para la implementación de comandos, con parámetros de 8 bits (hay 128 comandos posibles). De esta forma se implementó el comando FF*xx*h, donde *xx* es un número entero que se utiliza para determinar cada cuántos ciclos de adquisición se reportará una muestra de temperatura al bus de datos.

Performance del conjunto *µDAQ/µKeox*

Para comparar las prestaciones de esta nueva estructura, frente a las obtenidas en el *Capítulo III* para el sistema operativo RTLinux, se calculó el tiempo que transcurre entre la aparición de una interrupción externa del microcontrolador y la ejecución de la primer instrucción de la tarea asociada.

Los tiempos consumidos son los siguientes, dados en ciclos de máquina (o ciclos de instrucción) del microcontrolador (ver *figura IV.9*):

3 ciclos de máquina de respuesta del hardware (interno del microcontrolador [Intel Corp, 1983]);
un máximo de 19 ciclos de máquina debido al loop de toma de decisiones del kernel, sin incluir context switch; y
context switch manual.

Por ejemplo, para el caso presentado de atención a la interrupción correspondiente al Timer 0 (máxima prioridad, por lo tanto en el punto 2 consume 13 ciclos de máquina), en la cual no se hace context switch por no ser necesario, el tiempo que transcurre entre la aparición de la interrupción y la ejecución de la primer instrucción asociada, es de 22 ciclos de máquina. Como el microcontrolador consume 12 ciclos de clock por cada ciclo de máquina, funcionando con un oscilador de 20MHz se obtiene un tiempo de latencia de 1,1µseg.

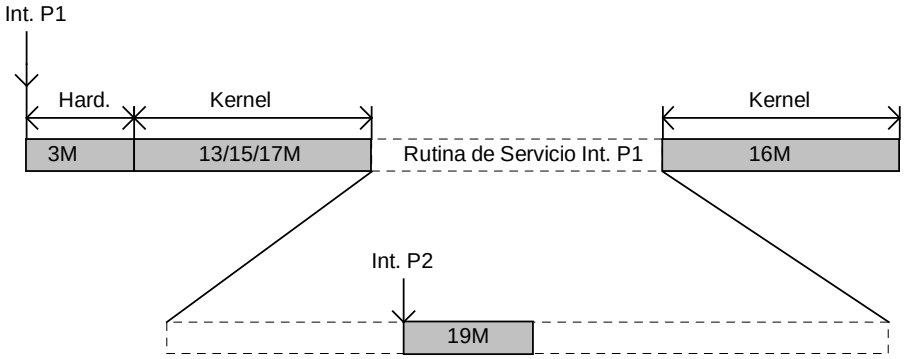


Figura IV.9: Tiempo de respuesta del ejecutivo µKeox a interrupciones externas.

Si bien este tiempo no es estrictamente comparable con el medido para RTLinux (porque el tiempo de éste incluía el accionamiento de un bit del port paralelo), puede verse que la respuesta de la placa es un orden menor que para Linux, debido a la simple implementación del algoritmo de toma de decisiones. Esta placa permite disponer de una herramienta preemptiva, con tiempos acotados, y con tiempos de respuesta relativamente cortos.

La placa *µDAQ* en el entorno Linux

En los puntos siguientes se muestran brevemente las dos formas de utilización de esta placa dentro del esquema propuesto. En primer término se presenta la estructura de un driver que permite la utilización de la placa *µDAQ* dentro del entorno POSIX.1 standard del Linux. Esta forma de trabajo permite la ejecución en tiempo real dentro de las placas, pero la comunicación con el sistema operativo no tiene esta característica. Tiene la ventaja de presentar la placa al usuario con una interfaz normalizada, y por lo

tanto portable. Luego se presenta una posible técnica para la utilización de la placa en el entorno RTLinux. Esta permite la operación de las placas en tiempo real, pero con una interfaz no normalizada.

El driver POSIX.1 para Linux

Para la implementación del driver que permita la utilización de la placa *µDAQ* en el entorno Linux POSIX.1, se pusieron en práctica los conceptos presentados en el *Capítulo III*. La estrategia de comunicación utilizada entre el Linux y la placa está basada en interrupciones. La placa funciona en forma autónoma, ejecutando las tareas que le fueron asignadas. Cada vez que tiene un dato disponible para el programa de usuario, genera una interrupción de la PC. En el otro sentido, cada vez que el programa en Linux desea enviar una palabra a la placa (para cambiar alguna configuración, como por ejemplo la tasa de envío de datos), escribe al port de la placa, lo cual genera una interrupción del microcontrolador, que será manejada por el ejecutivo *µKeox* de tiempo real. Este último se encargará, dentro de un esquema preemptivo, de hacer las modificaciones correspondientes en sus tareas.

Esta forma de funcionamiento permite a la placa un alto grado de autonomía, mientras que el programa de aplicación puede decidir si tomar o no los datos provenientes de la placa, según le convenga.

El driver se escribió como un módulo que puede ser agregado al kernel durante su ejecución. Este driver captura una de las interrupciones externas de la PC, que debe coincidir con la elegida por medio de los selectores presentes en la placa. Una vez instalado el driver en la memoria de sistema, la placa puede ser accedida como un archivo secuencial de caracteres, tal como se detalló en el capítulo anterior.

Un programa de aplicación que desee utilizar la placa debe primero abrir el dispositivo utilizando la llamada *open*. Para que esto sea posible, debe existir el punto de entrada correspondiente en el directorio */dev*, y además el dispositivo debe estar libre. Es el driver quien debe encargarse de evitar que dos programas de usuario tengan problemas de concurrencia sobre el dispositivo. Para ello utiliza un flag BUSY, como se muestra en la implementación de *open* en el *Apéndice III*, junto con el resto del código del driver.

El funcionamiento de un ciclo de escritura hacia la placa es el siguiente. Cuando un programa de aplicación ejecuta un *write*, el driver escribe la palabra indicada en la dirección del port de la placa (de 16 bits). La dirección de dicho port puede ser también modificada con selectores en la placa, para evitar conflictos de hardware en la PC, y debe coincidir con la dirección en la que escribe el driver. La decodificación por hardware de la placa genera una interrupción del microcontrolador, cuya rutina de atención debe encargarse de recolectar el dato que ha quedado almacenado en el latch.

Un ciclo de lectura se inicia también desde el programa de aplicación, al ejecutarse la llamada *read*. Al recibir el driver dicha llamada, pone en espera la lectura hasta recibir la interrupción correspondiente a la placa. La presencia de dicha interrupción indica que el microcontrolador tiene una palabra de 16 bits disponible en el latch, para ser leída por el Linux. Ante la llegada de la interrupción el driver acciona el hardware y traslada dicha palabra a la memoria de sistema. Por último libera la lectura, trasladando la palabra de 16 bits a la memoria de programa. Esta forma de funcionamiento de la lectura permite una mejor sincronización de los datos que provienen de la placa, con el sistema operativo Linux.

Por último, al recibir el driver la llamada *close*, libera el flag BUSY de la placa, para que pueda ser utilizada por otra aplicación, poniendo a la placa en un estado conocido (reset).

La placa *µDAQ* en RTLinux

Se presenta a continuación el listado de un módulo para el kernel de Linux, basado en la extensión de tiempo real del mismo (RTLinux). Como se mencionó anteriormente, esta es una alternativa con tiempos de acceso al hardware acotados, pero que no puede disponer de las facilidades POSIX.1 de Linux.

La forma de funcionamiento es la siguiente: se captura la interrupción seteada por jumpers en la placa y se le asigna la tarea de enviar a un FIFO de salida el dato presente en el port de entrada/salida correspondiente de la PC. En el otro sentido, se instala un manejador del FIFO de entrada, que se encarga de enviar a la placa un dato enviado desde Linux. Este procedimiento fue explicado en detalle anteriormente en el capítulo anterior. En la *figura IV.10* se presenta un esquema en bloques de la transferencia de datos presente en esta configuración.

Esta forma de funcionamiento permite la operación basada en eventos, lo cual disminuye notablemente la carga del sistema operativo. La aplicación presentada en el próximo capítulo utiliza, en una de sus

versiones dos placas manejadas con esta técnica por una tarea de tiempo real, con una interfaz a Linux compuesta por cuatro FIFOs.

El listado presentado a continuación es un módulo general para el manejo basado en eventos de la placa *µDAQ*, y será modificado en el capítulo siguiente para la operación con dos placas.

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <linux/rtl_version.h>
#include <asm/io.h>
#include <asm/rt_irq.h>
#include <rtl_fifo.h>

#define CONTADOR 0x280
#define IRQ_CONTADOR 7

unsigned short data;
unsigned short msg;

void int_contador(void)
{
   data = inw_p(CONTADOR);
   rtf_put(1,&data,2);
}

int hand_contador(unsigned int fifo)
{
   rtf_get(3, &msg, 2);
   outw_p(msg, CONTADOR);
   return 0;
}

int init_module(void)
{
   rtf_create(1, 4000);
   rtf_create(2, 4000);
   rtf_create_handler(3, &hand_contador);
   request_RTirq(IRQ_CONTADOR, int_contador);

   return 0;
}

void cleanup_module(void)
{
   rtf_destroy(1);
   rtf_destroy(2);
   free_RTirq(IRQ_CONTADOR);
}
```

Se presenta a continuación, y a modo de ejemplo solamente, el listado de una aplicación Tcl/Tk para Linux que permite utilizar los FIFOs para intercambiar datos con la placa. Nótese la utilización de los FIFOs como si se tratara de archivos ordinarios, lo cual facilita notablemente la implementación.

```
#!/usr/bin/wish

set i 0
set cuentas 0

set fd1 [open "/dev/rtf1" r]
set fd2 [open "/dev/rtf2" r]

proc Contador {} {
   global cuentas fd1 i
   binary scan [read $fd1 2] s data
   set cuentas [expr $data + $cuentas_]
   if { $i == 40 } {
          puts "$cuentas c/seg"
          set i 0
          set cuentas 0
   }
   incr i
}

fileevent $fd1 readable Contador
```

```
# Inicializacion del Contador
puts -nonewline $fd2 [ binary format s [ expr 65525 - 41660 ] ]
flush $fd2

# GUI (no presentada)
```

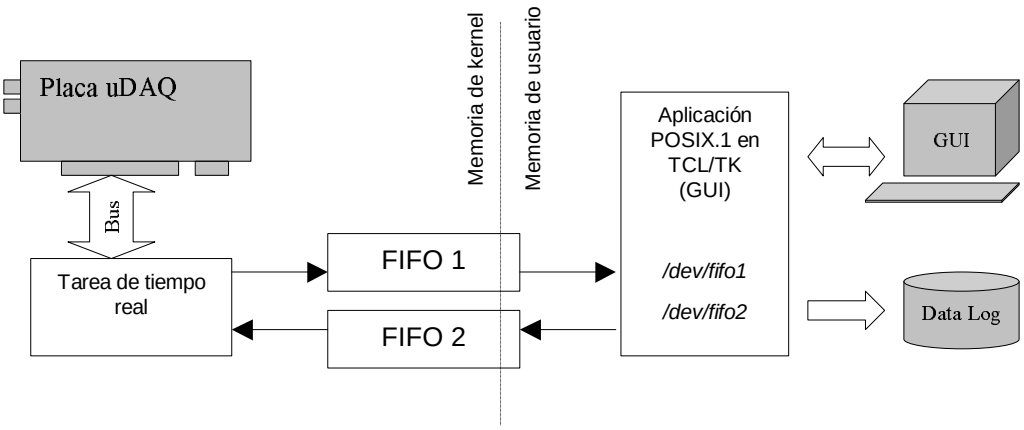


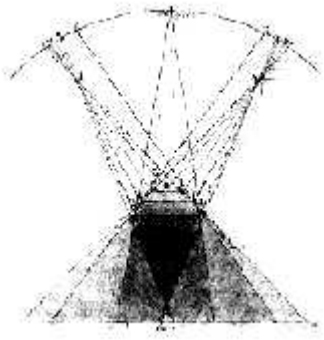
Figura IV.10: Esquemático de la transferencia de datos entre una aplicación POSIX y la placa, utilizando RTLinux

Resumiendo, en este capítulo se presentó un nuevo elemento de hardware que puede incorporarse a la estructura jerárquica de adquisición de datos y control. Se trata de un adaptador para PC, para ser montado en el bus AT, compuesto por dispositivos standard y de simple implementación. Contiene un microcontrolador de 8 bits que mantiene una comunicación bidireccional con la PC, y que permite la ejecución autónoma de un grupo de tareas, dentro de un esquema preemptivo, basado en prioridades. Si bien la extensión RTLinux presentó una buena performance en la ejecución de tareas de tiempo real (como se mostró en el *Capítulo III*), la nueva estructura propuesta lo mejora notablemente y permite liberar al Linux de un grupo importante de tareas.

Se detalló también el driver necesario para que éste sea utilizado en el entorno POSIX.1 de Linux, y se presentó un ejemplo de aplicación, que será utilizado en el capítulo siguiente, en la automatización de un experimento Mössbauer. Por último se mostró una posible utilización de la placa en el entorno RTLinux.

La experiencia es aquello que hace que una persona cometa nuevos errores, en lugar de los viejos.

Anónimo



Capítulo V
Aplicación al experimento

En el transcurso de este capítulo se muestra la implementación de un experimento de física nuclear, utilizando los diferentes elementos desarrollados en capítulos anteriores. Se presentan algunas de las variantes ensayadas, junto con las correspondientes conclusiones acerca de su funcionamiento. La diversidad de recursos disponible permitió un alto grado de optimización, obteniéndose un sistema robusto y de simple implementación.

Si bien la arquitectura resultante de hardware y software está directamente condicionada por las características del experimento implementado, el resultado obtenido presenta rasgos generales que permiten analizarlo como un entorno versátil de adquisición de datos y control. La mayoría de las partes que integran este entorno son reconfigurables, lo que permite considerar su utilización en otras aplicaciones de similares características. Tanto el hardware como el software desarrollados para este caso particular fueron diseñados en forma modular y bien documentados, para que dicha reutilización sea posible.

A continuación se presenta una breve descripción del experimento que se desea automatizar, del cual surgen los requerimientos técnicos del sistema. Luego se procede con un análisis más profundo de las posibilidades de agrupación de las tareas y su distribución entre las posibles plataformas, elemento clave del diseño, tal cual se comentó en los capítulos anteriores. Finalmente se muestra cómo se utilizaron los elementos hasta ahora presentados para la solución de los diferentes problemas. En algunos casos existen varias alternativas posibles, por lo tanto se presentan las ventajas y desventajas de cada una.

Descripción del experimento

El *efecto Mössbauer de absorción de rayos gamma* fue descubierto en 1958 por E.L. Mössbauer y es ampliamente utilizado en la actualidad en espectrometría de sólidos. Se basa en la absorción por resonancia de rayos gamma en la estructura cristalina de un material. Dicha absorción depende fuertemente del estado electrónico, químico y magnético de los átomos que componen dicho material [Cranshaw, 1974].

Con un espectrómetro Mössbauer se busca obtener la absorción del material como una función de la energía de los rayos gamma (fotones) incidentes, proporcionados por una fuente. La variación de energía se obtiene por efecto Doppler (el movimiento de la fuente de rayos gamma produce una variación de la frecuencia del fotón, y por lo tanto de su energía). Esto equivale a contar los fotones que atraviesan el material en función de la velocidad de la fuente.

Un esquema de un espectrómetro clásico [Battaioto et al. 1994] puede verse un la *figura V.1*. La fuente de rayos gamma se encuentra montada sobre un pickup electromagnético, cuyo movimiento produce el barrido en velocidad. La fuente es desplazada cíclicamente de forma tal que el rango de frecuencias de interés es recorrido varias veces por segundo. La *figura V.2* muestra la forma de la referencia digital de velocidad del control del pickup, para el caso de un experimento de aceleración constante (*dv/dt=cte*). Esta debe tener valor medio nulo para no producir desplazamientos acumulativos del pickup.

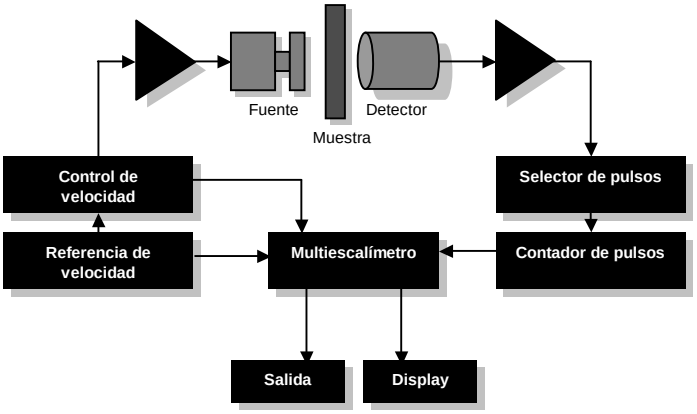


Figura V.1: Diagrama en bloques de un espectrómetro Mössbauer de aceleración constante

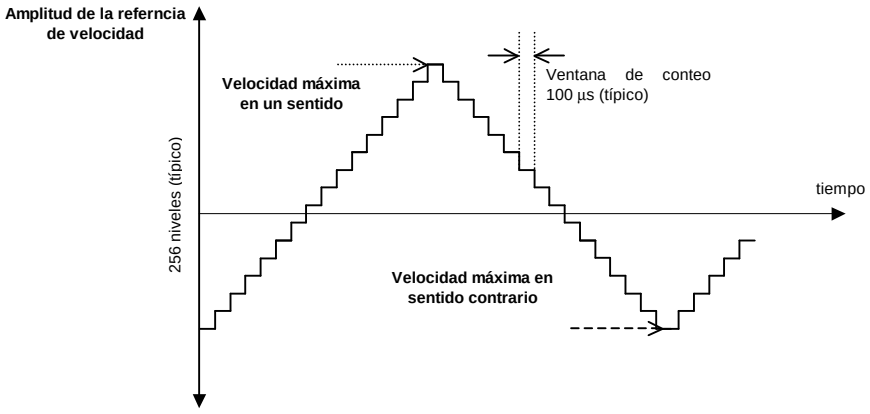


Figura V.2: Referencia de velocidad para un espectrómetro Mössbauer de aceleración constante

Para cada incremento de velocidad los pulsos provenientes del detector son contados por el sistema de adquisición. Si los incrementos de velocidad y el tiempo de conteo son constantes (dv/dt constante), se trata de un espectrómetro de aceleración constante. La forma de la referencia es una onda triangular. Un valor típico para la ventana de conteo es de 100µseg, y para un número de incrementos (canales) de velocidad 256.

El pickup está sincronizado con el sistema de adquisición, y para cada velocidad las cuentas del detector son sumadas en una posición diferente de la memoria del multiescalímetro (canal). El espectro se obtiene así como una gráfica de las cuentas (absorción) en función del número de canal (velocidad, por lo tanto energía).

Dentro de las principales características del experimento, pueden mencionarse que es no destructivo y que trabaja con muestras sólidas. Cuando el material consta de varias fases, el espectro resultante es la suma de los espectros individuales. En el caso particular de muestras delgadas, la intensidad del espectro de cada componente es proporcional a su concentración. Este método permite también poner de manifiesto los cambios que se presentan debido a la variación de la temperatura de la muestra [Chien, 1977].

El efecto Mössbauer es más evidente en el hierro y sus compuestos que en otros elementos y se ha transformado en una herramienta de estudio muy importante de este tipo de materiales, predominantes en

el desarrollo tecnológico. En la *figura V.3* puede verse un espectro Mössbauer de aceleración constante de un compuesto de hierro.

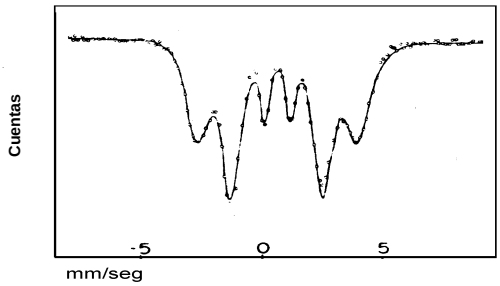


Figura V.3: Espectro Mössbauer de aceleración constante de un compuesto de Fe

La variante implementada en esta tesis es un espectrómetro Mössbauer a velocidad constante, con barrido automático de la temperatura de la muestra. En este caso se desea mantener constante la velocidad de la fuente de rayos gamma (energía constante) en una línea del espectro que se desee estudiar en particular, y producir un barrido de la temperatura del material en estudio. Se produce así un histograma con la absorción en función de la temperatura, para cada una de las líneas de energía del Mössbauer clásico. Por medio de este experimento se busca localizar cambios en la transmisión debido a transformaciones de fase (estructurales o magnéticas) y/o reacciones sólido-sólido o sólido-gaseoso.

En la *figura V.4* se muestra la variación del espectro Mössbauer de aceleración constante del $Fe_{80}B_{20}$ a diferentes temperaturas. La *figura V.5* muestra una experiencia de barrido de temperatura realizada manualmente, presentando la absorción en el centro del espectro (velocidad constante) en función de la temperatura. En el caso particular del experimento que se llevará a cabo, se desea obtener este tipo de resultados utilizando un sistema automático de recolección de datos. En la *figura V.6* pueden verse los espectros Mössbauer de aceleración constante en los puntos A, B y C de la *figura V.5*. Puede notarse cómo los espectros en A y C son diferentes a pesar de estar tomados a la misma temperatura. Esto se debe a que haber llevado la muestra hasta la temperatura B produjo un cambio irreversible en la estructura del $Fe_{80}B_{20}$. Estos fenómenos son del tipo que se desean estudiar sistemáticamente, y para lo cual se implementará la experiencia.

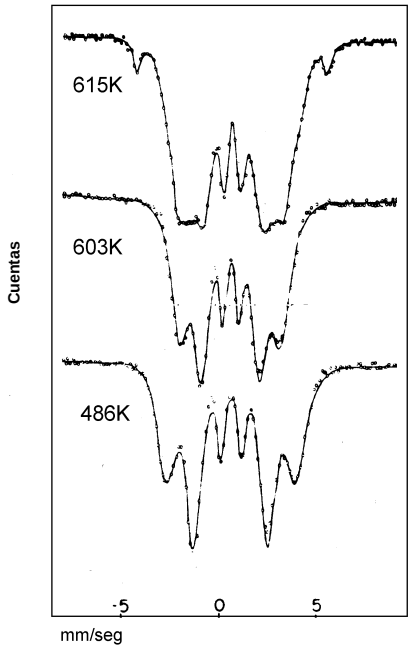


Figura V.4: Espectro Mössbauer del $Fe_{80}B_{20}$, a altas temperaturas [Chien, 1977]

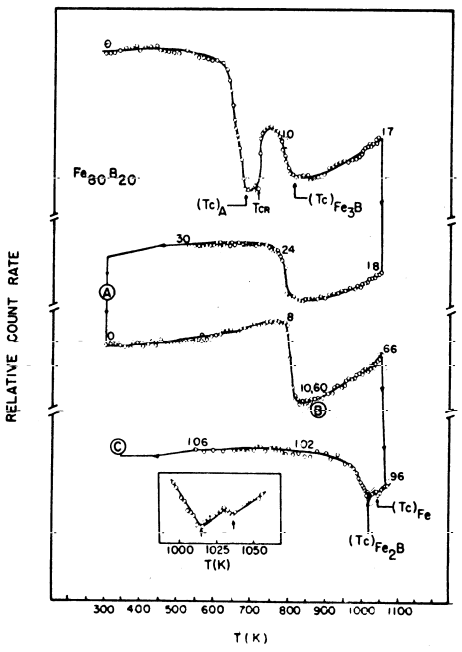


Figura V.5: Cuentas a velocidad constante en función de la temperatura [Chien, 1977]

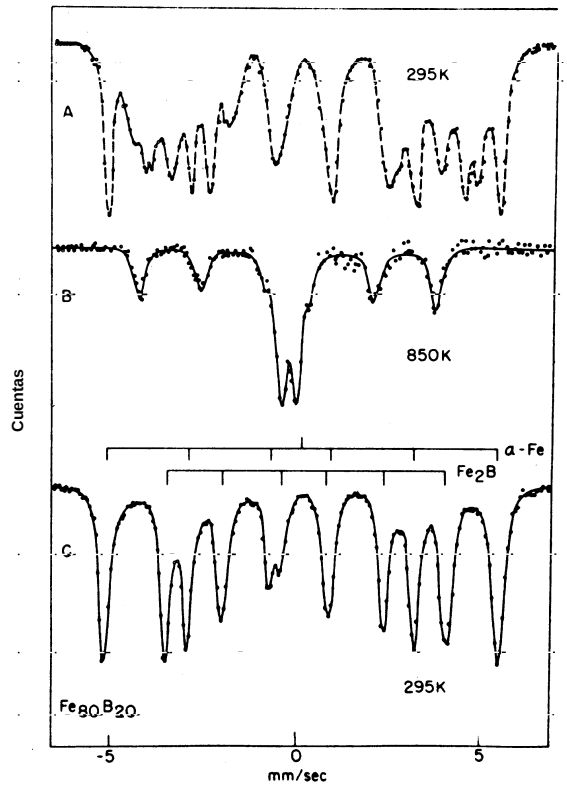


Figura V.6: Espectro Mössbauer del $Fe_{80}B_{20}$ en los puntos A, B y C de la figura V.5 [Chien, 1977]

Para este experimento, la referencia de velocidad para el pickup debe tener la forma mostrada en la *figura V.7*, conservando el valor medio nulo. La parte superior de la onda cuadrada es equivalente al movimiento a velocidad constante de la fuente, y la parte inferior es el regreso del pickup a la máxima velocidad posible, para comenzar un nuevo recorrido a la velocidad de interés. Durante el regreso del pickup, el conteo de eventos debe ser inhibido.

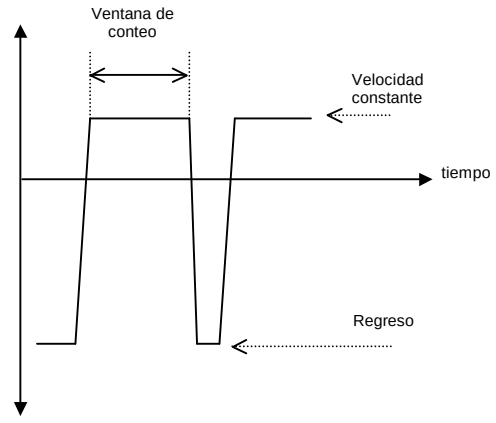


Figura V.7: Referencia de velocidad para el experimento de velocidad constante

Para la implementación se utilizará parte del equipamiento disponible para los experimentos Mössbauer clásicos. Tal es el caso de los siguientes equipos:

Un detector del tipo contador proporcional.
Un preamplificador y un amplificador.
Un pickup lineal, *Mössbauer Spectrometer Linear Motor Model K3* [A.S.A., 1985].

Un módulo *Single-Channel Analizer*, EG&G ORTEC Modelo 551 [EG&G Ortec, 1987].

Un control de velocidad para el pickup, con referencia programable, diseñado previamente en nuestro laboratorio [Martínez et al. 1999].

Un calefactor portamuestras con capacidad de vacío y termocupla tipo K montada, A.S.A. modelo VF-1000 [A.S.A., 1985].

Se verá luego que, dadas las características del calefactor, es deseable la implementación de una etapa de potencia que utilice la técnica PWM con ciclos enteros de línea. Debido a las características del experimento, y a la resolución deseada para el histograma resultante, es necesario que el control de temperatura permita trabajar entre temperatura ambiente y 800°C, con una resolución de 0.2°C (rango de trabajo del calefactor).

El resto del experimento será implementado utilizando una PC standard con sus correspondientes interfaces y sistema operativo. Se dispone de una configuración Pentium de 120MHz, lo cual debería ser suficiente para la implementación deseada.

Desde el punto de vista del usuario, es deseable el máximo nivel posible de automatización (ya que estas experiencias se ejecutan durante períodos prolongados de tiempo), una interfaz gráfica completa que permita observar el desarrollo del experimento, acceso remoto a las variables y los datos (incluso mientras se desarrolla la experiencia), y registro de los datos en un formato standard que permita el procesamiento posterior de la información.

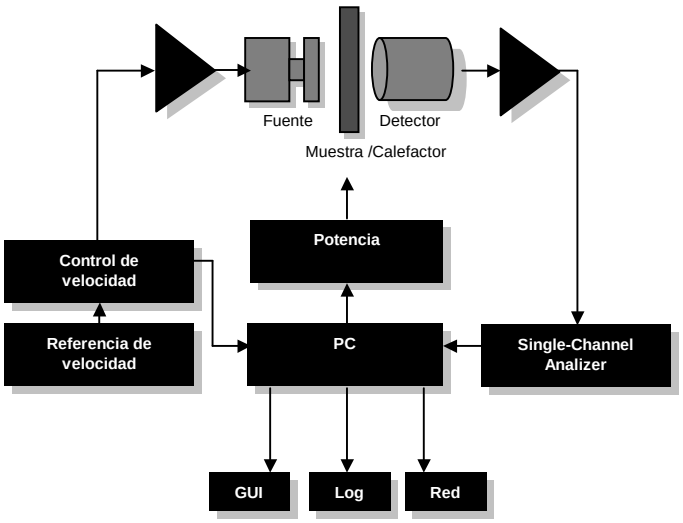


Figura V.8: Diagrama en bloques de la configuración deseada

En la *figura V.8* se presenta un diagrama en bloques general para la configuración deseada. A continuación se describen brevemente algunas características importantes de los tres instrumentos del experimento Mössbauer clásico que serán utilizados en este caso.

1. El selector de pulsos

El módulo SCA *Single-Channel Analizer ORTEC 551* es un selector de altura de pulsos (provenientes del preamplificador/amplificador) que genera una salida lógica cada vez que recibe un pulso de una altura prefijada, funcionando como selector de energías. La salida del módulo es un pulso cuadrado de amplitud 5V, de 500ns de ancho, con un *rise-time* menor que 20ns. La resolución entre pulsos especificada por el fabricante es de 800nseg, por lo que es de esperar que la máxima señal de salida de este módulo será como se muestra en la *figura V.9*.

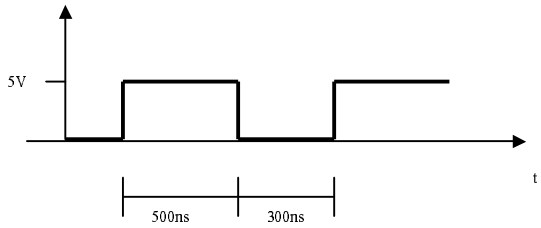


Figura V.9: Máxima tasa de salida del SCA.

Los materiales que serán estudiados son diferentes aleaciones metaestables y materiales magnéticos de base Fe. La fuente radiactiva utilizada para esta experiencia es ^{57}CoRh. El ^{57}Co decae en ^{57}Fe, lo cual lo hace apto para el estudio de materiales de base Fe. Es de esperar, en este caso, una tasa de arribo de 10^3 a 2.10^4cuentas/segundo o más (dependiendo de la actividad de la fuente), con una distribución del tipo Poisson. Se tomará en lo sucesivo una tasa de 10^5 para los cálculos. Dada la respuesta del SCA, puede calcularse la probabilidad de que, habiéndose detectado un evento, exista uno nuevo en el lapso de 800ns siguiente. Aplicando Poisson, la probabilidad de dos eventos con una tasa de 0.8 es P(2)=0.3%. Esto indica que el 0.3% de los eventos no serán detectados a causa de las características del SCA. El contador de pulsos que se implemente deberá conservar dicha característica, tratando de minimizar la pérdida de eventos.

2. El control de velocidad con referencia programable y el pickup

Para este tipo de experimentos es común utilizar un pickup del tipo lineal, como el presentado en la *figura V.10*. En este caso se dispone de un motor lineal modelo K3 de A.S.A. [A.S.A., 1985]. Se dispone también en el laboratorio de un control de velocidad analógico para dichos pickups, con un módulo de referencia de velocidad programable. Este último permite optar entre diferentes formas de onda, incluida la necesaria en este caso (ver *figura V.7*). Este equipo fue diseñado y construido en nuestro laboratorio [Martínez, 1999]. Es capaz de manejar una resolución en velocidad de 0.22mm/seg, necesaria para este tipo de experimentos. El rango de velocidades que serán utilizadas en este caso es de ±2mm/seg.

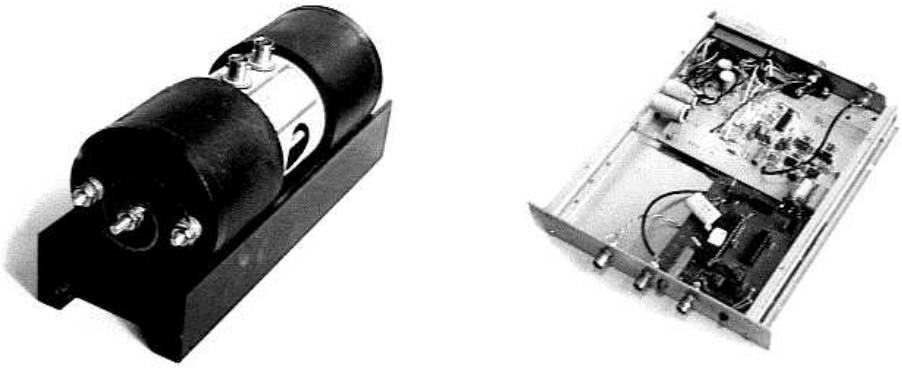


Figura V.10: El pickup lineal y el control de velocidad programable

3. El sistema calefactor

Se dispone en el laboratorio de un calefactor con capacidad de vacío *VF-1000*. Se trata de un recipiente cilíndrico cerrado donde se monta la muestra en estudio. Es posible el calentamiento de dicha muestra por medio de una resistencia eléctrica, y el sensado de la temperatura por medio de una termocupla tipo K. Un corte y vista exterior se muestran en la *figura V.11*.

Respecto de las referencias de dicha figura, A es la conexión para la válvula de vacío, E la aislación térmica, K/L/M el portamuestras y D son dos ventanas de Berilio de 0.006" de espesor, que permiten el paso de los fotones. C es un conector DIM externo para la potencia y la termocupla H que se encuentra montada en el interior. En la vista exterior pueden verse dos caños de cobre para la refrigeración por

circulación de agua, la válvula de vacío montada, una de las ventanas de Be en el centro y el conector DIM.

Según especificaciones, el funcionamiento de este calefactor debe mantenerse dentro de un rango de temperaturas de 300 a 1000°K y su consumo nominal es de 490Watt para 1000°K.

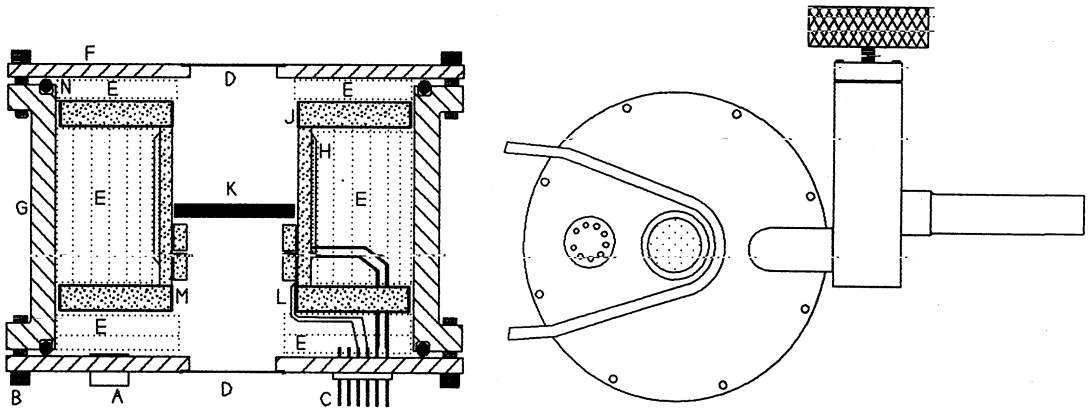


Figura V.11: Corte transversal y vista exterior del calefactor de vacío VF-1000

Con el objeto de caracterizar el comportamiento del calefactor se realizaron algunas mediciones, que se comentan a continuación. La resistencia medida del arrollamiento de alambre de kanthal fue de 10Ω. En la *figura V.12* se muestra una de las curvas de calentamiento ensayada aplicando una corriente de 1A a dicho arrollamiento. Aproximando la respuesta del calefactor a una exponencial con retardo, se deduce un retardo de 10s, una constante de tiempo de 1500s y una ganancia de 30°C/Watt. Estos son valores promedio, que varían para diferentes rangos de temperatura. Dichos valores son suficientes para calcular los parámetros de un control PID clásico.

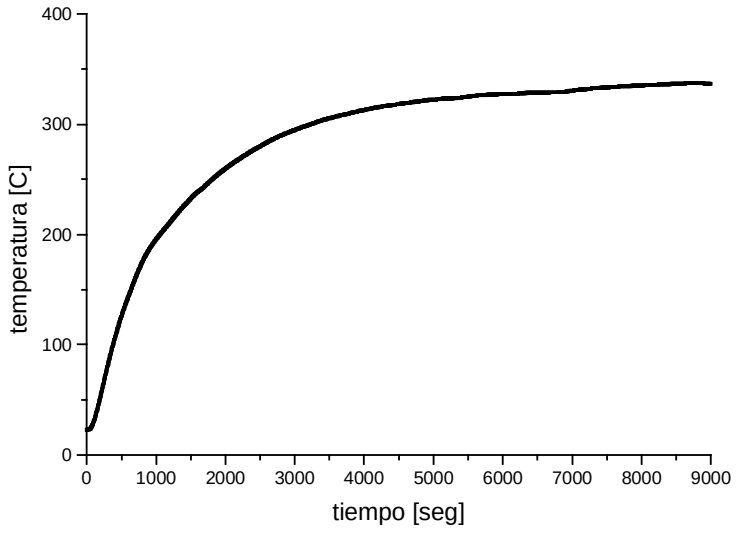


Figura V.12: Respuesta del calefactor a un escalón de potencia de 10Watts

Dada la constante de tiempo del calefactor, es de esperar que un control de temperatura que entregue la potencia directamente desde la tensión de línea estabilizada utilizando un transformador[27], produzca una estabilidad aceptable, sin presentar ripple.

Análisis del problema y distribución de las tareas

El primer paso a seguir hacia la solución del problema es identificar claramente cuáles son las tareas a realizar por el sistema a cargo de la automatización, y cómo pueden éstas agruparse en conjuntos relacionados que interactúan entre sí. Esta agrupación inicial es crucial en el diseño, como se mencionó anteriormente. En este caso particular, el sistema de computadora estará a cargo de tres grupos principales de tareas, que no son absolutamente independientes, pero tienen un alto grado de autonomía.

Un *primer grupo* debe encargarse de garantizar la estabilidad de temperatura de la muestra, indicando si ésta se encuentra dentro del rango deseado; debe permitir un barrido automático de la temperatura; y debe, además, reportar periódicamente la temperatura de dicha muestra a la interfaz gráfica. Como referencia rápida, este grupo será referido en lo sucesivo como *Control*.

Un *segundo grupo* de tareas debe encargarse de contar los eventos detectados y debe decidir si dichos eventos son válidos, en función de la velocidad (que debe ser constante) y la temperatura (que debe ser la indicada para el canal que está siendo procesado); este grupo debe encargarse, además, de reportar de alguna forma los eventos contabilizados. Este grupo será referido en lo sucesivo como *Contador*.

Por último existe un *tercer grupo* de tareas, con bajas restricciones de tiempo, compuesto por la interfaz gráfica, el registro de datos y la interfaz de red.

Estos tres grupos de tareas no son absolutamente independientes:

El *Control* debe recibir la configuración desde la interfaz gráfica y debe deshabilitar el contador si la temperatura no es la deseada.

El *Contador* recibe inhibiciones desde el *Control* y el control de velocidad externo del pickup, ya que el conteo debe deshabilitarse cuando el pickup recorre el camino de regreso (esta señal es provista por el control de posición disponible). Debe además enviar a las tareas de baja prioridad los resultados obtenidos.

El grupo de tareas de baja prioridad envía la configuración deseada al *Control* y recibe los resultados del *Contador*, además de ser el grupo de tareas encargado de la interacción con el usuario (los dos grupos anteriores no interactúan directamente con éste).

Si bien estos grupos de tareas no son absolutamente independientes (lo cual facilitaría mucho el diseño), su nivel de autonomía se considera suficiente como para adoptar en lo sucesivo la agrupación propuesta.

Dadas las herramientas presentadas en los capítulos anteriores, resulta evidente que el grupo de tareas de bajo nivel de prioridad debe ser manejado por el Linux POSIX.1, que dispone de la versatilidad necesaria en lo que se refiere a interfaces gráficas, acceso a medios de almacenamiento y a redes de datos. Estas tareas no necesitan más hardware que una placa Ethernet (para la conexión a la red local), algunos megabytes del disco rígido disponible (para almacenamiento) y un display tipo VGA (para la implementación de una interfaz con el usuario del tipo *XWindows*). Todos estos elementos están actualmente disponibles en una PC standard.

El *Control* requiere, en cambio, de hardware adicional. El calefactor tiene montada una termocupla (que requiere de un amplificador adecuado) y, dada la precisión requerida de 1/5 de grado hasta 800°C (4000 niveles lógicos), se muestra la utilización de la configuración presentada como ejemplo en el *Capítulo IV*, en el cual se montó sobre una placa *μDAQ* un control de temperatura de precisión variable con termocupla y salida PWM. Para su aplicación con este tipo de calefactores, es necesario modificar la etapa de salida del control, para implementar un control tipo PWM con ciclos enteros de línea. De esta forma, para suministrar la potencia al calefactor se utiliza solo un transformador con un elemento activo funcionando

[27] Para la tensión de línea, la frecuencia de la potencia es de 2x50Hz. Por lo tanto el período es de 10mseg, que es cinco órdenes menor que la constante de tiempo del calefactor.

como llave. En los puntos siguientes se presenta la modificación de la placa *µDAQ* para que funcione con ciclos enteros.

El *Contador* debe ser también implementado con hardware externo. La frecuencia máxima mostrada en la *figura V.9* equivale a una onda cuadrada de 1.25MHz, lo que está fuera del alcance de los periféricos disponibles en la PC, como puerta serie o paralelo. Este *Contador* necesita además de una base de tiempo muy estable, para que el tiempo de conteo en cada temperatura sea constante. Se propone la implementación de una nueva placa *µDAQ*, en la cual puede hacerse uso de los dos timers/counters del 8051, que puede operar a 20MHz, con un ciclo de instrucción de 1/12.

A continuación se procede con una descripción más detallada de la implementación de las dos placas *µDAQ*, y luego se presenta la estructura de software utilizada para el resto del experimento.

El hardware

Para la solución del problema se propone la estructura que se presenta en la *figura V.13*. Dos placas *µDAQ* se encargan de los dos grupos de tareas con requerimientos importantes de tiempo: el *Control* y el *Contador*, tal cual se los enunció anteriormente.

El *Control* fue descripto como ejemplo de aplicación en el *Capítulo IV*, y solo se hicieron aquí pequeñas modificaciones para adecuarlo a esta aplicación (juego de comandos, salida para ciclos enteros de línea, gate *TEMP_OK* para el *Contador*). Estas modificaciones se describen en el punto siguiente y complementan la información presentada en el *Capítulo IV*. La estructura del kernel *µKeox* se mantuvo, y la distribución de tareas es similar.

El *Contador* fue implementado sobre la misma estructura de hardware/software. Las consideraciones y características principales se presentan también a continuación. En este caso también la plataforma *µDAQ*-*µKeox* fue reutilizada en su totalidad.

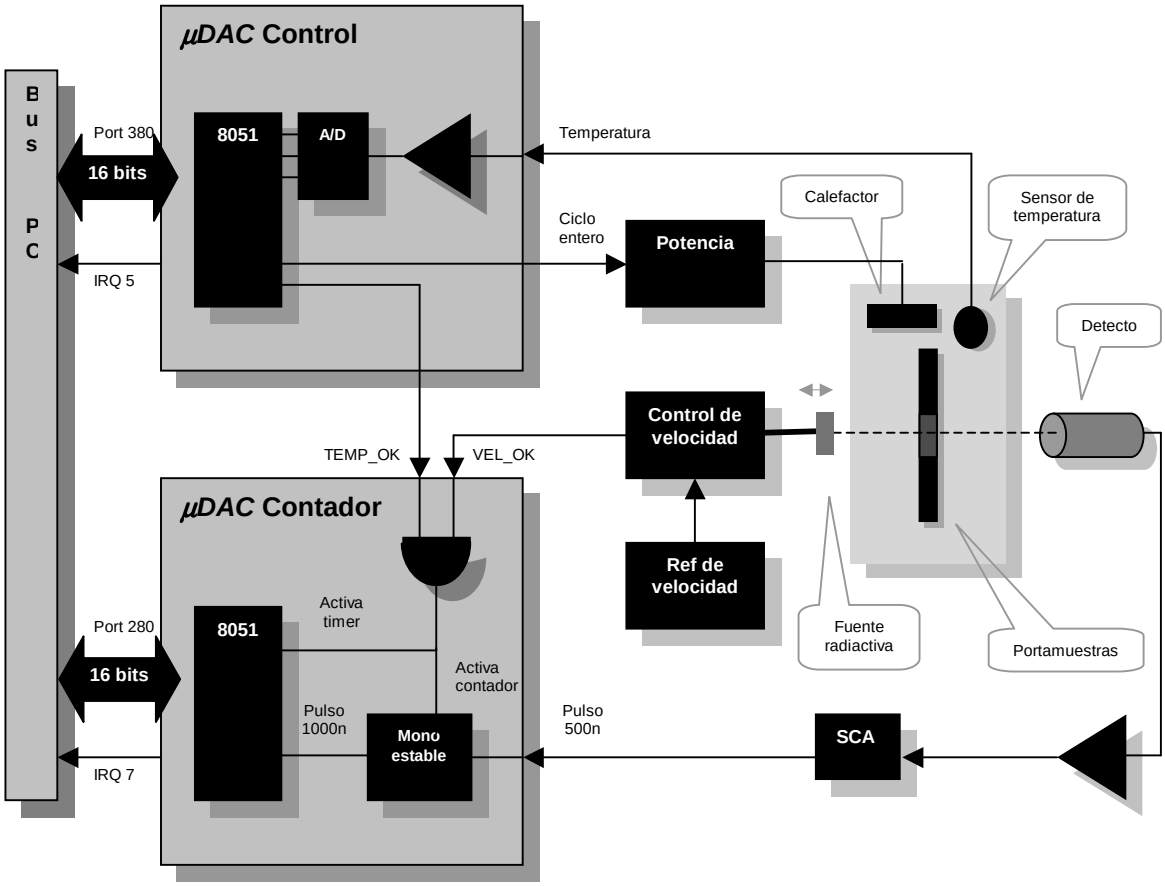


Figura V.13: Esquemático de la implementación propuesta

Como se mencionó anteriormente, el SCA, el control de velocidad programable y el calefactor de vacío están disponibles.

La etapa de potencia externa está compuesta por un transformador y una llave MOSFET, que permiten utilizar directamente la tensión de línea para entregar potencia al calefactor. Se utiliza la técnica de modulación de ancho de pulso, cuidando de conmutar la corriente entregada al calefactor durante los cruces por cero de la misma, para evitar interferencias electromagnéticas. Como ya se comentó, la constante de tiempo de calentamiento del mismo permite la utilización de esta técnica.

La estructura básica de funcionamiento está centrada en el sistema operativo de la PC. Si bien el grado de autonomía de las placas (con sus grupos de tareas asignadas) es alto, y el experimento podría llevarse a cabo con el control centrado en ellas, se prefirió delegar la evolución del experimento al sistema operativo de la PC. Esto acarrea la complicación de que la comunicación entre las placas y la PC tiene ahora requerimientos de tiempo para evitar la pérdida de datos. Pero la ventaja es que se mantiene el núcleo del experimento en el elemento más versátil y fácil de reconfigurar: el Linux. Esto es una ventaja en el caso de que se presenten modificaciones no previstas en el experimento, lo cual sucederá seguramente. Un cambio de estrategia, estando el control centralizado en una de las placas (programada en código assembler) llevaría a que dichos cambios consumirían una excesiva cantidad de tiempo de reprogramación. Por lo tanto, se asignaron las tareas con requerimientos estrictos de tiempo a las placas *µDAQ*, pero conservando la estructura principal del experimento en un programa de usuario de Linux.

Así, la placa encargada del control de temperatura (*Control*) recibe la temperatura de referencia desde Linux, a través del bus. Dicha placa deberá consumir los recursos necesarios del microcontrolador correspondiente, para garantizar dicha temperatura hasta que el control central indique un cambio. Además se requiere que, periódicamente, reporte a través del bus la temperatura de la muestra, para poder informar al operador a través de la interfaz de usuario. Esta temperatura es solo indicativa y Linux solo verifica que no exceda los límites de alarma, en cuyo caso se aborta la ejecución del experimento.

Sin embargo, la misión más importante de esta placa es la de indicar de alguna forma a la placa contadora si la temperatura está dentro del rango deseado por el control central (±0,2°C de la temperatura correspondiente al canal en desarrollo). En caso contrario deberá inhibir el conteo de los eventos. Para ello se optó por una señalización directa entre las placas. La placa controladora de temperatura comanda una línea TTL que se conecta a la placa contadora, funcionando como inhibidor directo del contador. De esta forma se evita una señalización por software de un mensaje de tan alta importancia. Este es uno de los *GATE* del contador, que se indica como *TEMP_OK* en la *figura V.13*. Un uno lógico indica que la temperatura es la deseada por el control central, y un cero lo contrario.

Por último, la placa encargada del control de temperatura debe disponer de un juego de comandos que permitan al control central Linux (a través del bus AT de la PC) indicar la temperatura de referencia, arrancar y detener el control, apagar la potencia en caso de alarma, modificar la sensibilidad del *GATE*, y cambiar la tasa de envío de la temperatura a través del bus.

La segunda placa *µDAQ*, encargada del conteo de los eventos (*Contador*), debe ejecutar las siguientes tareas: contar los pulsos provenientes del SCA (siempre que la temperatura y la velocidad sean correctas) y descargar periódicamente los registros del contador a través del bus, hacia el Linux. Para ello se utiliza un segundo *GATE* por hardware, disponible en el control de velocidad programable, que indica cuándo la velocidad es la programada. Esta señal se indica como *VEL_OK* en la *figura V.13*. Tanto *VEL_OK* como *TEMP_OK* deberán inhibir instantáneamente el conteo de eventos al tomar el nivel lógico cero.

Para la implementación del contador de eventos se utilizaron algunas características propias de los *counters/timers* del 8051, que se muestran luego. Esta placa deberá también disponer de un juego de comandos que permita al control central la operación. Linux debe ser capaz de arrancar y detener el conteo, limpiar los registros del contador y establecer el período de descarga.

Nótese que ambas placas comunican al Linux los resultados obtenidos a través del bus, con la misma estrategia. En la inicialización, el control central decide la tasa de datos que desea obtener de cada placa y las programa para que a intervalos constantes presenten los datos en el bus, generando una interrupción de la PC, sin la intervención posterior del control central. Esta técnica permite desligar al Linux de hacer polling sobre los dispositivos en búsqueda de los datos, pero lo obliga a estar preparado para la recepción de éstos.

Con estos dos grupos de tareas asignados a las dos placas, el control central de la PC debe encargarse de lo siguiente. A través de la interfaz gráfica (o algún otro medio) debe obtener del operario el rango de barrido de la temperatura y el paso de ésta, como así también el tiempo deseado de acumulación de

cuentas en cada canal (esto último dependerá de la actividad de la fuente radiactiva y la estadística deseada). Una vez que el control externo de velocidad programable está calibrado a la velocidad deseada (que se mantendrá constante a lo largo del experimento), el control central, luego de las inicializaciones correspondientes, comienza el desarrollo automático del experimento. Para cada canal debe programar la placa controladora de temperatura y esperar una cierta cantidad de descargas de la placa contadora. Entre tanto, debe encargarse de una interfaz gráfica que mantenga informado al operario del desarrollo del experimento y presentación de los datos hasta ahora obtenidos para su consulta remota. El control central debe encargarse, además, de monitorear el estado general del experimento, para detectar condiciones predeterminadas de alarma. Concluido el barrido de temperatura, el control central debe presentar los datos en forma standard, a través de una interfaz adecuada de usuario.

En los dos puntos siguientes se presentan algunos detalles adicionales acerca de la implementación de las dos placas *µDAQ*. Luego se presentan las técnicas utilizadas para la implementación del programa de control central en Linux.

1. La placa *µDAQ Control*

Como se mostró en el ejemplo de aplicación del capítulo anterior, se puede utilizar una placa *µDAQ* para implementar un control de temperatura. Para el sensado de la variable de interés se utiliza una termocupla tipo K. Con los timers internos del microcontrolador se implementa un conversor A/D doble rampa de precisión variable. Luego, utilizando una librería disponible de operaciones en punto flotante [Spinelli, 1996] se calcula el algoritmo PID digital. Por último, se utiliza la técnica de modulación de ancho de pulso con ciclos enteros de la tensión de línea como salida. La resolución del control puede ser de hasta 16 bits.

Las cuatro tareas que debe ejecutar el microcontrolador presente en esta placa, junto con sus asignaciones a las interrupciones disponibles son las siguientes:

Interrupción externa 0: Conmutación de la rampa del conversor A/D.

Interrupción externa 1: Señal desde la PC, y tarea de comunicación con el bus AT.

Interrupción del timer 0: Base de tiempo del conversor A/D, y tarea de manejo de datos y algoritmo de control PID.

Interrupción del timer 1[28]*:* Detección del cruce por cero de la tensión de línea, y tarea de manejo del modulador de ancho de pulso.

La asignación de prioridades a las tareas se hizo según el criterio *rate-monotonic*[29], como se mostró en el *Capítulo I*. La estructura del kernel *µKeox* fue utilizada para proveer un entorno preemptivo basado en prioridades, y la comunicación entre tareas se realizó utilizando memoria compartida.

Solo una de las tareas fue modificada respecto de la estructura presentada en el capítulo anterior. La salida del control anteriormente implementada utilizaba una fuente de tensión constante y aplicaba la técnica PWM para regular la potencia aplicada al calefactor, como puede verse en la *figura IV.5*. En este caso se modificó para comandar la etapa de potencia descripta anteriormente.

A la tarea de control de temperatura se le agregó el accionamiento de una señal TTL cuando la temperatura esta dentro de un cierto rango, que puede ser programado a través del bus. Esta se utiliza para habilitar el conteo en la otra placa (*TEMP_OK*).

El juego de comandos accesible a través del bus, asociados a la interrupción externa 1, puede verse en la *tabla V.1*. La recepción de una palabra de 16 bits por parte de la placa, en los ports asociados, activa el comando correspondiente[30], con la prioridad adecuada. El rango FFFFh-FD00h se reserva para su interpretación como comandos y el rango FCFFh-0000h se utiliza para enviar la temperatura deseada desde la PC.

---

[28] La interrupción del timer fue reprogramada para operar como interrupción externa, utilizando la técnica de utilizar el modo contador hasta uno.

[29] La tarea de menor duración recibe la más alta prioridad.

[30] Para mayor detalle del mecanismo de decodificación de direcciones de la placa, ver el *Apéndice II*.

Comando	Acción
FFnn	Divisor en hexadecimal (cada nn muestras envía una al bus). Rango 0-255 (default 1)
FEnn	Sensibilidad del Gate nn décimas de grado en hexadecimal. Rango 0-255 (default 2)
FD00	SET: Reporta la temperatura de set actual
FD01	STOP: Detiene el control de temperatura
FD02	START: Reinicia el control de temperatura
mmmm	Cambia la temperatura de set a mmmm décimas de grado. Rango 0-12000 [0000h-FCFFh] (default 0000h)

Tabla V.1: Juego de comandos para la placa Control

El esquemático completo de la placa puede verse en el *Apéndice II*, incluyendo el conversor A/D y la etapa de potencia. En el *Apéndice III* se presenta el listado completo de las tareas asignadas a la placa, junto con la implementación completa del ejecutivo de tiempo real.

En la *figura V.14* puede verse una fotografía de la placa terminada. Los conectores BNC presentes son:

CZ (entrada): Señal TTL proveniente del detector óptico de cruce por cero (montado junto con la fuente de alimentación y la etapa de potencia)

PWM (salida): Señal TTL de conmutación de la llave del transformador.

TEMP_OK (salida): Señal TTL para la inhibición del *Contador*.

Cable de termocupla: Directo a la placa, sin conector.

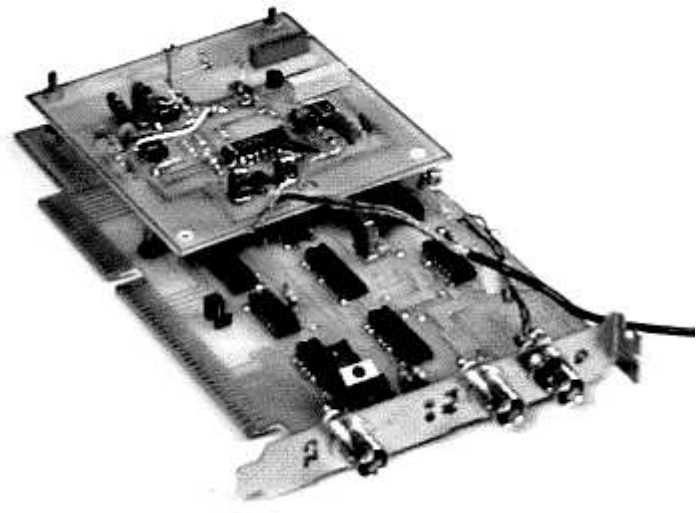


Figura V.14: La placa µDAQ Control

La etapa de potencia

Dadas las características de consumo y las constantes de tiempo del calefactor utilizado, se optó en este caso por entregar potencia al calefactor directamente desde la tensión de línea, utilizando un transformador con una llave optoacoplada, comandada directamente desde la placa *Control*. La tensión de línea que se entrega al calefactor debe conmutarse durante los cruces por cero de ésta, para evitar interferencias electromagnéticas. Para el sensado de los cruces por cero de la tensión de línea se utiliza un comparador optoacoplado que excita una de las interrupciones externas del microcontrolador. La tarea asociada a esta interrupción se encarga del algoritmo de modulación de ancho de pulso. El esquemático de la electrónica asociada a la potencia puede verse en el *Apéndice II*. En la *figura V.15* puede verse el equipo construido. Los dos conectores BNC corresponden a la salida *CZ* y la entrada *PWM*, que se conectan a la placa *Control*. La bornera corresponde a la salida de potencia.

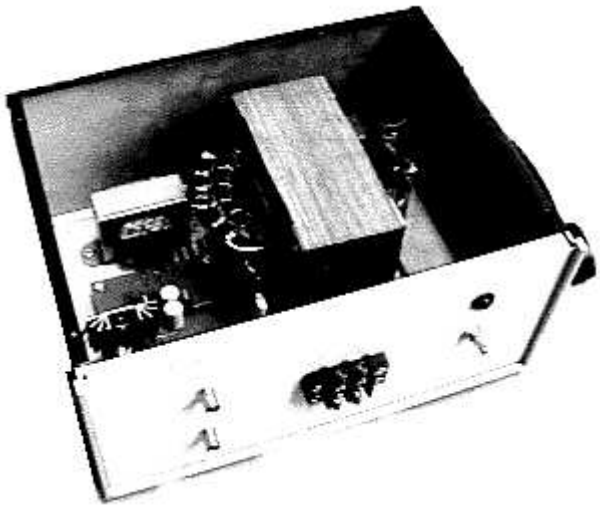


Figura V.15: Etapa de potencia

2. La placa *µDAQ Contador*

Para la implementación de este módulo también se reutilizó la estructura *µDAQ-µKeox* con la siguiente asignación tareas en sus interrupciones:

Interrupción externa 0: Inhibición por hardware del contador interno.

Interrupción externa 1: Señal desde la PC, y tarea de comunicación con el bus AT.

Interrupción del timer 0: Base de tiempo del contador y sincronización del experimento.

Interrupción del timer 1: Contador de eventos.

Con el microcontrolador funcionando a 20MHz, considerando que el ciclo de instrucción es de 12 ciclos de clock, y teniendo en cuenta que para que un evento sea contabilizado por uno de los *counters/timers* del 8051 debe durar al menos 1 ciclo de instrucción [Intel Corp, 1983], es necesario colocar un monoestable que garantice el ancho de los pulsos a contar. Este debe ser de al menos 2x12/20MHz, o sea 1.2µs. La máxima señal de salida del SCA puede tener 0.8µs (*figura V.9*), por lo tanto es necesario ensanchar dicho pulso. Recalculando, para una distribución de Poisson con una tasa de 1.2, la probabilidad de dos eventos es P(2)=0.8%. Esto significa un incremento de 0.5% en las pérdidas de eventos, lo cual se considera aceptable[31].

Utilizando la configuración interna de timers presentada en la *figura V.16*, puede utilizarse el *Timer 0* como base de tiempo y el *Timer 1* como contador, ambos de 16 bits.

La secuencia de funcionamiento es la siguiente: se programa T0 para que genere una interrupción periódicamente (T0 en modo 1 puede contar hasta FFFFh x 12 / 20MHz). Mientras tanto, T1 cuenta los eventos, si los gates están en nivel lógico 1. Cuando llega la interrupción asociada a T0, se envía al bus el contenido de los registros de T1. Cualquiera de los gates (*TEMP_OK* y *VEL_OK*) en nivel lógico 0 inhibe tanto el contador T1 como la marcha del timer T0. El período del timer 0 es programable desde la PC, y debe cuidarse de no hacerlo excesivamente largo, ya que puede desbordarse el contador T1. Esto dependerá de la tasa de arribo de eventos en cada experimento, lo que está directamente relacionado con la actividad de la fuente radiactiva.

[31] Para que el deterioro de la estadística sea nulo, debería utilizarse un microcontrolador de las mismas características, funcionando a 30MHz, no disponible actualmente.

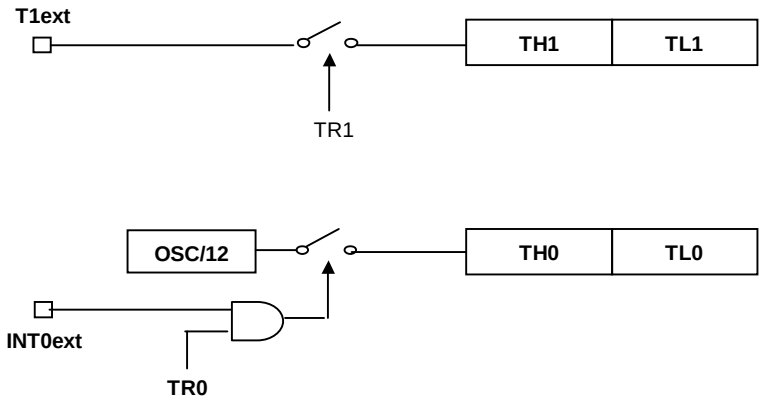


Figura V.16: Configuración de hardware utilizada para los timers del 8051

En la *figura V.17* puede verse la utilización del monoestable LM74121 para el ensanchado del pulso proveniente del SCA. Se muestra también la implementación por hardware de los gates para el contador. Si *TEMP_OK* ó *VEL_OK* son cero, se inhibe tanto el monoestable (a través de A1 y A2), evitando la llegada de pulsos al Timer 1, como el Timer 0 (a través de INT0).

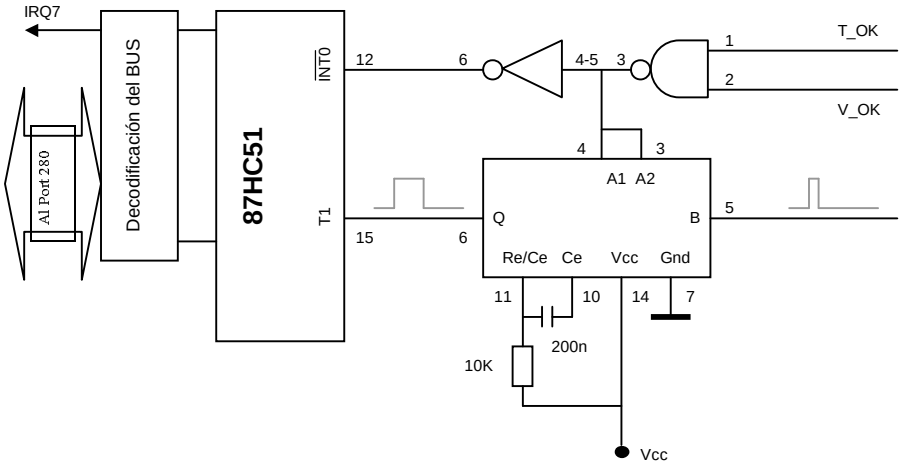


Figura V.17: Esquemático de la placa contadora

Al igual que para la placa anterior, para el comando del *Contador* desde el bus se implementaron los comandos que se muestran en la *tabla V.2*. Dichos comandos son palabras de 16 bits en binario, enviados desde la PC a través del bus. El rango FFFFh-FE00h se reserva para comandos y el rango FDFFh-0000h se utiliza para enviar el ciclo de conteo deseado desde la PC.

Comando	Acción
FF00	STOP: Detiene el contador y el timer
FF01	START: Arranca el timer y el contador
FF02	NOGATE: Deshabilita los gates de velocidad y temperatura
FF03	GATE: Habilita los gates de velocidad y temperatura
FF04	RESET: Limpia el contador y reinicializa el timer
FEnn	Divisor: cada nn períodos de conteo reporta uno al bus
mmmm	Cambia el número de ciclos de instrucción para el período de conteo a mmmm

Tabla V.2: Juego de comandos para la placa Contador

El esquemático completo de la placa puede verse también en el *Apéndice II,* y en el *Apéndice III* se presenta el listado completo de las tareas. En la *figura V.18* puede verse una fotografía de la placa terminada. Los conectores BNC presentes son:

C (entrada): Señal TTL proveniente del SCA.

VEL_OK (entrada): Señal TTL de inhibición proveniente del control de velocidad externo.

TEMP_OK (entrada): Señal TTL de inhibición proveniente del *Control.*

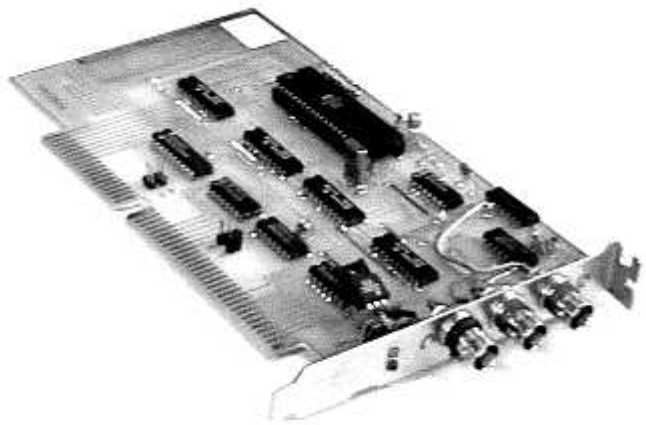


Figura V.18: La placa μDAQ Contador

El software

Como se adelantó en el capítulo anterior, existen dos posibilidades bajo Linux para comandar las placas *μDAQ*. La primera es utilizar la capacidad POSIX.1 del sistema, implementando los respectivos *device drivers*. La segunda opción es utilizarlas dentro del entorno RTLinux. Ambas tienen sus ventajas y desventajas.

La opción POSIX.1 es altamente portable y de simple implementación en el caso de disponer de los *device drivers* para los dispositivos. Como las placas *μDAQ* fueron desarrolladas en nuestro laboratorio, los drivers correspondientes no están disponibles, lo cual significa una complicación adicional. Sin embargo, Linux provee información suficiente[32] y las herramientas necesarias para la implementación de device drivers de caracteres [Rubini, 1998]. La estructura básica de éstos se mostró en el *Capítulo III*. En la estructura propuesta para la implementación POSIX.1, las placas ejecutan sus propias tareas dentro de un esquema preemptivo, basado en prioridades, con las características propias de tiempo real del conjunto *μDAQ-μKeox*. Sin embargo, la comunicación de las placas con el resto del sistema se hace a través de llamadas Linux, dentro de un esquema de tiempo compartido (*time-slice*). Por lo tanto es de esperar que existan retardos indeseables cuando se ejecutan las llamadas *read* y *write* a los drivers de las placas, ejecutados por el programa de usuario. Dada la baja tasa de transferencia entre las placas y Linux, ésta puede ser una solución viable, dentro de un rango acotado de aplicaciones. Esta opción fue implementada para su experimentación, teniendo en cuenta sus limitaciones. En la forma de funcionamiento presentada anteriormente, en el caso de una lectura al driver (no sincronizada con la operación de las placas) deberá

[32] Esta es una de las características que hacen al Linux un sistema operativo apropiado para el desarrollo de aplicaciones, frente a otros sistemas comerciales.

aguardarse a la llegada de una interrupción que valide los datos presentes en el port. Esta forma de funcionamiento tiene las características indeseables del polling. Las lecturas y escrituras a los drivers estarán conviviendo en el mismo nivel de jerarquía que la interfaz gráfica y los accesos a la red, por lo que tareas de bajo nivel de prioridad que signifiquen una carga importante para la CPU podrían ocasionar la pérdida de datos provenientes de las placas.

Con la otra opción, el entorno RTLinux, se sacrifica la portabilidad (ya que ésta es una solución propietaria de Linux y sólo existe para esta plataforma). La ventaja principal es que utilizando una tarea RTLinux, instalada en el kernel como control central del experimento, pueden jerarquizarse las tareas ejecutadas en el sistema operativo. De esta forma pueden priorizarse los accesos al hardware frente a las tareas de segundo orden de importancia. La existencia de estructuras del tipo FIFO para la comunicación entre la tarea de tiempo real y una aplicación POSIX, permite la utilización de la totalidad de los recursos del sistema operativo, fuera del espectro de tiempo real. La tarea de tiempo real se encargará de recolectar los datos provenientes de las placas (con una prioridad alta) y ponerlos en diferentes FIFO, de donde serán tomados por una aplicación POSIX.1, en cuando sea posible. Esta estructura evita terminantemente la posibilidad de pérdida de datos, pero presenta al programador una nueva "capa" sobre la cual debe programarse una parte de la aplicación. Esta opción también fue implementada y sus características principales se presentan luego.

Estas dos opciones tratan, básicamente, con el acceso a los recursos de hardware, elemento que presenta las más importantes restricciones de tiempo. Para el resto del experimento (la aplicación POSIX.1, el acceso a través de la red y el registro de datos) se utilizaron herramientas standard de Linux, disponibles en cualquier distribución. Dichas herramientas se presentan en los *Puntos 2 y 3*, a continuación.

1a. La solución POSIX.1 para el acceso al hardware

Para la implementación de esta opción de acceso al hardware (cuyo diagrama en bloques puede verse en la *figura V.19*) se utilizó el driver genérico para la placa *µDAQ* presentado anteriormente, con las modificaciones correspondientes para satisfacer las necesidades de las placas *Control* y *Contador*. El funcionamiento de ambos drivers es tal como se describió en el *Capítulo III*, utilizando las facilidades *interruptible_sleep_on* y *wake_up* en la lectura. El funcionamiento está completamente basado en eventos (utilizando las interrupciones en ambos sentidos) por lo cual la utilización de los drivers no significa una carga importante para la CPU. El listado de los drivers puede encontrarse en el *Apéndice III*, junto con los archivos necesarios para compilarlo y utilizarlo.

Este mecanismo tiene la ventaja de presentar a las placas de hardware como dos archivos standard de UNIX, que pueden ser abiertos, escritos, leídos y cerrados. Como estos cuatro accesos a los archivos se hacen a través de la interfaz POSIX.1 de Linux, el programa de aplicación que utilice esta facilidad será portable entre sistemas. Otra ventaja es que, al ser POSIX.1 la interfaz con las placas, es posible programar la aplicación que las utiliza en cualquier lenguaje compatible con POSIX.1 (C, Pascal, Fortran, Perl, Tcl/Tk, Java, etc., todos ellos disponibles en cualquier distribución de Linux).

Sin embargo debe tenerse en cuenta que, al producirse una lectura sobre una de las placas disparada con la llamada *read* de POSIX.1, esta operación quedará pendiente hasta que se produzca la interrupción que valida los datos presentes en los ports de entrada/salida asociados (lectura *bloqueante*). Debe recordarse además que dicha llamada se produce en el esquema de *scheduling time-slice* de Linux, por lo cual es de esperar que existan retardos indeseables en la ejecución de dicha llamada. Este funcionamiento no satisface los requisitos básicos de los sistemas de tiempo real, pero dada la baja tasa de transferencia de datos, y la eficiencia del funcionamiento basado en eventos, es posible su utilización con ciertas restricciones. Las malas propiedades de tiempo real de los sistemas de tiempo compartido quedaron en evidencia en las mediciones presentadas al final del *Capítulo III*.

Ambos drivers se utilizaron en una primera versión POSIX.1. El control central del experimento se escribió en lenguaje C. Este tiene como misión principal la automatización de los diferentes pasos y el registro de datos en archivos standard. El acceso a través de la red se hizo utilizando las herramientas standard para TCP/IP de Linux. La interfaz gráfica para el usuario se escribió en el lenguaje interpretado de muy alto nivel Tcl/Tk. Este último tiene la ventaja de permitir la elaboración de interfaces muy complejas, con histogramas, menú desplegable y botones. Sin embargo, pudo notarse (en algunos casos extremos) que dicha interfaz gráfica (sobre todo los módulos de histograma dinámicos) produce un consumo excesivo de la CPU. Esto puede ocasionar la pérdida de datos provenientes de las placas *µDAQ*, debido a que todas las tareas (incluida la recolección de datos) se ejecutan dentro de un esquema de tiempo compartido.

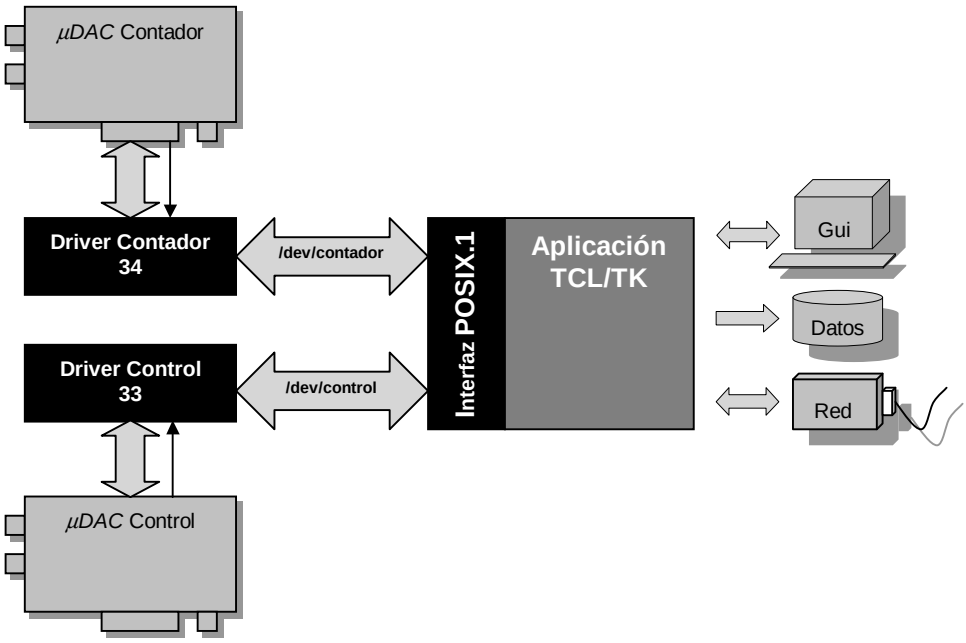


Figura V.19: Arquitectura de software para la implementación POSIX.1

Para la PC disponible (Pentium 120MHz) se encontró que la ejecución del software del experimento, con cuatro histogramas dinámicos, en conjunto con un acceso intensivo a la red y al disco duro, produce la pérdida de datos. Aunque este consumo es mucho mayor de lo necesario, y es poco probable que la CPU llegue a estar tan solicitada durante el transcurso del experimento, se optó por experimentar la opción RTLinux.

1b. La solución RTLinux para el acceso al hardware

Esta solución no produce efectos visibles respecto de la anterior. Simplemente garantiza la recolección de todos los datos provenientes de las placas. La estructura se modifica, utilizando los conceptos presentados en el *Capítulo III* y el *Capítulo IV*, como se muestra en la *figura V.20*. La aplicación POSIX.1 de Linux (presentada en el *punto 2*) es la misma, con modificaciones simples.

El módulo RTLinux tiene la estructura presentada al final del *Capítulo IV* para el acceso a una placa *µDAQ* genérica, modificado para manejar las dos placas necesarias en este caso: el *Control* y el *Contador*. El listado completo de este módulo se presenta en el *Apéndice III*. El funcionamiento de este módulo es simple, y se describe a continuación.

Primero crea cuatro FIFO para el intercambio de datos con Linux:

FIFO 1: datos provenientes de la placa *Contador*, enviados a Linux.
FIFO 2: datos provenientes de la placa *Control*, enviados a Linux.
FIFO 3: comandos provenientes de Linux para la placa *Contador*.
FIFO 4: comandos provenientes de Linux para la placa *Control*.

Estos FIFO son accesibles desde Linux como archivos */dev/rtf**. Desde el módulo RTLinux se acceden utilizando los comandos *rtf_put* y *rtf_get*, como se mostró en el *Capítulo III*. Nótese que los FIFO son de lectura o escritura, pero no ambas. Por lo tanto son necesario cuatro de ellos para comunicarse con las dos placas. La aplicación Linux debe escribir a un archivo y leer de otro. Este funcionamiento difiere del mostrado en el punto anterior, en el cual la lectura y la escritura se hacían a un mismo *device driver*. La aplicación Linux debe modificarse adecuadamente.

Una vez establecidos los canales de comunicación con Linux, el módulo RTLinux instala dos rutinas de atención de las interrupciones correspondientes a las placas. Dichas rutinas toman una palabra de 16 bits del bus ante la llegada de la interrupción y la colocan en el FIFO correspondiente. La aplicación Linux puede demorar todo lo que necesite en tomar estos datos, ya que los FIFO pueden hacerse tan grandes como sea necesario (y la RAM de la PC lo permita).

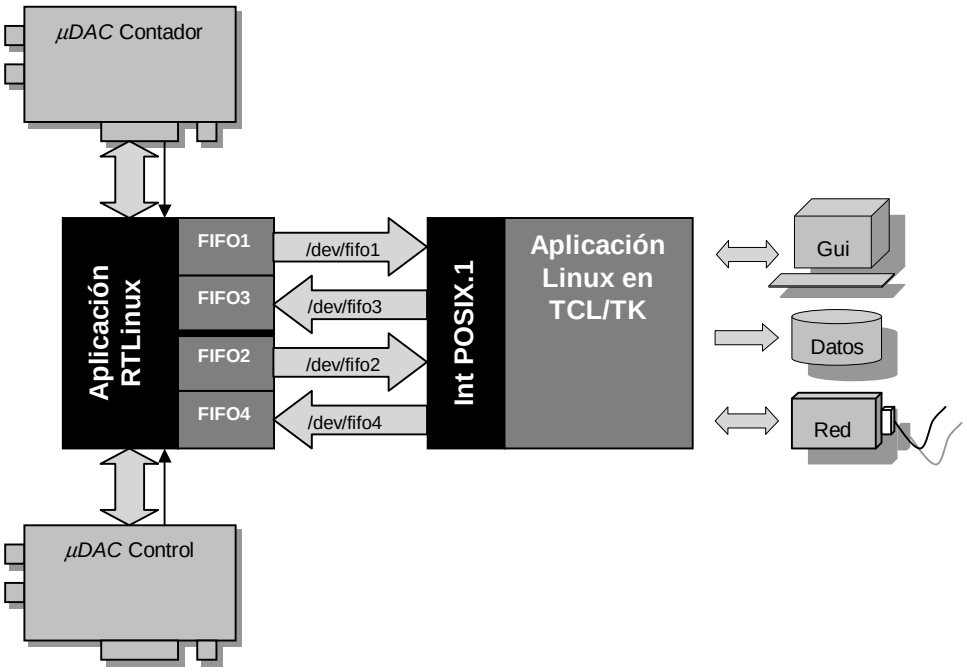


Figura V.20: Arquitectura de software para la implementación RTLinux

Para establecer el flujo de datos en el otro sentido (desde Linux hacia las placas), RTLinux provee la capacidad de instalar un *handler* (o manejador) para cada FIFO de lectura con *rtf_create_handler*. De esta forma se puede instalar una función, que será ejecutada cada vez que una palabra se presente en los FIFO que llevan datos desde Linux hacia las placas (FIFO3 y FIFO4). Esta tarea toma una palabra de 16 bits del FIFO y la envía al hardware asociado. Esta implementación tiene todas las ventajas del funcionamiento basado en eventos, en ambos sentidos. Por lo tanto, la carga ocasionada a la CPU es muy baja.

De esta manera, el módulo RTLinux se encarga de priorizar la transferencia de datos desde y hacia el hardware. Se agregó además el monitoreo de la temperatura del calefactor, independientemente del control central, para detectar situaciones peligrosas o de alarma. En este caso, el módulo RTLinux tiene la capacidad de abortar el experimento. Para esta implementación, ninguna otra tarea fue asignada a RTLinux, reservando el control central para la aplicación POSIX.1.

Si se analiza globalmente el funcionamiento de esta implementación, desde el punto de vista de la aplicación POSIX.1 que llevará adelante el experimento, puede verse que no varía mucho respecto de la implementación anterior. La diferencia principal es que cada pieza de hardware es accedida a través de dos archivos secuenciales (y no de uno, como el standard POSIX.1 requiere), uno de lectura y uno de escritura. Esta característica la aparta del standard, pero en lo que se refiere a la implementación del control central, las modificaciones son mínimas. Lo que sí es sacrificado definitivamente es la portabilidad. Esta implementación no es posible fuera del entorno Linux-RTLinux, ya que esta facilidad no está disponible para otros sistemas.

Un comentario adicional debe hacerse respecto de la implementación de este módulo. Desde el punto de vista de la automatización, sería posible utilizar al módulo RTLinux como control central del experimento. Esta posibilidad se descartó debido a que las herramientas disponibles para la programación dentro de dichos módulos son escasas y poco versátiles. Tal como se comentó respecto de la posibilidad de incluir el control central en los *device drivers*, esta posibilidad se descartó también en este caso, ya que un futuro cambio de estrategia del experimento significaría una reprogramación de los elementos de bajo nivel. Esto consumiría, sin dudas, una cantidad importante de tiempo. Se prefirió, en cambio, la delegación del control central a la aplicación POSIX.1 de Linux, al igual que en la implementación anterior.

El listado completo del módulo RTLinux, junto con las herramientas necesarias para el compilado e instalación del mismo, pueden verse en el *Apéndice III*.

2. El control central

Una vez garantizado el acceso al hardware en un entorno conocido y predecible, ya sea por medio de la solución POSIX.1 o la solución RTLinux, debe procederse al manejo de los datos en un nivel más alto. Las características de este experimento presentan algunos requisitos de sincronización que pueden satisfacerse eficientemente en un entorno de programación de alto nivel. Dada la autonomía de las placas *µDAQ* encargadas del *Control* y el *Contador*, es necesaria la implementación de un nuevo elemento, encargado de la evolución del experimento, denominado *control central*. Este elemento debe interaccionar tanto con el usuario (para obtener los parámetros del experimento –rango de barrido de la temperatura, tiempo de acumulación de eventos para cada canal, etc.) como con el hardware, a través de los *device drivers* o el módulo RTLinux (para llevar adelante la adquisición de datos). También debe encargarse del registro de los datos obtenidos, tanto para el procesamiento posterior, como para el acceso remoto, a través de la red local.

Dada la diversidad y la complejidad de las operaciones involucradas, es necesaria la utilización de un lenguaje de programación que disponga de dichas facilidades como elementos standard. Una solución posible es la utilización del lenguaje de programación C. Este es el lenguaje nativo de los sistemas UNIX, es compatible con POSIX.1 y dispone de una gran cantidad de librerías de funciones. Sin embargo, se trata de un lenguaje de nivel intermedio, en el cual la programación de interfaces gráficas y accesos a través de la red pueden llegar a ser muy complejos.

Otra opción es la utilización de algún lenguaje de programación interpretado de alto nivel, como Perl o Tcl/Tk, que disponen de módulos gráficos y de red reutilizables, de simple programación. La versatilidad y la capacidad de reutilización de dichos módulos llevan, sin dudas, a una menor eficiencia en la ejecución, sobre todo en los lenguajes interpretados. Sin embargo, la utilización de lenguajes de nivel intermedio para la programación de aplicaciones de muy alto nivel necesitan de un nivel de excelencia en las técnicas de programación que no está disponible en este caso. No solo se requiere de un esfuerzo enorme para la programación en nivel intermedio, sino que de no hacerse muy eficientemente, los resultados son de una calidad inferior a los obtenidos con un lenguaje de alto nivel. Tal es el caso de programación de interfaces gráficas utilizando las librerías Xwindows o Motif para C. En este caso, la programación de un histograma gráfico que acepte datos en punto flotante puede llevar varias páginas de código, que estará muy propenso a errores. Este elemento deberá ser someterse a un depurado (*debugging*) intensivo antes de su utilización en la aplicación. En cambio, utilizando un lenguaje interpretado de alto nivel, es posible reutilizar módulos de histograma, previamente escritos y depurados por diferentes autores. Es posible que este módulo disponga de facilidades que no serán utilizadas en este caso, pero si se trata de un módulo de alta calidad (de los cuales hay varios bien conocidos) esto no afectará la performance. De este modo, la utilización de un histograma en un lenguaje interpretado de alto nivel se reduce a la inclusión de sólo una línea en el script. La instancia adicional de interpretación de dicha línea se diluye, sin dudas, en la performance, pues para la programación de dicho módulo se han utilizado técnicas fuera del alcance de esta tesis.

En el entorno de aplicaciones de adquisición de datos y control, se ha difundido la utilización del lenguaje *Tool Command Language* (Tcl), desarrollado por John K. Ousterhout [Ousterhout, 1999]. Se trata de un lenguaje interpretado orientado a strings, con características similares al Perl. Su éxito se debe a dos razones fundamentales. La primera es la existencia de la extensión Toolkit (Tk), del mismo autor. Esta extensión permite un manejo simple y eficiente de los elementos básicos de las interfaces standard tipo Windows (botones, canvas, etc.). El segundo motivo de su éxito es el más importante. Esta herramienta mantuvo una filosofía abierta de desarrollo (a pesar de provenir de una firma comercial), poniendo a disposición de los interesados toda la información necesaria para realizar aportes a este producto. Contempla, además, mecanismos simples para realizar extensiones modulares. Este fue el detonante para la aparición de muy diversos paquetes de software que se complementan a la perfección con el conjunto Tcl/Tk, agregándole funcionalidad al lenguaje. Tal es el caso de la extensión BLT que agrega varios módulos gráficos (histogramas, gráficos xy), vectores en punto flotante, y otras facilidades de utilidad en un entorno de laboratorio. Una interfaz gráfica Tcl/Tk con elementos de la extensión BLT puede verse en la *figura V.21*.

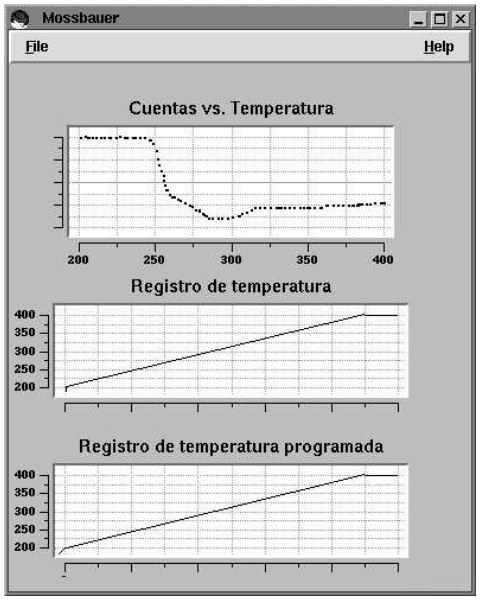


Figura V.21: Interfaz gráfica Tcl/Tk para el usuario

Otra ventaja de utilizar lenguajes modernos de alto nivel es que contemplan algunas facilidades necesarias para la programación de entornos gráficos basados en eventos, no disponibles en los lenguajes clásicos. Tal es el caso de la facilidad *fileevent* de Tcl, por ejemplo. Esta permite asociar una función a un archivo (y por lo tanto a uno de los elementos de hardware descriptos). Dicha función se ejecuta cuando el archivo está en condiciones de ser leído. Mientras tanto la lectura queda suspendida. Este funcionamiento es equivalente a instalar una interrupción de software en el dispositivo de hardware. Por lo tanto pueden evitarse el polling sobre los dispositivos y las lecturas bloqueantes. Esta facilidad no está disponible en C, y si se deseara implementarla, requeriría de una programación intensiva.

En caso que se desee ejecutar una secuencia de comandos que debe tener una buena performance, se dispone de las fuentes del intérprete Tcl/Tk, escrito en C. Por lo tanto pueden crearse nuevos comandos (secuencia de instrucciones en C) que se compilan dentro del intérprete. Esta técnica fue implementada en la presente aplicación para tratar de mejorar los tiempos de acceso al hardware a través de los *device drivers*. Sin embargo, no se obtuvo una mejora evidente, por lo que se concluye que el código presente en el intérprete es de las mismas características que el agregado.

Solucionado el tema de la interfaz gráfica, resta analizar la implementación del registro de datos y el acceso a través de la red. Dadas las características normalizadas de las interfaces POSIX.1 a archivos en UNIX, el registro de datos es una tarea standard, no conflictiva. Todos los lenguajes de programación (incluido Tcl, por supuesto) tienen la capacidad de manejar archivos.

En lo que respecta al acceso a los datos y al control del experimento, utilizando la red local e Internet, los lenguajes modernos de alto nivel proveen herramientas útiles, pero de una complejidad elevada. Es posible la utilización del protocolo TCP/IP y sus características asociadas para la implementación de canales de comunicación remotos. Sin embargo, el trabajo en las distintas capas de los protocolos de red requiere de una cuidadosa programación, fuera del alcance de esta tesis. Se optó, debido a esto, por la utilización de herramientas de más alto nivel.

3. La interfaz de red

Utilizando el intercambio de información por medio de archivos, dentro del entorno POSIX.1, se explotaron los utilitarios de red disponibles en Linux. Se trata de paquetes de software, algunos de ellos incluidos en la distribución base de Linux. Son programas de usuario, como *telnet* o *ftp*, que pueden ser automatizados utilizando técnicas de *shell programming*, como se describió en la *Introducción*. Un ejemplo primitivo de estas facilidades sería utilizar el servidor de FTP (File Transfer Protocol) para obtener desde una computadora remota (con sistema operativo Windows y una herramienta de FTP) un archivo de datos obtenido durante el transcurso del experimento. Otro ejemplo podría ser la utilización de una herramienta de telnet desde una Macintosh, para generar una consola de la computadora que

comanda el experimento, y así enviar comandos a éste. Otra posibilidad es publicar los resultados en forma de páginas de web, utilizando el servidor HTTP (Hyper Text Transfer Protocol) [W3C, 1999] de Linux. Estos pueden ser leídos utilizando un navegador (Netscape o Explorer) desde cualquier computadora conectada a la red.

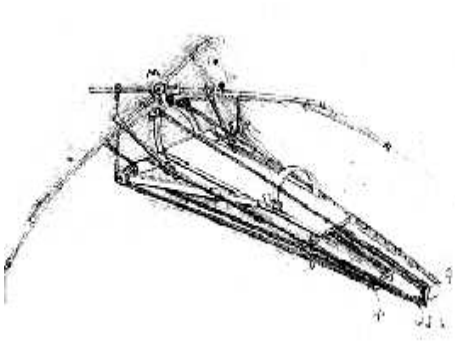
Un elemento fundamental de los sistemas UNIX es el entorno gráfico Xwindows. Este tiene la capacidad de ejecutarse localmente, presentando su salida tanto en la consola local como en la consola de una computadora remota, siempre que disponga de un cliente de X. Es posible entonces, sin programación adicional, presentar el entorno gráfico completo del experimento en una computadora que se encuentra fuera del laboratorio.

Existen herramientas más avanzadas, como por ejemplo las extensiones CGI-BIN [W3C, 1999] o PHP3 [PHP, 1999] para el HTTP. Estas permiten la construcción de páginas web interactivas, definiendo un grupo de comandos que pueden ser ejecutados desde diferentes páginas diseñadas para el acceso remoto al experimento.

Debe mencionarse en este punto el lenguaje semi-interpretado Java, de Sun Microsystems [Sun, 1999]. Este posee la capacidad de recibir a través de la red programas que se ejecutan en la CPU local. Este agrega todo un nuevo espectro de funcionalidad a las aplicaciones remotas.

I do not fear computers. I fear lack of them.

Isaac Asimov



Conclusiones

En sus primeros capítulos esta tesis se encargó de explorar las diferentes posibilidades para la construcción de sistemas de adquisición de datos y control, utilizando el sistema operativo Linux como la herramienta de implementación. Primero se estudiaron los aspectos más importantes de los sistemas de tiempo real y de los sistemas UNIX en general, en un recorrido por el estado del arte de los temas de interés. A continuación se presentaron las muy diversas herramientas disponibles en el entorno compuesto por una PC ejecutando el sistema Linux y su extensión de tiempo real RTLinux. Una vez estudiada la capacidad de dicho entorno, y detectadas sus limitaciones, se propuso un nuevo elemento de hardware/software, que permite el manejo de tareas con estrictos requerimientos de tiempo en ese contexto.

Este nuevo elemento, en conjunto con la enorme cantidad de recursos disponibles en Linux (provenientes de su filosofía abierta de desarrollo y la estandarización POSIX), proveen un entorno muy versátil de desarrollo para aplicaciones de automatización y control.

Si bien el conjunto propuesto no tiene la funcionalidad necesaria como para ser considerado un sistema completo de tiempo real (y mucho menos un sistema POSIX.4), fueron provistas las características funcionales necesarias para la implementación de un experimento con restricciones importantes de tiempo. Se proporcionó capacidad de tiempo real y hardware dedicado a dicha implementación en el punto en que era indispensable (para el acceso a las variables del experimento), y funcionalidad POSIX.1 con hardware standard donde era necesario normalizar las interfaces para simplificar la programación (interfaces gráficas y acceso a la red).

La arquitectura de hardware se llevó a cabo utilizando elementos standard, disponibles en el mercado local, evitando utilizar tecnologías propietarias o muy recientes, que puedan hacer peligrar la disponibilidad de los componentes a largo plazo. El hardware de la PC mantiene una estructura clásica, por encima de los avances tecnológicos. Si bien el bus AT puede llegar a desaparecer en futuros modelos de PC, la implementación de las placas *μDAQ* para el bus PCI (u otro que pudiera aparecer) no debería ser complicada. Los microcontroladores de 8 bits se han transformado en un standard para aplicaciones industriales, y nada hace sospechar que pudieran discontinuarse.

Respecto de la arquitectura de software, puede decirse que se utiliza un número elevado de lenguajes de programación (assembler en las placas, C en los drivers/módulos de tiempo real, y Tcl/Tk para las tareas sin restricciones de tiempo); sin embargo, todos ellos fueron utilizados en un nivel moderado de complejidad. Con conocimientos medios de programación en cualquier lenguaje (por ejemplo C), es posible comprender las estrategias utilizadas.

La utilización de tres lenguajes de programación diferentes tiene su base en que el software del experimento está programado en tres niveles distintos:

El software para los *embedded systems* (placas *µDAQ* con microcontrolador, kernel *µKeox* y tareas asignadas) está escrito en assembler. Esto garantiza que los tiempos estén acotados, dentro de un esquema preemptivo, basado en prioridades. Al no utilizarse ningún compilador, no existen implementaciones desconocidas de funciones, cuyos tiempos de ejecución no puedan calcularse. La estructura del kernel de tiempo real garantiza la ejecución de las tareas en el orden adecuado y con tiempos conocidos.

Los *device drivers* POSIX.1 y los módulos RTLinux son programados utilizando una versión restringida de C, sin funciones externas ni librerías. Esto permite la programación en un lenguaje estructurado, sin alejar demasiado al programador del hardware.

Por último, fuera del campo del tiempo real, la aplicación de usuario está programada en un lenguaje de alto nivel, y se ejecuta dentro de un esquema de tiempo compartido. Esto hace que sea casi imposible calcular la duración exacta de una tarea.

Nótese que el nivel más bajo y más alto de programación se corresponden con las tareas con mayores y menores restricciones de tiempo, respectivamente. La interacción con el hardware es crítica para la evolución del experimento, mientras que la interfaz gráfica puede tolerar demoras significativas. Esto lleva a una conclusión evidente, pero que no está de más mencionarla en este caso. Las características de tiempo real se mantendrán más fácilmente cuando se mantenga la programación cercana al hardware. Cuanto más alto sea el nivel de abstracción del hardware, más lejos se encuentra el programador de éste, y por lo tanto se pierden las características de tiempo real.

Ante esta situación, en la estructura presentada hay una ventaja evidente. Las tareas de más alto nivel se encargan de interactuar con el hardware más complicado del experimento, como son las placas y protocolos de red, placas de video, monitores y discos duros. Por el otro lado, las tareas de tiempo real, y por lo tanto de bajo nivel, se encargan de interactuar con el hardware de laboratorio, diseñado específicamente, y más fácil de manejar. Por lo tanto es importante destacar que se elevó el nivel de complejidad del conjunto sólo en los puntos críticos, en los cuales es estrictamente necesaria una performance de tiempo real.

Otra relación inversamente proporcional que puede verse es la que mantienen las características de tiempo real frente a la portabilidad entre sistemas. El único standard para sistemas de tiempo real, que garantizaría la portabilidad de aplicaciones entre ellos, es POSIX.4 (el cual debe considerarse como una utopía por el momento). No existe aún un conjunto de hardware/software que sea 100% compatible con POSIX.4. Sólo existen implementaciones parciales. Mientras tanto, han aparecido diversas soluciones al problema de tiempo real, todas ellas fuera del standard e incompatibles entre sí. Por lo tanto, cuanto mayor sea la funcionalidad de tiempo real de un sistema, menor será su portabilidad, por el momento. La solución RTLinux no es una excepción a esta regla. El código de un módulo RTLinux sólo es útil para un kernel de Linux, debidamente recompilado.

Una propiedad importante de los elementos presentados es su modularidad. Estos tres elementos están compuestos por una parte de hardware y una de software, estando sus interfaces perfectamente definidas. Cada elemento opera en un nivel de abstracción diferente, más cerca o más lejos del hardware y programados en más bajo o más alto nivel. Sus características se resumen en la siguiente tabla.

Elemento	Hardware	Software	Interfaz con elemento siguiente
Bajo Nivel	Placa *µDAQ*	Kernel *µKeox*	Set de comandos en hexadecimal
Nivel Intermedio	Bus AT	Drivers POSIX.1/Módulos RTL	Filesystem compatible POSIX.1
Alto Nivel	IBM PC	Aplicaciones POSIX	Archivos Linux

Cualquiera de estos tres elementos puede ser reemplazado por otro, o bien nuevos elementos pueden ser agregados, mientras se mantengan las interfaces hacia los elementos de nivel superior e inferior. La estructura presentada permite, por ejemplo, reemplazar las placas *µDAQ* por implementaciones completamente diferentes, siempre que continúen recibiendo los comando y enviando los datos a través del bus, con el formato especificado.

Tal es el caso del proyecto final de grado de un alumno de la Facultad de Ingeniería, UNLP [Jackson, 1999]. En este proyecto se tomó como marco de referencia el presentado en esta tesis, para la elaboración

de un nuevo elemento de hardware para incorporar a futuros experimentos. Se trata de un sensor de temperatura, con un conversor A/D de 12 bits, conectado a la puerta paralelo de la PC, funcionando ésta en modo EPP. El driver de la puerta paralelo fue modificado para soportar el nuevo hardware. Se agregó, entonces, un nuevo elemento de bajo nivel (el conversor con interfaz de puerta paralelo) y un nuevo elemento de nivel intermedio (el device driver correspondiente). De esta manera, la aplicación Tcl/Tk que comanda el experimento puede utilizar dicho sensor de temperatura accediéndolo como si se tratara de un archivo POSIX.

Otro ejemplo de la modularidad de estos elementos es la posible utilización de instrumentos de medición con interfaz GPIB. Un driver POSIX.1 genérico para placas controladoras GPIB standard está disponible en Linux, por lo tanto el bus GPIB (y por lo tanto todos los instrumentos conectados a él) pueden accederse como archivos secuenciales.

Haciendo un análisis global del sistema, los tres elementos presentados podrían interpretarse como tres *clases*, en una estructura de razonamiento *orientada a objetos*. El punto clave de esta comparación son la funcionalidad y las interfaces bien definidas de los tres elementos. Podría así definirse las tres clases M*ódulo de Instrumentación de Bajo, Medio y Alto Nivel*. Cada clase presenta diferentes instancias u *objetos*. Por ejemplo, la placa *Control* es un objeto perteneciente a la clase *Módulo de Bajo Nivel*. Su interfaz está definida por los comandos que se envían a través del bus y su funcionalidad está bien definida. Por lo tanto podría reemplazarse dicho objeto por otro, cuyo funcionamiento interno sea completamente diferente, mientras respete la funcionalidad y las interfaces.

Una modificación del experimento que será considerada a corto plazo es el agregado de una tercera placa *μDAQ* que contenga el control de velocidad del pickup con referencia programable. Sería deseable agregar funcionalidad a éste, tal que permita tanto su utilización para experimentos de velocidad constante como de aceleración constante, manteniendo la comunicación a través del bus AT, tal que permita su manejo directamente desde la interfaz gráfica de usuario.

Otra modificación que se encuentra actualmente en análisis es la posibilidad de "cerrar" completamente el conjunto Linux-RTLinux en lo que se refiere a su acceso desde la consola (*embedded system*). Sería deseable montar la PC y las placas *μDAQ* en un módulo NIM (standard muy utilizado en el ámbito de trabajo del Dto. de Física de la UNLP), instalado directamente en el rack del experimento, conservando sólo los accesos remotos. Sin embargo, esto sólo será posible luego de transcurrido un período de prueba prolongado, que permita a los usuarios ganar "confianza" sobre los nuevos recursos disponibles.

Un campo de estudio que queda abierto a partir de este trabajo es el agregado de funcionalidad de tiempo real a los módulos RTLinux. Algunos trabajos han aparecido recientemente en este tema. Por ejemplo existe una extensión que permite la utilización de la puerta serie directamente desde un módulo, otra que permite utilizar memoria compartida, semáforos y SVIPC, y otra que proporciona una librería de operaciones en punto fijo [RTLinux Home Page, 1998]. También se encuentran disponibles diversos módulos que permiten reemplazar la técnica de *scheduling* utilizada por RTLinux. Sería deseable agregarle funcionalidad GPIB a RTLinux, como así también acceso a una red y a archivos de tiempo real. Esto es posible, una vez más, gracias a la filosofía abierta de desarrollo adoptada por los autores de RTLinux.

Las computadoras son inútiles. Solo pueden dar respuestas.

Pablo Picasso

Apéndice I
La licencia pública GNU

En este apéndice se transcriben dos documentos de distribución pública: la *Licencia GNU* y el *Manifiesto GNU*. Entre ambos expresan la filosofía que ha sido el pilar sobre el cual se han desarrollado miles de herramientas de software de altísima calidad, que complementan muy eficazmente al Linux, haciéndolo un sistema operativo completo y versátil. Debe aclararse que el Linux propiamente dicho (el kernel y el sistema operativo base) no adhiere a la licencia GNU, sino que tiene su propia licencia, en otros términos, pero bastante similar. En cambio, la mayor parte de los paquetes de software que pueden encontrarse en las diferentes distribuciones comerciales de Linux (Red Hat, Slackware, etc.) suscriben a ésta licencia, que expresa los términos básicos de distribución y garantía para dichos paquetes de software.

El software desarrollado para la implementación de éste experimento, los drivers, y los módulos de tiempo real, son también distribuidos bajo la *GNU General Public License*, cuyo texto se transcribe sin modificaciones (excepto de formato) lo cual valida dicha licencia.

El otro documento que se transcribe es el *Manifiesto GNU*, un documento escrito por Richard Stallman, en los comienzos del proyecto GNU, para solicitar apoyo y participación en el proyecto. Se transcribe también en su versión original, como acostumbra acompañar a los paquetes de software que suscriben a la *Licencia GNU*.

GNU General Public License

```
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and conditions for copying, distribution and modification

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

> Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

> You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you

provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

> These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.
>
> Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.
>
> In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

> The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.
>
> If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

> If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.
>
> It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.
>
> This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

> Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT

LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

El Manifiesto GNU

A continuación se presenta el *GNU Manifesto*, escrito por Richard Stallman, en los comienzos del proyecto GNU, para solicitar apoyo y participación en el proyecto. Este documento transmite la filosofía que fue el punto de partida para el desarrollo de miles de herramientas de software de muy alta calidad, sin las cuales el éxito de Linux no hubiera sido tan grande.

What's GNU? Gnu's Not Unix!

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it. Several other volunteers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use TeX as our text formatter, but an nroff is being worked on. We will use the free, portable X window system as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer file names, file version numbers, a crashproof file system, file name completion perhaps, terminal-independent display support, and perhaps eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at machines in the 68000/16000 class with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

To avoid horrible confusion, please pronounce the `G' in the word `GNU' when it is the name of this project.

Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per liter of air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

Some Easily Rebutted Objections to GNU's Goals

> "Nobody will use it if it is free, because that means they can't rely on any support."
>
> "You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.

We must distinguish between support in the form of real programming work and mere handholding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distribution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just hand-holding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

> "You cannot reach many people without advertising, and you must charge for the program to support that."
>
> "It's no use advertising a program people can get free."

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?

> "My company needs a proprietary operating system to get a competitive edge."

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefiting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.

> "Don't programmers deserve a reward for their creativity?"

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

> "Shouldn't a programmer be able to ask for a reward for his creativity?"

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like

the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented--books, which could be copied economically only on a printing press--it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

"Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies--such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referee we've got does not seem to object to fights; he just regulates them ("For every ten yards you run, you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fight.

"Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

> "We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

> "Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware, asking for donations from satisfied users, or selling hand-holding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

> Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.
>
> But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing--often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.
>
> The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.
>
> The consequences:
>
> - The computer-using community supports software development.
> - This community decides what level of support is needed.
> - Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany

productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

Hardware: partes de una computadora que pueden ser pateadas.

Jeff Pesis

Apéndice II
El hardware del experimento

En este apéndice se presentan los detalles constructivos de los diferentes elementos de hardware que componen el experimento. La primera parte es una descripción detallada acerca de la implementación de una placa *µDAQ* genérica, con todos los componentes reutilizables. En la segunda parte se presentan los esquemáticos de los circuitos de todas las partes construidas, incluyendo las dos implementaciones sobre la placa *µDAQ,* el control de velocidad y la etapa de potencia.

La placa *µDAQ* genérica

La placa es una interfaz standard para bus AT de la PC y se compone básicamente de cuatro partes:

Fuentes de alimentación.
Decodificador de direcciones.
Latch bidireccional de 16 bits.
Microcontrolador.

Los recursos del microcontrolador no utilizados (dos ports, control de timers y una interrupción externa) se pusieron disponibles en una zona libre de la placa preparada para la implementación de circuitos adicionales. Esta zona libre fue utilizada en la placa *contador* para montar la lógica de inhibición y el monoestable, y en la placa *control* para montar el conversor A/D, el mecanismo de autocero, el amplificador de termocupla y el sensor de temperatura ambiente (para más detalles ver el *Capítulo V*). Las conexiones con el exterior de dichos circuitos están previstas utilizando conectores BNC montados en el frente de la placa.

Fuentes de Alimentación

Utilizando las líneas del bus de 12V (positivo y negativo) se montaron en la placa dos reguladores (7805 y 7905) para obtener 5V positivo y negativo. Esto permite independizar a la placa de posibles fluctuaciones de la tensión de alimentación de 5V de la PC.

Si bien la fuente negativa no es utilizada en esta placa genérica, se montó previendo el caso en que se utilicen amplificadores operacionales (como se hizo en la placa *control*).

Se agregó en el frente un led indicador de 5V positivo.

Decodificador de direcciones

El decodificador de direcciones se implementó en forma standard, utilizando dos comparadores de 4 bits (2x74LS85), que conectados en cascada permiten comparar las seis líneas superiores de dirección (A4 a A9) y el Adress Enable (AEN) del bus con valores prefijados por *jumpers* en la placa. De esta manera, con diferentes combinaciones de dichos *jumpers*, se puede direccionar el latch de 16 bits en el port 200, 280, 300 o 380, a elección.

Además de los dos comparadores se implementó un bloque de lógica (implementado con 4 puertas NAND 74LS02) que cumple las siguientes funciones:

Ante cada direccionamiento del port en coincidencia con un ciclo de escritura del bus (IOW bajo) se accionan los clock de los latches que llevan el dato del bus a los ports del microcontrolador. Además se genera la interrupción externa del microcontrolador. En este caso se utilizó un monoestable (74LS121) para ensanchar la señal IOW del bus, tal que produzca la interrupción del microcontrolador.

Ante cada direccionamiento del port en coincidencia con un ciclo de lectura del bus (IOR bajo) se accionen las líneas OC de los latches que llevan el dato del microcontrolador al bus, accionando además la línea I/OCS16 del bus. Esta línea indica que la lectura debe hacerse en 16 bits, y debe ser accionada a través de un buffer tri-state (74LS244) para no afectar el funcionamiento del bus.

Latch

El latch de 16 bits bidireccional está implementado con cuatro flip-flops octales tipo D (4x74LS374), conectado como se muestra en el circuito esquemático. La palabra de 16 bits que va del microcontrolador al bus es cargada en los flip-flops 1 y 3 accionando los clocks de éstos, y es leída accionando los OC. La palabra que va del bus al microcontrolador es cargada en los flip-flops 2 y 4 accionando los clocks, y es leída accionando los OC.

Microcontrolador

Se utilizó la versión AT89C51 (Atmel) del microcontrolador 8051 de Intel. Es una implementación CMOS con EEPROM, que permite su funcionamiento con clocks externos de hasta 20MHz.

El microcontrolador se conectó al latch utilizando dos ports de 8 bits (P1 y P2). Utiliza la línea P0.1 para generar la interrupción y cargar el latch, y la línea P0.0 para leerlo. Todos los recursos no utilizados están disponibles para las diferentes implementaciones.

A la línea de interrupción se le agregó un juego de jumpers, que permite conectarla a las líneas de interrupción del bus 5, 7, 10 u 11, según se desee.

A la línea P0.2 se le conectó un led, que se montó en el panel frontal. Esta señal puede ser utilizada como indicador de actividad.

Esquemáticos

A continuación se presentan los siguientes esquemáticos:

MOSS_00: Diagrama en bloques y conexionado.
MOSS_01: Control analógico de velocidad.
MOSS_02: Etapa de potencia y Detector de cruce por cero.
MOSS_03: Entrada al contador y Generador de referencia de velocidad.
MOSS_04: Placa *µDAQ* genérica.
MOSS_05: Conversor A/D.

Real programmers can write assembly code in any language.

Larry Wall

Apéndice III
El software del experimento

En este apéndice se presenta el código correspondiente a las diferentes capas de software utilizadas en la implementación del experimento. Se muestran las dos opciones de acceso al hardware mencionadas en el *Capítulo V*: la implementación POSIX.1 y la implementación RTLinux, junto con ejemplos de su utilización. Por último se presenta el código de la interfaz de usuario, escrita en Tcl/Tk. El listado de las implementaciones *µKeox* para las placas no se presenta listado, debido a su extensión. Pueden encontrarse, junto con el resto de los programas, en el diskette que acompaña a esta tesis.

El driver POSIX.1 para la placa *µDAQ*

Se presenta a continuación el listado completo de un driver genérico para la utilización de la placa *µDAQ* en Linux. Recordemos que la funcionalidad de lectura de la placa es la siguiente: cada vez que tiene un dato disponible genera una interrupción de hardware. Esta facilidad se utiliza en el driver para sincronizar la lectura del dato. Cuando el programa de usuario ejecuta un *read* sobre la placa, esta operación se queda en espera hasta que una nueva interrupción llega. En ese momento la espera termina y el driver lee el dato del latch de la placa. Este funcionamiento se obtiene utilizando las colas de espera, disponibles en el kernel (wait queues).

En lo que respecta a la escritura, el procedimiento es más simple. El driver solo debe encargarse de escribir la palabra de 16 bits al port correspondiente a la placa *µDAQ*. Esta escritura genera una interrupción del microcontrolador presente en dicha placa. Es el microcontrolador quien debe ocuparse de proveer un "driver" para retirar del latch dicha palabra (que en general será de configuración) y tomar las decisiones correspondientes.

Debe recordarse que los valores de interrupción y port de entrada/salida utilizados dentro del código del driver (en este caso interrupción 7 y port 380) deben coincidir con los valores seteados por jumpers en la placa *µDAQ*. Lo mismo sucede con el número de identificación del driver. En este caso se utiliza un MAYOR igual a 33, que debe coincidir con el numero MAYOR de */dev/daq*.

A continuación se muestra el archivo de configuración necesario (*Makefile*) para compilar e instalar el driver utilizando la herramienta *make*, junto con un ejemplo de la utilización del driver desde un programa en C.

Los drivers específicos para las placas *Contador* y *Controlador* (implementadas utilizando sendas placas *µDAQ*) pueden encontrarse en el diskette que acompaña a esta tesis, junto con los *Makefile* y ejemplos de aplicación, listos para ser utilizados.

```
//*****************************************************
// udaq.c
// Character Device Driver genérico para la placa µUDAQ
//
// Crear el device en el filesystem con el comando:
//      mknod /dev/udaq c UDAQ_MAYOR UDAQ_MENOR
//
// Alejandro Veiga
// Primera versión: Junio de 1997
// Update: Febrero de 1999
//*****************************************************

// Kernel includes (ver man pages)

#include <linux/module.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/sched.h>
#include <linux/malloc.h>
#include <linux/ioport.h>
#include <linux/fcntl.h>
#include <linux/delay.h>
#include <time.h>
#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <sys/ioctl.h>

// Estos valores deben coincidir con el seteo de jumpers
// en la placa.
#define UDAQ_PORT     0x380
#define UDAQ_IRQ      7

// Estos valores deben coincidir con los de /dev/udaq
#define UDAQ_MAYOR    33
#define UDAQ_MENOR    0

// 1: verbose, 0: siencioso
#define UDAQ_DEBUG    1

// 1: ocupado, 0: libre
#define UDAQ_BUSY     1
#define UDAQ_FREE     0

// GLOBALES
// Wait queue utilizada por el read, para que la interrupción lo despierte.
struct wait_queue *udaq_wait_q;
// Declaración del formato de datos (16 bits)
unsigned short data_out;
int udaq_busy = UDAQ_FREE;
int irq = UDAQ_IRQ;


//=====================================================
//
// Rutina de atención de la interrupción proveniente
// de la placa µDAQ.
//
//=====================================================
static void
udaq_irq(int irq)
{
#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq_irq: interrupción recibida.\n");
#endif

       // Solamente despierto al read (si es que habia uno dormido)
       wake_up(&udaq_wait_q);

#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq_irq: read despertado.\n");
#endif
}
//=====================================================
```

```
//=====================================================
//
// Implementación de las llamadas
// open, read, write y close
//
//=====================================================

//=====================================================
static int
udaq_open(struct inode * inode, struct file * file)
//=====================================================
{
       unsigned int menor = MINOR(inode->i_rdev);
       int ret_code;

       // Verifico el valor de MENOR
       if (menor != UDAQ_MENOR)
              return -ENODEV;

       // Si no está ocupada, la ocupo.
       if (udaq_busy ==  UDAQ_BUSY)
              return -EBUSY;
      udaq_busy = UDAQ_BUSY;


       // Solicito la interrupción y verifico errores
       ret_code = request_irq(irq,(void *)udaq_irq,
                                          SA_INTERRUPT,"udaq",NULL);
       if (ret_code) {
              printk(KERN_WARNING "udaq: no pudo inicializarse la irq.\n");
              return ret_code;
       }

#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq_open: dispositivo abierto e irq \
                                          exitosamente instalada.\n");
#endif
       return 0;
}

//=====================================================
static int
udaq_read(struct inode * inode, struct file * file,
              char * buf, int count)
//=====================================================
{
       short data = 0xffff;

       // Solo acepta una palabra de datos a la vez.
       if(count!=1) return -EINVAL;

#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq_read: yendo a dormir.\n" );
#endif

       // Espero la interrupción de hardware
       interruptible_sleep_on(&udaq_wait_q);

#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq_read: despertado por irq.\n" );
#endif

       // Leo el dato que envía la placa hacia la memoria del kernel
       data = inb_p(UDAQ_PORT);

#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq_read: Dato leido: %d.\n", data);
#endif

       // Envío el dato a la memoria de programa utilizando el
       // buffer de caracteres
       put_fs_word(data, buf);

#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq_read: Dato enviado al programa.\n");
#endif

       return 0;
}
```

```
//=====================================================
static int
udaq_write(struct inode * inode, struct file * file,
              const char * buf, int count)
//=====================================================
{
       short data = 0;

       // Tomo la palabra a enviar desde la memoria de programa
       data = get_fs_word(buf);

#ifdef UDAQ_DEBUG
      printk(KERN_DEBUG "udaq_write: recibido: %d.\n", data);
#endif

       // Envío el dato al port
       outb_p(data, UDAQ_PORT);

#ifdef UDAQ_DEBUG
      printk(KERN_DEBUG "udaq_write: palabra enviada.\n");
#endif

       return 0;

}

/*=====================================================*/
static void
udaq_close(struct inode * inode, struct file * file)
/*=====================================================*/
{
       free_irq(irq,NULL);
       udaq_busy  = UDAQ_FREE;

#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq_close: dispositivo e irq liberados.\n");
#endif
}

//=====================================================
// Por último, la interfaz con el kernel:
// file_operations, init_module y cleanup_module
//=====================================================

//=====================================================
// Registro las operaciones en el kernel
//=====================================================
static struct file_operations udaq_fops = {
       NULL,          // seek no implementado
       udaq_read,
       udaq_write,
       NULL,          // readdir no implementado
       NULL,          // select no implementado
       NULL,          // control no implementado
       NULL,          // mmap no implementado
       udaq_open,
       udaq_close
};

//=====================================================
int
init_module(void)
//=====================================================
{
#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq:\n");
       printk(KERN_DEBUG "udaq:\n");
       printk(KERN_DEBUG "udaq: init_module: comenzando inicialización.\n");
#endif
       // Registro el device driver utilizando el número MAYOR
       if (register_chrdev(UDAQ_MAYOR, "udaq", &udaq_fops)) {
              printk(KERN_ERR "register_chrdev failed: goodbye world :-(\n");
              return -EIO;
       }
#ifdef UDAQ_DEBUG
       else   printk(KERN_DEBUG "udaq: driver registrado.\n");
#endif
       return 0;
}
```

```
//=====================================================
void
cleanup_module(void)
//=====================================================
{
#ifdef UDAQ_DEBUG
       printk(KERN_DEBUG "udaq: cleanup_module called\n");
#endif
       // Verifico que no esté ocupado
       if (udaq_busy) {
              printk(KERN_WARNING "udaq: dispositivo ocupado; el
                     módulo no pudo retirarse\n");
              return -EBUSY;
       }
       // Elimino del kernel
       if (unregister_chrdev(UDAQ_MAYOR, "udaq") != 0) {
              printk("udaq: cleanup_module failed\n");
              return -EIO;
       }
#ifdef UDAQ_DEBUG
       else
              printk(KERN_DEBUG "udaq: cleanup_module succeeded\n");
#endif

}
```

Makefile

Este archivo, junto con el utilitario *make*, permite el compilado e instalación del módulo correspondiente al *device driver* de la placa genérica. Nótese la utilización de los flags de optimización de módulos para el compilador.

```
CC=gcc
CFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer \
         -fno-strength-reduce -pipe -m486
MODCFLAGS := $(CFLAGS) -DMODULE -DMODVERSIONS -DCPU=486 -DEXPORT_SYMTAB \
             -D__KERNEL__ -DLINUX

all:           udaq.o

udaq.o: udaq.c udaq.h
                       $(CC) $(MODCFLAGS) -c -o udaq.o udaq.c

install:       udaq.o
                       insmod udaq.o
                       lsmod
```

Ejemplo de programa de aplicación en C

```
// Este ejemplo muy simple muestra cómo el nuevo
// dispositovo de Linux, accesible desde /dev/daq
// una vez que el módulo anterior está instalado,
// puede ser manejado en forma idéntica a un archivo.
#include <stdio.h>
#include <fcntl.h>
#include "udaq.h"

void main(){

       int fd;
       short data = 0x0fff;

       // Abro el dispositivo, verificando errores posibles
       fd = open("/dev/udaq",O_RDWR);
       if (fd<0) {
              perror("No pudo abrirse el dispositivo.");
              exit(-1);
       }
       printf("Dispositivo abierto.\n");

       // Escribo una palabra de 16 bits
       write(fd,&data,1);

       // Leo una palabra de 16 bits, sincronizado con la interrupción
       read(fd,&data,1);
```

```
        printf("Palabra leida: %d\n",data);

        // Cierro el dispositivo y salgo
        close(fd);
        exit(0);

}
```

La aplicación RTLinux

Una alternativa a la utilización de los drivers POSIX.1 se presentó anteriormente. Aquí se lista la aplicación completa, con la utilización de los cuatro FIFOs para el acceso a los datos de las placas μDAQ.

```
#define MODULE

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <linux/rtl_version.h>
#include <asm/io.h>
#include <asm/rt_irq.h>
#include <rtl_fifo.h>

#define CONTROL 0x380
#define IRQ_CONTROL 5

#define CONTADOR 0x280
#define IRQ_CONTADOR 7

unsigned short data;
unsigned short msg;

void int_contador(void)
{
        data = inw_p(CONTADOR);
        rtf_put(1,&data,2);
}

void int_control(void)
{
        data = inw_p(CONTROL);
        rtf_put(2,&data,2);
}

int hand_contador(unsigned int fifo)
{
        rtf_get(3, &msg, 2);
        outw_p(msg, CONTADOR);
        return 0;
}

int hand_control(unsigned int fifo)
{
        rtf_get(4, &msg, 2);
        outw_p(msg, CONTROL);
        return 0;
}

int init_module(void)
{
        rtf_create(1, 4000);
        rtf_create(2, 4000);
        rtf_create(3, 4000);
        rtf_create(4, 4000);
        rtf_create_handler(3, &hand_contador);
        rtf_create_handler(4, &hand_control);
        request_RTirq(IRQ_CONTADOR, int_contador);
        request_RTirq(IRQ_CONTROL, int_control);

        return 0;
}

void cleanup_module(void)
{
        rtf_destroy(1);
        rtf_destroy(2);
```

```
        rtf_destroy(3);
        rtf_destroy(4);
        free_RTirq(IRQ_CONTADOR);
        free_RTirq(IRQ_CONTROL);
}
```

Makefile

Este archivo, junto con el utilitario *make* (standard para todas las distribuciones Linux), permite el compilado e instalación del módulo listado anteriormente.

```
INCLUDE = /usr/src/rtl/include
CFLAGS = -O2 -Wall

all: app moss install

app: app.c
        gcc -I${INCLUDE} ${CFLAGS} -o app app.c

moss: rtl_moss.c
        gcc -I${INCLUDE} ${CFLAGS} -D__KERNEL__ -D__RT__  -c rtl_moss.c

clean:
        rm -f app rtl_moss.o

install:
        rmmod rtl_moss
        insmod rtl_moss
```

Ejemplo de aplicación en Tcl/Tk

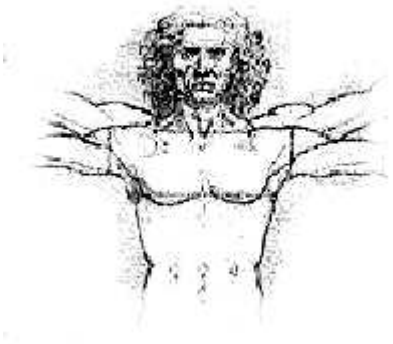
El siguiente programa de aplicación utiliza el módulo RTLinux presentado previamente, y provee una interfaz gráfica para el usuario. Esta escrito en el lenguaje de programación Tcl/Tk [Welch, 1997], disponible en todas las distribuciones de Linux.

```
#!/usr/bin/bltwish

namespace import blt::*

# Custom varialbles
set DIR "/home/moss/public_html"

# Definiciones para utilizar Jtools (Selector de archivos, prompt, etc.)
set jstools_library /usr/lib/jstools
set jstools_pkg [file join $jstools_library pkg]
set auto_path  [concat  [list $jstools_pkg]  [list $jstools_library]  $auto_path]
#set bg_color [j:prompt_colour_rgb]
#puts $bg_color
set color "#d5cbb3"

# Seteo de T_MAX T_MIN y T_STEP desde el archivo de configuracion
source ./Mossbauer.conf
set DIM [ expr (( $T_MAX - $T_MIN ) / $T_STEP) + 1 ]
set RUN 0
set TIEMPO_MAX 10000

set i 1
set temp 0
set cuentas 0
set cuentas_ 0
set canal 0
set tiempo 0
set set_point 0

set font "*-utopia-bold-r-normal--16-*-*-*-*-*-*-*"

set FIFO1 [open "/dev/rtf1" r]
set FIFO2 [open "/dev/rtf2" r]
set FIFO3 [open "/dev/rtf3" w]
set FIFO4 [open "/dev/rtf4" w]
set LOG1  [open "$DIR/contador.log" w]
set LOG2  [open "$DIR/control.log" w]
```

```
proc Contador {} {

        global cuentas_ cuentas FIFO1 i canal yVectH DIM RUN LOG1

    if { $RUN == 0 } {
        binary scan [read $FIFO1 2] s data
        set cuentas [expr $data * 40]
    }

    if { $RUN == 1 } {
        binary scan [read $FIFO1 2] s data
        set cuentas [expr $data * 40]
        set yVectH($canal) [expr $yVectH($canal) + $data]
        if { $i == 40 } {
        #       set cuentas "[ int $yVectH($canal)] c/seg"
                set i 0
                incr canal
        }
        if { $canal == $DIM } {
                SalvarTodo
                set RUN 0
                puts $LOG1 "END : [ exec date +%H:%M:%S ]"
                flush $LOG1
                set canal 0
        }
        incr i
    }

}

proc Control {} {
        global temp FIFO2 TIEMPO_MAX
        global tiempo set_point xVect yVectT yVectS

        binary scan [read $FIFO2 2] s data
        set temp $data
        set xVect(++end) $tiempo
         set yVectT(++end) $temp
         set yVectS(++end) $set_point
         if { $tiempo > $TIEMPO_MAX } {
                xVect delete 0
                yVectT delete 0
                yVectS delete 0
         }
         incr tiempo

}

proc SalvarTodo {} {
        global DIR T_MIN T_MAX T_STEP DIM xVectH yVectH
        set HTML1 [open "$DIR/contador.html" w]
        set HTML2 [open "$DIR/control.html" w]
        set DAT1  [open "$DIR/contador.dat" w]
        set DAT2  [open "$DIR/control.dat" w]

        puts $DAT1 "[ exec date +%H:%M:%S ]"
        puts $HTML1 "<TITLE>Corrida finalizada [ exec date +%H:%M:%S ]</TITLE>"
        puts $HTML1 "<BODY BGCOLOR=#CCCCCC>"
        puts $HTML1 "<H3>Histograma terminado [exec date +%H:%M:%S]</H3>"
        for {set j 0} {$j < $DIM} {incr j} {
                puts $DAT1 "$xVectH($j)  $yVectH($j)"
                puts $HTML1 "$xVectH($j)  ->  $yVectH($j) <BR>"}
        puts $HTML1 "</BODY>"

        close $HTML1
        close $HTML2
        close $DAT1
        close $DAT2
}

proc SendContador { param } {

         global FIFO3

         if { $param == 10 || $param == "Div2" } {
                 puts -nonewline $FIFO3 [ binary format s 65026 ]
                # 65024 + $divisor
         }
         if { $param == 4 || $param == "Reset" } {
                 puts -nonewline $FIFO3 [ binary format s 65284 ]
         }
         if { $param == 3 || $param == "NoGate" } {
                 puts -nonewline $FIFO3 [ binary format s 65283 ]
```

```
        }
        if { $param == 2 || $param == "Gate" } {
                puts -nonewline $FIFO3 [ binary format s 65282 ]
        }
        if { $param == 1 || $param == "Start" } {
                puts -nonewline $FIFO3 [ binary format s 65281 ]
        }
        if { $param == 0 || $param == "Stop" } {
                puts -nonewline $FIFO3 [ binary format s 65280 ]
        }
        if { $param > 10 } {
                puts -nonewline $FIFO3 [ binary format s [ expr 65525 - $param ] ]
        }

        flush $FIFO3
}

proc SendControl { num } {
       global FIFO4 set_point
       set set_point $num
       puts -nonewline $FIFO4 [ binary format s $num ]
       flush $FIFO4
}

# INIT
proc Init {} {
       global T_MAX T_MIN T_STEP DIM
       global xVectH yVectH LOG1 LOG2 canal i
       global xVect yVectT yVectS

       puts $LOG1 "INIT: [ exec date +%H:%M:%S ] "
       puts $LOG2 "INIT: [ exec date +%H:%M:%S ] "
       flush $LOG1
       flush $LOG2

       set canal 0
       set i 1

       # Inicializacion de vectores de registro de temperatura
       vector xVect(0)
       vector yVectT(0)
       vector yVectS(0)

       # Inicializacion de vectores para el histograma
       vector xVectH($DIM) yVectH($DIM)
       set acc $T_MIN
       for {set j 0} {$j < $DIM} {incr j} {
              set xVectH($j) $acc
              set yVectH($j) 0
              set acc [ expr $acc + $T_STEP ]
       }

}

# QUIT
proc Quit {} {
       global FIFO1 FIFO2 FIFO3 FIFO4 LOG1 LOG2

       SendControl 0

       SalvarTodo
       puts $LOG1 "QUIT: [ exec date +%H:%M:%S ] "
       puts $LOG2 "QUIT: [ exec date +%H:%M:%S ] "
       close $FIFO1
       close $FIFO2
       close $FIFO3
       close $FIFO4
       close $LOG1
       close $LOG2

       destroy .
}

# Inicializaciones varias
Init

#####################################################################
# GUI

# Color de fondo y titulo
. configure -bg gray
wm title . "Mossbauer"
```

```
frame .menu -relief raised -borderwidth 2
menubutton .menu.help -text Help -menu .menu.help.m -underline 0
menu .menu.help.m
.menu.help.m add command -label "About" -underline 0 \
        -command {  }
menubutton .menu.file -text File -menu .menu.file.m -underline 0
menu .menu.file.m
.menu.file.m add command -label "Run" -underline 0 \
        -command {
               if {$RUN == 1} { j:alert -text "Already running"};
               if {$RUN == 0} { set RUN 1;
               puts $LOG1 "RUN : [exec date +%H:%M:%S]";
               flush $LOG1 }
               }
.menu.file.m add command -label "Stop" -underline 0 \
        -command {
               if {$RUN == 0} { j:alert -text "Already stopped"};
                if {$RUN == 1} { set RUN 0;
               puts $LOG1 "STOP: [exec date +%H:%M:%S]";
               flush $LOG1 }
               }
.menu.file.m add command -label "Reset" -underline 0 \
        -command { Init }
.menu.file.m add command -label "Cambiar Temp" -underline 0 \
        -command {
               set temp_set [j:prompt -text "Entre la nueva temperatura" \
                -title "Temperatura"];
               SendControl $temp_set;
               puts $LOG1 "TEMP: [exec date +%H:%M:%S ] to $temp_set";
               flush $LOG1
               }
.menu.file.m add command -label "Quit" -underline 0 \
        -command { Quit }

label .cuentas -textvariable cuentas -font $font -relief sunken -bg $color \
       -width 8
label .temp -textvariable temp -font $font -relief sunken -bg $color \
        -width 8
# Plot estilo Fisica
graph .h -title "Cuentas vs. Temperatura" -bg gray \
        -plotbackground white  \
        -height 2i -width 5i
.h grid configure -hide no
.h axis configure x -min $T_MIN -max $T_MAX
.h element create T -xdata xVectH -ydata yVectH -linewidth 0 \
        -symbol cross -pixel 0.025i \
        -color #222222 -label ""

graph .gT -title "Registro de temperatura" -bg gray \
        -plotbackground white  \
        -height 1.8i -width 5i
.gT axis configure y -min $T_MIN -max $T_MAX
.gT grid configure -hide no
.gT element create T -xdata xVect -ydata yVectT -symbol "" \
        -color #AA0000 -label ""

graph .gS -title "Registro de temperatura programada" -bg gray \
        -plotbackground white  \
        -height 1.8i -width 5i
.gS axis configure y -min $T_MIN -max $T_MAX
.gS grid configure -hide no
.gS element create T -xdata xVect -ydata yVectS -symbol "" \
        -color #00AA00  -label ""

# PACK
table . .menu 0,0 -fill x -cspan 10
table .menu .menu.file 0,0
table .menu .menu.help 0,25
table . .cuentas 1,0 -cspan 1
table . .temp 1,1 -cspan 1 -padx {0 5}
table . .h 2,0 -cspan 2 -padx {0 15}
table . .gT 3,0 -cspan 2 -padx {0 15}
table . .gS 4,0 -cspan 2 -padx {0 15}

############################################################
# Inicializacion del Contador y el Control. Arranque.
SendControl [ expr 65280 + 10 ]
SendControl 0
SendContador 41660
fileevent $FIFO1 readable Contador
fileevent $FIFO2 readable Control
```



Siglas utilizadas

ANSI: American National Standards Institute

API: Application Program Interface

ASCII: American Standard Code for Information Interchange

ASIC: Application-Specific Integrated Circuit

ATM: Asynchronous Transfer Mode

BASIC: Beginners All-purpose Symbolic Instruction Code

BSD: Berkeley Source Distribution

C: lenguaje de programación

C++: extensión al lenguaje C

CAD: Computer Aided Design

CGI: Common Gateway Interface

CISC: Complex Instruction Set Computer

CPU: Central Processing Unit

DAQ: Data Acquisition

DHCP: Dynamic Host Configuration Protocol

DNS: Distributed Name Service

DSP: Digital Signal Processing

EISA: Extended ISA

EPP: Enhanced Parallel Port

FTP: File Transfer Protocol

GCC: Gnu C Compiler

GIF: Graphics Interchange Format

GNU: GNU 's Not UNIX

GPIB: General Purpose Interface Bus

GUI: Graphical User Interface

HTML: HyperText Markup Language

HTTP: HyperText Transfer Protocol

IEEE: Institute of Electrical and Electronics Engineers

ISA: Industry Standard Architecture

IP: Internet Protocol.

IPC: Inter-Process Communication

IRQ: Interrupt Service Request

ISBN: International Standard Book Numbering

ISDN: Integrated Services Digital Network

ISO: International Organisation for Standardisation

LAN: Local Area Network

MS-DOS: Microsoft Disk Operating System

NFS: Network File System

OSI: Open Systems Interface

PC: Personal Computer

PCI: Peripherial Component Interconnect

PID: Proporcional, integral y derivativo

POSIX: Portable Operating System Interface

PWM: Pulse Widht Modulation

QNX: producto comercial

RAM: Ramdom Acess Memory

RISC: Reduced Instruction Set Computer

ROM: Read Only Memory

RS232: Standard para el puerto serie de comunicaciones

RT-IX: producto comercial

SCA: Single Channel Analiser

SCO: Santa Cruz Operation

SCSI: Small Computer Systems Interface

SMB: Server Message Block

SPEC: Standard Performance Evaluation Corporation

SV: System V

SVIPC: System V Inter-Process Communications

SVR3: System V release 3

SVR4: System V release 4

Tcl: Tool command language

TCP: Transfer Control Protocol

URL: Uniform Resource Locator

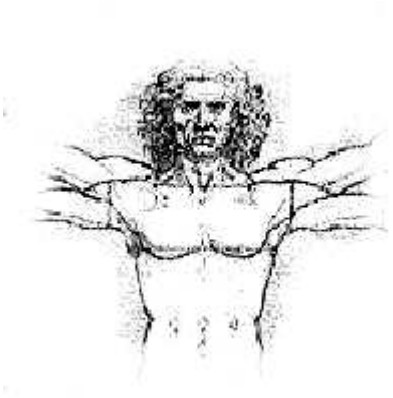
VGA: Video Graphic Adapter

VLSI: Very Large Scale Integration

VXWorks: producto comercial

WAN: Wide Area Network

WWW: World-Wide Web



Bibliografía

A.S.A. *Model VF-1000 Vacuum Furnace. Instruction Manual.* 1985.

A.S.A. *Model K3 Mössbauer Spectrometer Linear Motor. Instruction Manual.* 1985.

Bach, Maurice. *The design of the UNIX operating system*. Prentice Hall. 1986.

Barabanov, M. y V. Yodaiken. *Introducing Real Time Linux*. Linux Journal # 34. 1997.

Baruch, R. y C. Schoretrer. *Writing Character Device Drivers for Linux.* Linux Lab Project Documentation. 1994.

Battaioto, P., P.E.Battaiotto, J.M.Catalfo, N.Martínez, M.A.Mayosky, G.M.Toccacelli. *A 96-bit DSP-based Mossbauer Spectrometer*. IECON 94. 20th. International Conference on Industrial Electronics, Control and Instrumentation. Boloña, Italia, septiembre de 1994.

Bauman, P. *The Linux Serial Programming HOWTO.* Linux Documentation Project. 1997.

Brent, Welch. *Practical Programming in Tcl and Tk.* ISBN: 0-13-616830-2. 1997.

Buttazzo, Giorgio. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers. 1997.

Chien, C.E. *Mössbauer study of a binary amorphous ferromagnet* $Fe_{80}B_{20}$. Physical Review. Pp. 1003-1015. Volume 18, Number 3. Agosto de 1977.

Cranshaw, T.E. *Mössbauer Spectroscopy.* Journal of Physics E: Scientific Instruments. Pp. 497-505. Volume 7. 1974.

Dibble, Peter. *OS-9 Insights. Advanced Programmers Guide.* Microware Systems Corp. 1988.

Dijkstra, Edsger. *Cooperating sequential processes.* Technical Report EWD-123, Eindhoven, Holanda. Technological University. 1965.

EG&G ORTEC. *Model 551 Timing Single-Channel Analyzer. Operating and Service Manual.*

Egan, Janet y Thomas Teixeira. *Writing a Unix Device Driver.* John Willey & Sons, Inc. 1992.

Eggebretch, L. *Interfacing to the IBM Personal Computer.* Macmillan Computing Publishing. 1991.

Epplin, J. *Linux as an Embedded Operating System.* Embedded Systems Programming. Octubre 1997.

Gallmeister, Bill O. *POSIX.4 Programming for the real world.* O'Reilly & Associates, 1995.

González Vázquez, J. *Introducción a los Microcontroladores.* McGraw Hill, 1992.

Hwang, K. y F. Briggs. *Computer Architecture and Parallel Processing*. Mc Graw Hill. 1984.

IEEE. Institute of Electrical and Electronic Engineers. *Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (POSIX.1),* and *Amendment: Real time Extension (POSIX.4).* Institute of Electrical and Electronic Engineers. 1-55937-061-0.

Intel Corp. *MCS-51 Architectural Overview, Hardware Description of the 8051 and 8052, MCS-51 Programmers Guide and Instruction Set y MCS-51 Data sheets.* 1983.

Jackson, Gastón. *Interface de adquisición para el sistema operativo Linux.* Trabajo final, Facultad de Ingeniería, UNLP. 1999.

Kirch, Olaf. *Linux Network Administration.* Linux Documentation Project. 1993.

Laplante, Philip A. *Real-Time Systems Design and Analisis.* IEEE Press, 1992.

Leffler, Samuel J., Kirk McKusick, mike Karels y John Quarterman. *The design and implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley, 1990.

Linux Documentation Project. http://sunsite.unc.edu/mdw/linux.html.

Linux Lab Project. http://www.llp.fu-berlin.de.

Markus, K. *Linux POSIX.1b Compatibility.* Linux Documentation Project. 1996.

Martínez Pérez, J. *Prácticas con Microcontroladores.* McGraw Hill, 1993.

Martínez, N., E. Spinelli y A. Veiga. *Control de temperatura para pequeños hornos cerámicos.* Anales de la 80º Reunión Nacional de la Asociación Física Argentina. 1995.

Martínez, N., E. Spinelli y A. Veiga. *Plataforma de bajo costo para la implementación de algoritmos de control digital.* Proccedings del VII Congreso Latinoamericano de Control Automático. 1996.

Martínez, N., E. Spinelli y A. Veiga. *Control y supervisión remota de experimentos de laboratorio.* Anales de la 82º Reunión Nacional de la Asociación Física Argentina. 1997.

Martínez, N. y E. Spinelli. *Control de velocidad con referencia programable.* Publicación LEICI número 03-37-99. UNLP. 1999.

Mohr, J. *The State of Linux.* Byte. Enero 1997.

Moses, J. *Is POSIX Appropiate for Embedded Systems*? Embedded Systems Programming. Julio 1995.

Oualline, Steve. *Practical C.* O'Reilly & Associates, Inc. 1993.

Ousterhout, John. *Personal Home Page.* http://www.scriptics.com/people/john.ousterhout/. 1999.

PHP. *PHP Home Page*. http://www.php3.net 1999.

QNX Software Systems Ltd. *QNX Operating System.* 1996.

RTLinux Home Page. http://www.rtlinux.org. Referencias a las extensiones citadas pueden encontrarse en ftp://rtlinux.cs.nmt.edu/pub/rtlinux/v1/drivers.

Rubini, Alessandro. *Linux Device Drivers.* O'Reilly & Associates, Inc. 1998.

Schroter, C. *Linux-GPIB User's Guide.* Linux Lab Project Documentation. 1996.

Spinelli, E. y A. Veiga. *Juego de rutinas de punto flotante y entrada/salida para el Microcontrolador 8051 de Intel.* Publicación LEICI 40-95-01, Facultad de Ingeniería, UNLP. 1995.

Spinelli, E. y A. Veiga. *Dual Slope A/D converter for the 8051.* Electronic Engineering, Enero 1996.

Stankovic, John A. *Real-Time and Embedded Systems.* Departament of Computer Science, University of Massachusetts, Amherst, MA 01003. 1985.

Stankovic, John A. y Krithi Ramamritham. *Tutorial Hard Real-Time Systems.* IEEE Computer Society. 1987.

Stankovic, John A. *Real-Time Computing.* Departament of Computer Science, University of Massachusetts, Amherst, MA 01003. Abril de 1992.

Stankovic, John A. y Krithi Ramamritham. *Advances in Real-Time Systems*. IEEE Computer Society. 1993.

Stevens, W. *Advanced Programming in the Unix Enviroment*. Addison Wesley. 1992.

Stewart, Z. *IBM Parallel Port FAQ/Tutorial*. ftp://ftp.rmii.com/pub2/hisys/parport. 1994.

Sun Microsystems. *Java Home Page*. http://java.sun.com. 1999.

Sweet, M. *Serial Programming Guide for Compliant Operating Systems*. http://easysw.com/~mike/serial. 1996.

Tanenbaun, Andrew S. *Sistemas Operativos. Diseño e implementación*. Prentice Hall Inc. 1988.

Tanenbaun, Andrew S. *Modern Operating Systems*. Prentice Hall Inc. 1992.

Tanenbaun, Andrew S. *Sistemas Operativos Distribuidos*. Prentice Hall Inc. 1996.

Veiga, A. y E. Spinelli. *Microcontroladores en Tiempo Real*. Proccedings del III Congreso Argentino de Ciencias de la computación. 1997.

Veiga, A., E. Spinelli y N. Martínez. *Testing Real-Time Linux*. http://beba.fisica.unlp.edu.ar. 1998.

Veiga, A., E. Spinelli y N. Martínez. *Uso del Linux en Supervisión y Control de Experimentos*. Proccedings del VIII Congreso Latinoamericano de Control Automático. 1998.

Veiga, A., M. Mayosky y N. Martínez. *A hardware/software environment for real time data acquisition and control*. Conference Record de la IEEE Real Time 99 Conference. En evaluación para *IEEE Transactions of Nuclear Science*. 1999.

W3C. *The World Wide Web Consortium Home Page*. http://www.w3.org. 1999.