CPL

UTX/32™ Release 2.1

Real-Time User's Guide

January 1988

GOULD
Electronics

323-005550-000

Limited Rights

Gould Inc., Computer Systems Division
6901 West Sunrise Boulevard
Fort Lauderdale, Florida 33313

History

The *UTX/32 Real-Time User's Guide*, Release 2.1, Publication Order Number 323-005550-000, was printed in January 1988.

This document contains the following pages:

Title page
Copyright page
History page, page iii/iv
Contents, pages v through ix
Figures, page x
Chapter 1, pages 1-1 through 1-4
Chapter 2, pages 2-1 through 2-8
Chapter 3, pages 3-1 through 3-4
Chapter 4, pages 4-1 through 4-6
Chapter 5, pages 5-1 through 5-2
Chapter 6, pages 6-1 through 6-3/6-4
Chapter 7, pages 7-1 through 7-3/7-4
Chapter 8, pages 8-1 through 8-8
Chapter 9, pages 9-1 through 9-2
Chapter 10, pages 10-1 through 10-6
Chapter 11, pages 11-1 through 11-4
Chapter 12, pages 12-1 through 12-5/12-6
Chapter 13, pages 13-1 through 13-3/13-4
Appendix A, pages A-1 through A-40
Appendix B, pages B-1 through B-21/B-22

Contents

Figures

1 Introduction

This introductory chapter provides the following information about this document:

- Its scope and purpose
- A summary of its contents
- Reader prerequisites
- Related documentation
- Typographic conventions

1.1 Scope and Purpose of this Guide

This guide is an introduction to the real-time features of the current release of Gould UTX/32™, which is a real-time-enhanced UNIX® system that can serve as both a development and a target environment for real-time applications. It is intended for users who will be developing, maintaining, or running real-time application programs under this operating system.

1.2 Summary of Contents

This guide is divided into thirteen chapters and two appendixes. The first two chapters acquaint readers with resources available to them and provide a conceptual overview of UTX/32 real-time features. The remaining chapters describe specific functional extensions and real-time performance enhancements to the UTX/32 operating system.

Chapter 1	Provides general information about this document
Chapter 2	Describes the real-time enhancements to standard UTX/32
Chapter 3	Explains various aspects of real-time scheduling
Chapter 4	Describes principles of cyclic scheduling and the cyclic scheduler
Chapter 5	Describes the real-time timer services
Chapter 6	Describes the prepage and lockdown facility
Chapter 7	Describes the System V shared memory interface
Chapter 8	Discusses the direct file system facility
Chapter 9	Describes the high-speed input/output support included in this release

Chapter 10	Explains the connected interrupts functionality
Chapter 11	Describes how to use the **suspend** and **resume** system calls
Chapter 12	Describes memory classes that are used for special purposes
Chapter 13	Describes instruction execution modes, both privileged and unprivileged
Appendix A	Provides general examples of several real-time features
Appendix B	Provides different versions of a model real-time application

1.3 Reader Prerequisites

Readers should already have a working knowledge of UNIX in general and, ideally, of standard UTX/32 features. This document assumes that the audience understands basic UNIX functionality and thus describes only real-time features new to this release.

1.4 Related Documentation

The documentation for the real-time features of UTX/32 includes the following guides and manuals:

UTX/32 Software Release Notes
This document describes the product release and should be read first. Because the real-time environment is new to this release, the release notes are of special interest to real-time users.

UTX/32 Documentation Guide
This document provides an overview of the entire documentation set, as well as a comprehensive UTX/32 glossary.

UTX/32 Real-Time User's Guide
This document.

UTX/32 BSD User's Reference Manual and *UTX/32 BSD Programmer's Reference Manual*
These are collections of manual pages with introductions, consisting of BSD manual pages, manual pages documenting System V functionality that has been ported to the BSD environment, and real-time-specific manual pages, which have section specifiers of the form (*n*RT), such as *dfcreate*(1RT). For system calls and subroutines, the specifier is (*n*RT) for C-callable versions and (*n*RF) for FORTRAN-callable versions. (To indicate both versions in this guide, references will sometimes be made in the form of *dfdelete*(3RT/RF), for example.)

These manual pages are all accessible on line by using the **man** command; see the *man*(1) manual page.

UTX/32 Input/Output Subsystem Guide

This guide describes changes and additions to the UTX/32 input/output subsystem. Of special interest to real-time users are extensions to the I/O interface to support class E devices, the high-speed device (HSD) interface driver and how to customize it, and the direct I/O facility.

1.5 Typographic Conventions

The typographic conventions for this document are described below.

Prompts

The following prompts are used in this document:

`#`	Superuser prompt
`%`	C shell prompt

Nonprinting and control characters

Nonprinting characters obtained by striking special keys are displayed within angle brackets. For example, <DEL> indicates the delete key, <CR> a carriage return.

In this guide, a <CR> is assumed at the end of every command line unless otherwise stated. The <CR> is displayed only if nothing else is entered on the line or if the sequence of keystrokes would otherwise be unclear.

Control characters are represented using the caret notation. For example, ^D indicates <CTRL>-d. In examples, control characters are shown as echoing on the terminal screen. Whether they echo on your terminal depends on its settings; see *stty*(1).

Boldface

Command and utility names, filenames, pathnames, and words from code are printed in boldface.

Example:

The **nroff** command is used to format text.

Exception: When such a term is long and all uppercase, such as PLOCK_FRACTION, it is not printed in boldface.

`Lineprinter` and **`lineprinter bold`**

Displays of code and user sessions are printed in lineprinter font. In displays of interactive user sessions, text typed by the user is printed in lineprinter bold.

Example:

```
% ls
file1      file2      file3
```

Italics

Variable expressions that must be replaced with a value are printed in italics. Square brackets ([ ]) around an italicized variable expression signify that specifying the value is optional.

Example:

% **cd** [*directory*]

Italics are also used to introduce new terms, for titles of documents or manual pages, and occasionally for emphasis.

Examples:

UNIX manual pages are often referred to as *manpages*.

See *mount*(8) for further information.

Ellipses

Vertical or horizontal ellipses (. . .) indicate that information has been omitted.

Example:

```
% rsh fang
   .
   .
   .
% logout
```

Blank pages

Since each major section of the document begins on a right-hand (odd-numbered) page, blank left-hand (even-numbered) pages occasionally precede new sections. You can be assured that such a page is intended to be blank if the preceding page has a double page number, such as 4-5/4-6.

2 The UTX/32 Real-Time Environment

2.1 Real-Time-Enhanced UNIX

This release of UTX/32, a UNIX-based operating system for virtual CONCEPT Product Line (CPL) machines, is the first version of UTX/32 to support real-time features. This chapter presents an overview of these real-time features and of the standard UTX/32 UNIX environments.

A real-time operating system is one that meets the requirements of the real-time software applications that run on it. In general terms, these requirements are for determinism, control, and performance. *Determinism* means predictability of program execution, such that each time a real-time application executes, its computations and input/output are performed in the same relative order. *Control* is the ability to regulate the use of resources. *Performance* for a real-time system means fast response time during both computation and input/output, such that neither is affected by the limitations of the other.

Unfortunately, many operating systems designed to meet these requirements have serious disadvantages:

- They provide poor development environments and a limited range of support software.
- They are proprietary systems. This makes applications less portable and standardization more difficult.
- Standard tools are generally not portable to them, so developers must, in most cases, build their own.

On the other hand, operating systems that offer good development environments and higher levels of standardization tend to be the time-sharing systems, such as UNIX. Designed to maximize throughput, these systems are unsuited for real-time execution. A real-time application can, of course, be developed on a time-sharing system, but it cannot be tested or used there. This makes development more difficult, since testing concurrent with development is not possible.

UTX/32 solves these problems by providing a UNIX development environment as well as an environment that can meet real-time execution requirements.

UTX/32 provides the following:

- Full BSD and System V UNIX functionality, which is development-oriented
- Enhancements that enable the operating system to meet the deterministic execution requirements of a wide range of real-time applications

The following sections describe the UTX/32 time-sharing development environment and the real-time execution environment.

2.2 The Standard UTX/32 Environment

The standard UTX/32 environment offers two UNIX operating system environments, each of which offers benefits to the user.

2.2.1 The BSD and System V Environments

The two major UNIX systems are the Berkeley Standard Distribution (BSD) and AT&T's System V. UTX/32 provides both a BSD environment and a System V environment. Gould enhancements are also included in UTX/32. For a discussion of this split environment, refer to the *UTX/32 Operations Guide* and to the *sv*(1) UTX/32 manual page. For a discussion of how this split environment affects the real-time user, see Section 2.3 of this guide, "The Real-Time Environment."

2.2.2 What UNIX Offers Developers

UTX/32 is fully available to users as a single-user or a multi-user development environment. The following list attempts to convey at least some of the advantages of this environment:

The UNIX standard
UNIX is a well-known standard for operating systems. As a result, a wide variety of third-party software is available for UNIX programmers, much of it free.

Fairness and throughput
UNIX was developed as a time-sharing system for relatively slow machines. The result is an emphasis on fairness and throughput that is beneficial to a multiuser development environment.

Wide selection of tools
UNIX provides a wide selection of basic tools that can be combined. Because the numerous UNIX tools assume a standard, byte-stream format for their input and output, they can be combined in many ways using pipes and I/O redirection.

Programmable shell
UNIX users are not limited to a single command interpreter. The most commonly used command interpreters, the Bourne shell and the C shell, are programmable, and many useful scripts can be written for them.

Choice of editors
A variety of line, view, stream, and more versatile editors, such as Emacs, run under UNIX.

Networking
Networking software available on UNIX includes the Network File System (NFS), support for local area networks, interfaces to high-performance workstations, and access to world-wide networks such as Usenet (the UNIX user community) and the ARPANET.

Information and communication management facilities

Electronic mail and other facilities make it easy to store and exchange information on the system and between systems.

Online documentation

Basic documentation of UNIX utilities, system calls, and libraries is available on the system.

Because UNIX is a commonly used standard operating system, many books describe the UNIX development environment and how to use it. A few of them are:

- Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.
- Henry McGilton and Rachel Morgan. *Introducing the UNIX System.* New York: McGraw Hill, Inc., 1983.
- S. R. Bourne. *The UNIX System.* Reading, Massachusetts: Addison-Wesley, 1983.
- M. G. Sobell. *A Practical Guide to the UNIX System.* Menlo Park, California: Benjamin Cummings, 1984.

Also, many UNIX journals are published.

2.3 The Real-Time Environment

2.3.1 Overview

A real-time operating system must meet certain specific requirements in order to provide the determinism, control, and performance needed to execute real-time applications. Standard UTX/32 includes features that meet some of these real-time requirements. In certain critical areas, however, the real-time environment bypasses or extends standard UTX/32.

Figure 2-1 illustrates, in a very general way, how these new real-time features fit into standard UTX/32. The drawing on the left represents the levels in standard UTX/32, the processing paths between the levels, and examples of services available at each level. The drawing on the right represents the levels in UTX/32 with its real-time enhancements. This drawing also contains arrows that indicate direct paths between the process interface level and the internal services or hardware interface levels. Real-time applications can use these direct paths to bypass the overhead typically encountered in a timesharing environment.

The features of the UTX/32 real-time environment are available within the UTX/32 BSD environment (see Section 2.2, "The Standard UTX/32 Environment.") Since some of these features are standard System V features, UTX/32 makes them available in the BSD environment by including them in the real-time library. For more information on the real-time library, refer to *intro*(2) and *intro*(3) in the *UTX/32 BSD Programmer's Reference Manual* (or type "man 2 intro" and "man 3 intro" on the system).

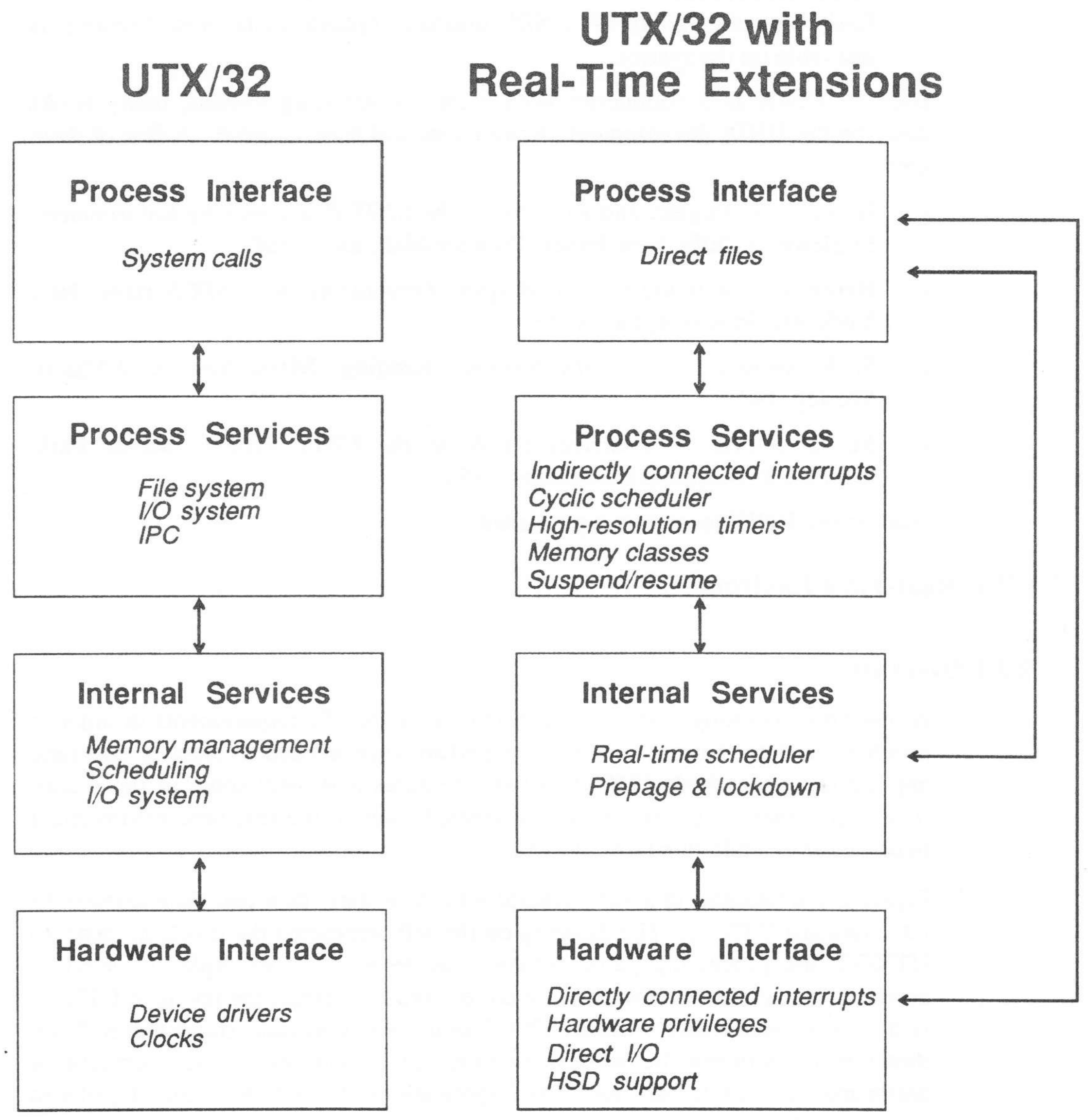

Figure 2-1. Standard UTX/32 and UTX/32 with Real-Time Extensions

This chapter introduces the UTX/32 real-time environment by summarizing, for each real-time-critical area of the operating system:

- The real-time requirements in that area
- The UTX/32 real-time features that meet each requirement
- If a feature is real-time specific, whether it bypasses or extends standard (non-real-time) UTX/32 functionality

Typical real-time application models are then discussed.

Most of these features are described in Chapters 3 through 13 of this guide. The specific system calls and library routines associated with these features are also documented in the *UTX/32 BSD Programmer's Reference Manual.* Special files are in the *UTX/32 Special Files Reference Manual*; commands are in the *UTX/32 User's Reference Manual* or the *UTX/32 System Administrator's Reference Manual*.

2.3.2 Process Scheduling

(1) *Requirement:* Determinism and control in process scheduling.

UTX/32 real-time feature: Real-time, priority-based preemptive scheduling. See Chapter 3, "Real-Time Scheduling," for more information.

Bypass or extension? Bypasses standard UTX/32 time-shared scheduling.

(2) *Requirement:* Frame-based, repetitive, process execution.

UTX/32 real-time feature: Cyclic scheduling. See Chapter 4, "Cyclic Scheduling," for more information.

Bypass or extension? Extends standard UTX/32.

(3) *Requirement:* High-speed context switching.

UTX/32 real-time feature: Suspend and resume primitives. See Chapter 11, "Suspend and Resume," for more information.

Bypass or extension? Extends standard UTX/32.

2.3.3 Timer Services

Requirement: High-resolution event scheduling and execution-time measurement.

UTX/32 real-time feature: Enhanced timer services. These provide interval and time-of-day measurements from a high-resolution clock. See Chapter 5, "Timer Services," for more information.

Bypass or extension? Extends standard UTX/32 timer services.

2.3.4 Paging and Swapping

Requirement: Control over process paging and swapping.

UTX/32 real-time feature: Prepage and lockdown. This facility allows the user to make all or part of a process resident in memory. See Chapter 6, "Prepage and Lockdown," for more information. This feature is standard in the System V environment and provided in the (BSD) real-time library.

2.3.5 Interprocess Communication

Requirement: Deterministic interprocess communication.

UTX/32 real-time features: Shared memory, semaphores, messages, and signals. See Chapter 7, "Shared Memory," for more information about shared memory. Signals are standard BSD. Shared memory, sempahores, and messages are standard System V but are also provided in the (BSD) real-time library.

2.3.6 The File System

Requirement: A file system that supports contiguous files, deterministic transfer times, and asynchronous I/O capability.

UTX/32 real-time feature: Direct files. The direct file system provides predictable disk I/O for files. See Chapter 8, "Direct File System," for more information.

Bypass or extension? Bypasses the UTX/32 file system.

2.3.7 Input/Output

(1) *Requirement:* Support for common real-time devices.

UTX/32 real-time feature: Support for class E devices. UTX/32 includes a generic, user-extendible high-speed device (HSD) driver. See Chapter 9, "High-Speed Input/Output Support," for more information.

Bypass or extension? Extends the UTX/32 I/O interface.

(2) *Requirement:* Direct access to I/O devices for processes, including asynchronous I/O capability.

UTX/32 real-time feature: Direct I/O. See Chapter 8, "Direct File System," for more details.

Bypass or extension? Bypasses standard UTX/32 I/O.

2.3.8 Hardware Control

(1) *Requirement:* Access to special-purpose memory.

UTX/32 feature: Memory classes. See Chapter 12, "Memory Classes."

Standard or new? New.

Bypass or extension? Extends standard UTX/32 functionality.

(2) *Requirement:* Minimal interrupt response time.

UTX/32 feature: Directly connected interrupts. See Chapter 10, "Connected Interrupts."

Standard or new? New.

Bypass or extension? Extends standard UTX/32 functionality.

(3) *Requirement:* Execution of privileged instructions by user processes.

UTX/32 feature: Privileged instruction execution mode. See Chapter 13, "Instruction Execution Modes."

Standard or new? New.

Bypass or extension? Extends standard UTX/32 functionality.

2.3.9 Application Models

UTX/32 supports the execution of real-time applications on several models:

- Clock-driven control processes
- Clock-driven independent processes
- Cyclically scheduled processes (frame-based execution)
- Event-driven processes

The high-resolution clock and cyclic scheduler support the clock-driven and cyclically scheduled models, respectively, which are highly sensitive to context-switching time. Direct I/O and connected interrupts support the event-driven model, where I/O response time is critical.

Appendix B provides two versions of a simple real-time application written in C. The first version runs off the high-resolution clock, the second uses cyclic scheduling.

2.4 Notes to FORTRAN Programmers

The FORTRAN programmer should be aware of the following:

- Using the UTX/32 F77 compiler is not recommended. The Gould Common FORTRAN compiler is strongly recommended instead, because it provides a syntax that is common to MPX-32™, UTX/32 on CPL machines, and UTX/32 on NPL machines.
- This release of the operating system includes support for global commons and datapools, but this functionality is only accessible by using Gould Common FORTRAN, a product that is available separately.
- It is often possible to use C subroutines to advantage in places where you may be accustomed to using assembly language code. FORTRAN programs can call C subroutines that define system calls. See Section A.6, "Direct I/O," for an example.

2.5 Special Considerations

The following are warnings to heed when using the real-time features:

- Do not log in and work at the console. UTX/32 console I/O is unavoidably slow and resource consumptive in the real-time environment.
- When compiling or linking any real-time program, you must load the real-time library using the **–lrt** option to the compiler or linker.

```
% cc -O -o control control.c -lrt
```

- If the program is in FORTRAN, you must also load the the real-time FORTRAN library using **–lrtf**. Because FORTRAN library functions may call functions from the C real-time library, **–lrt** must follow **–lrtf**.

```
% fort -O -o diotest.f -lrtf -lrt
```

3 Real-Time Scheduling

3.1 Aspects of Real-Time Scheduling

UTX/32 real-time scheduling has two components:

- Preemption based only on real-time priority
- Processor targeting

These topics will be addressed separately.

3.2 Real-Time Priority

3.2.1 Overview

The UTX/32 timesharing scheduling algorithm continuously adjusts the priority of each process based on the recent resource use of the process. The adjustment is intended to give each process a fair share of the system's resources. Since real-time applications may need to give very unbalanced shares of resources to specific processes, UTX/32 includes scheduling options allowing an application to assign nonadjustable real-time priorities to processes. This results in stricter control over process ordering.

Real-time processes have priority values in the range 0-63, with 0 being the highest priority. The process with the highest priority executes continuously until it voluntarily relinquishes the processor to await an event, which is known as *blocking*, or until a process with a higher real-time priority becomes runnable.

Real-time priorities are logical process priorities; they do not affect interrupt servicing. Real-time priorities do not change except as the result of a **setrealpriority** or **unixscheduling** system call. See *realpriority*(2RT/RF).

3.2.2 User Interface

UTX/32 provides system calls to switch a process between real-time and standard UTX/32 scheduling and to determine the real-time priority of a process.

1. To make a process run at a real-time priority, use the following code:

```
oldprio = setrealpriority(pid, newprio)
```

   This sets **oldprio** to the old real-time priority of the process with the process identification number of **pid**, or to NONRT if the process was formerly a standard UTX/32 (non-real-time) process (NONRT is defined in **/usr/include/sys/types.h**). It assigns **newprio** to be the current real-time priority of the process.

2. To find out the real-time priority of a process, use the following code:

   ```
   realprio = getrealpriority(pid)
   ```

   This sets **realprio** to the real-time priority. If the process is a standard UNIX process, **realprio** is set to −1 and **errno** is set to EINVAL.

3. To revert to standard UTX/32 scheduling, use the following code:

   ```
   error = unixscheduling(pid)
   ```

   This assignment works without comment, unless the process was already a standard UTX/32 process, in which case **error** is set to −1 and **errno** is set to EINVAL.

3.3 Processor Targeting

3.3.1 Overview

Virtual CPL processor architecture is asymmetrical; both a CPU and an IPU are supported. To improve total throughput, the UTX/32 scheduler reviews how processes use the processor services and restricts those processes that use CPU-only services to running on the CPU. Such a process may have to wait for the CPU to become free even though the IPU is idle. Since real-time applications often do very explicit load balancing of their own, UTX/32 includes options allowing a process to indicate which processor it should be run on; those options override the targeting done automatically by the time-sharing scheduler.

Processor targeting permits the user to indicate that processes should run on particular processors whenever possible. On virtual CPL machines, all processes *can* run on the CPU, but processes doing I/O or system calls *must* run on the CPU and cannot run on the IPU. Processor targeting lets the user specify that important computations should run only on the IPU, where they are not subject to external interrupts. Such a process will run on the CPU during execution of system calls but will be moved back to the IPU immediately, instead of moving back slowly under UTX/32's adaptive load balancing algorithm.

If only invalid processors are specified when targeting processes, **settargetcpumask** returns an error, setting **errno** to EINVAL. If valid as well as invalid processors are specified, the valid processors are enabled and no error is returned. In particular, an error will occur if a task is targeted only to the IPU on a CPU-only CPL system. Applications using processor targeting should use **getactivecpumask** to check the system configuration before trying to assign tasks to particular processors. This consideration is especially important when developing code on a CPU-only system if that code will eventually be run on a CPU/IPU system.

3.3.2 User Interface

UTX/32 provides system calls to target a process to a set of processors, to determine the targeting of a process, and to determine what processors are available. CPU targeting defines symbolic names for processors. It also defines **cpumask_t**, an integer type to which logical operations can be applied. **cpunumber_t** is the actual number of a processor. Numbers are converted to masks by P_CPUMASK().

- To target a process to a set of processors, use the following code:

  ```
  mask = settargetcpumask(pid, newmask)
  ```

  This sets **mask** to the old CPU targeting of the process and assigns the process to the processors specified by **newmask**. If any invalid processors are specified by **newmask**, **errno** is set to EINVAL.

- To find out what processors a process is targeted to, use the following code:

  ```
  mask = gettargetcpumask(pid)
  ```

  This sets **mask** to the current processor targeting of the process.

- To find out what processors are on the system, use the following code:

  ```
  mask = getactivecpumask()
  ```

  This sets **mask** to the set of processors available as targets.

For examples of real-time scheduling, refer to Section A.1, "Real-Time Scheduling," and Section B.4, "Cyclic Scheduling Model."

3.4 Special Considerations

3.4.1 Interactive Control of Real-Time Processes

Running real-time processes from a shell session may lead to confusing problems since real-time processes take priority over the normal shell. In order to keep interactive control of real-time processes in development, the priority of the shell may be boosted above that of the processes being tested. In a networked environment, it may also be necessary to boost the priority of processes handling user input, like **telnetd**. See *csh*(1), *sh*(1), and *nice*(1) for more information about boosting the priority of shells and processes.

3.4.2 FIFO Processing in a Multiprocessor System

Even with real-time scheduling, it is difficult to guarantee first-in/first-out (FIFO) processing of tasks on a multiprocessor such as the dual CPU/IPU configuration. The real-time implementation ensures that event handling occurs in a FIFO order, but not total execution. Differing interrupt loads and the requirement that system calls be executed on the CPU may result in different effective execution speeds on different processors. A process that started executing on the IPU may be "passed" by a process at the same priority that starts later but runs on the CPU, or vice versa. If the overall load is well understood, processor targeting may help in serializing the execution of a particular set of processes.

3.4.3 Shared Memory

The CPU/IPU implementation for virtual CPL machines does not guarantee cache consistency between the two processors under all circumstances. Although the inconsistency is rare, processes that share memory across processors must implement their own synchronization and may be sensitive to the inconsistency. To ensure that all users of a given segment will be in the same cache, processes using shared memory are usually marked automatically so that they will not execute on the IPU. Only non-targeted processes can be thus marked.

3.4.4 Targeting Nonexistent Processors

As previously mentioned, if all specified processors are invalid, **settargetcpumask** returns an error. For more information, see Section 3.3.1, "Overview."

3.4.5 Non-Real-Time Processes

For non-real-time UTX/32 processing, processor targeting is less strict, involving biasing rather than targeting. An idle CPU may pick up IPU-targeted tasks.

4 Cyclic Scheduling

4.1 The Cyclic Scheduler

The cyclic scheduler provides an efficient, reliable, and easy-to-use way to schedule processes periodically, according to the specifications supplied by each process. One or more processes may be scheduled for the same time. These processes subsequently run according to their real-time priority. The cyclic scheduler can also synchronize processes so that they maintain a constant phase relationship.

The cyclic scheduler divides each second into frames, based upon the line frequency. For example, on a 60 Hz system, each frame will be 16.667 milliseconds long. Every cyclically scheduled process defines two variables in setting up its cycle and its scheduling: **cycle_length** and **cycle**.

4.2 Cycles and Frames

cycle_length is the number of frames in the process's cycle. The number of frames in a cycle is usually, but not always, equal to the line frequency or to a fractional multiple of the line frequency. On a 60 Hz system, typical cycles would have lengths of 60, 30, 20, 15, or even 10 frames. In the current implementation, the number of frames cannot be greater than the line frequency, but since line frequency standards vary, the cyclic scheduler can handle any line frequency up to 256 Hz. Note, however, that the length of each individual frame is determined by the line frequency, not the number of frames per cycle. Thus, in the 60 Hz system example, the frame duration is 16.667 milliseconds, no matter what the **cycle_length** might be.

cycle is an array of dimensions 1 by **cycle_length** that defines the process-specific cycle. The maximum number of frames is equal to the line frequency. Therefore, for any cycle with a **cycle_length** less than the line frequency, the cyclic scheduler will ignore frames from **cycle_length + 1** to the line frequency. A cyclically scheduled process is scheduled to perform its work only in certain frames, known as *set frames*, during its process-specified cycle. The process defines those set frames in the array variable **cycle**. For example, on a 60 Hz system, a typical process might have a **cycle_length** of 20 frames and a **cycle** which sets every other frame. The 1st through the 20th frames define the process's cycle and can be set to run or left unset. The 21st through the 60th frames will be ignored by the cyclic scheduler.

Set frames can come singly or in groups, and any pattern of set frames is permitted—every second frame, every fifth frame, two set frames followed by an open frame—whatever is desired. A cyclically scheduled process is expected to finish its work before the next set frame; that is, each set frame should represent one complete execution of a cyclically scheduled process. The process must suspend itself when it is done with the task by invoking the **cycsuspend** system call. The process then waits for its next set frame. A process that continues running past a set frame will miss that frame and be scheduled for the next set frame. Figures 4-1 and 4-2 demonstrate these basic concepts of cyclic scheduling.

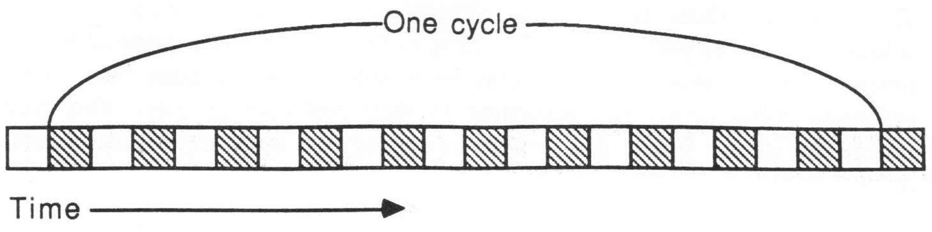

Figure 4-1. A Cycle of 20 Frames

Figure 4-1 illustrates an isolated cycle with a length of 20 frames. (The ignored frames are not pictured.) Thus, for a 60 Hz system, each frame lasts 16.667 milliseconds. Every odd frame in the cycle is set, indicated by the shading. This cycle and framing would be appropriate for a process that needs to run 10 times each cycle and requires less than 16.667 milliseconds to complete its work each time it runs. Note that the process can actually run for 33.333 milliseconds, because it has an unset frame separating each set frame.

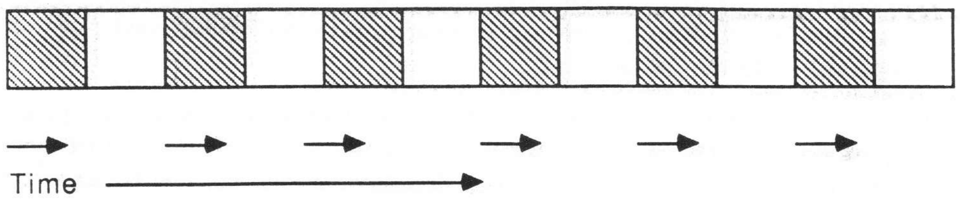

Figure 4-2. A Well-scheduled Cyclic Process

Figure 4-2 illustrates a cycle segment showing a well-scheduled cyclic process. The arrows beneath the frames represent process execution time. Notice that the process initiates at the beginning of each set frame, and finishes and suspends before the end of the set frame.

For examples of cyclic scheduling, refer to Section A.2, "Cyclic Scheduling," and Appendix B, "Model Real-Time Applications."

4.3 User Interface

4.3.1 Scheduling

Processes request cyclic scheduling by calling the **cycsetdata** library routine and passing three parameters: the cycle, the cycle's length, and a variable called **frames_lookback**, which helps cyclic processes stay on schedule. **cycgetrate** returns the hardware-dependent line frequency, which can be used in setting the cycle. See *cycsetdata*(3RT/RF) and *cycgetrate*(3RT/RF).

Once the process has called **cycsetdata**, cyclic scheduling begins at the next clock tick, with frame number zero. The process will be scheduled for its first set frame after the next clock tick. If more than one process is scheduled to begin at the same time, real-time priority scheduling takes place. The process with the highest priority will run first. If two processes having the same real-time priority are scheduled for the same time, the order of their execution is undefined.

4.3.2 Synchronization

Cyclically scheduled processes often need to run synchronously. **cycsync** resets the frame counters of all cyclically scheduled processes to zero. The processes will be resynchronized at frame zero of the next clock tick. Processes do not need to be **cycsuspend**ed at the time of a **cycsync**; however, any processing they do will not be synchronized with other processes until a **cycsuspend** is issued. See *cycsync*(3RT/RF) and *cycsuspend*(2RT/RF).

4.3.3 Delayed Execution

If a process must coordinate activities with several other processes, then it may choose to delay execution until all of those processes have completed their startup procedures. Processes request this delay by setting the **wait_for_sync** flag in the **cycsetdata** routine. One of the processes will then call the **cycsync** library routine. Cyclic scheduling will commence at the next clock tick for all cyclically scheduled processes. All frame counters are reset to zero. Those processes that are scheduled to run in frame zero will be scheduled; the rest will remain suspended until their first set frame.

4.3.4 Signal Handling

A cyclically scheduled process may receive signals while suspended or while waiting for cyclic synchronization. If this happens, the process will field the signal, execute the signal handle if one has been specified, return from the system call with an error, and set **errno** to EINTR. If they do not want to service interrupts, processes need to guard against the possibility by blocking or ignoring signals.

Signals may be blocked or ignored during the system call. Ignored signals will be thrown away, but blocked signals will remain.

4.3.5 Lockdown

It is strongly suggested that cyclically scheduled processes be locked into memory using the **plock** system call, especially if they use real-time priorities. This will avoid possible deadlock, since real-time priority processes run before standard UTX/32 priority processes (including the swapper). See *plock*(2RT/RF) and Chapter 6, "Prepage and Lockdown," of this document.

4.3.6 Rescheduling and Termination

Processes may change their cyclic scheduling parameters at any time by calling **cycsetdata** again. The new parameters will take effect with the next clock tick. If this is done, a **cycsync** must be issued to resynchronize all cyclically scheduled processes. In C, cyclic scheduling may be terminated by calling **cycsetdata** with a null pointer for the first parameter. FORTRAN users should supply a third parameter of zero to terminate cyclic scheduling.

Processes can get their current cyclic scheduling parameters by calling the **cycgetdata** library routine; see *cycgetdata*(3RT/RF).

4.4 Special Considerations

Although cyclically scheduled processes are expected to suspend before their next set frame, this does not always happen. When a process suspends in a set frame other than one in which it started or resumed, it is said to have overrun. The cyclic scheduler does not schedule processes for frames into which an overrunning process spills, and these become *missed frames*. Several set frames may be missed at one time if a process severely overruns a set frame. Missing frames can be hazardous to certain cyclic processes. The cyclic scheduler has services that can help cyclic processes adjust to minor overruns. Figure 4-3 illustrates a poorly scheduled cyclic process.

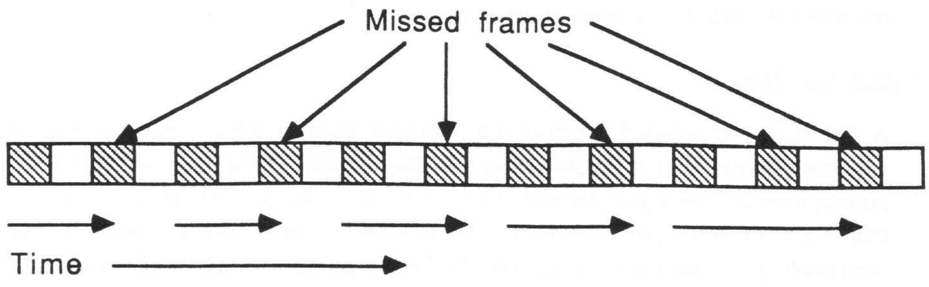

Figure 4-3. A Poorly Scheduled Cyclic Process

The process consistently overruns, misses frames, and will not perform its work as frequently as desired. Note that the amount of work can build up, causing the process to miss more and more frames as time passes.

4.4.1 Setting frames_lookback

Some processes will not be harmed if they miss a frame. On the other hand, processes sometimes overrun a frame through no fault of their own, and it may be important that they not miss that frame. Cyclically scheduled processes can request help adjusting to missed frames, depending on their needs. This is done by an appropriate setting of **frames_lookback**. This variable determines scheduling when a cyclic process overruns, as follows:

frames_lookback=0
The cyclic scheduler will assume that the current frame has been missed, whether or not it is set, and will schedule the process for the next set frame. This is the usual setting for a well-scheduled cyclic process that will never miss a frame, or will miss one so rarely that such a mishap does not matter.

frames_lookback>0
The cyclic scheduler will look at (**frames_lookback – 1**) frames prior to the current frame. If any of those frames is set, then the process is scheduled immediately, that is, it returns immediately from a **cycsuspend** system call. This feature is intended to be used by well-scheduled cyclic processes that occasionally miss a frame but still want to run, even though they run a bit late; that is, the processes are expected to catch up by the next set frame. Figure 4-4 illustrates such a process.

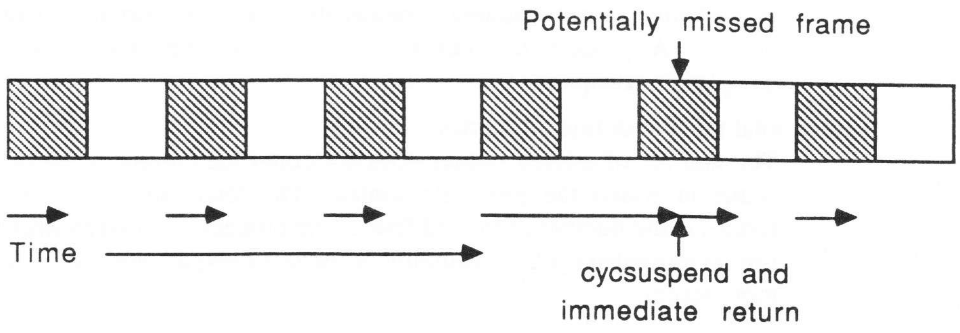

Figure 4-4. A Cyclic Process Using **frames_lookback**

This process occasionally misses a frame and would benefit from a judicious setting of **frames_lookback**. Given that this process usually runs and completes within a single set frame, setting **frames_lookback** to 1 or 2 would force the process to run at its usual time. The process would then be caught up.

Large Values of frames_lookback

Setting **frames_lookback** to (**cycle_length – 1**) or larger effectively turns off cyclic scheduling, since the process will resume after every suspension. This is rarely desired of cyclic processes.

Caution is needed in using the **frames_lookback** feature. It should not be used to fix a poorly scheduled set of processes. If a process is continually missing frames, then the amount of work it does must be cut, or the cycle must be rearranged. Using large values for **frames_lookback** only delays this inevitable rearrangement.

4.4.2 Monitoring Missed Frames

Processes can monitor the state of their missed frames in three ways:

cycsuspend

Returns the current frame number so the process knows what frame it is in when it begins to run. Frame numbers begin with zero, not one; therefore, the first frame in the cycle is frame 0, not frame 1.

cycgetframe

Returns the current frame at any time.

cycgetstatus

Returns the current status of any cyclically scheduled process. The status information contains:

The number of missed frames since the process last resumed

The number of missed frames is the number of frames the process has missed since it last resumed, whether the process is currently running or not. A process has not missed a frame until the frame has completely transpired.

The total number of missed frames

The number of missed frames does not count the current frame or the frame in which the process resumed. The total number of missed frames is the number of missed frames the process has missed since the time it requested cyclic scheduling or the last **cycsync**, whichever is the more recent.

The current state of the process

The state of a cyclically scheduled process is either running or suspended. See *cycsuspend*(3RT/RF), *cycgetframe*(3RT/RF), *cycgetstatus*(3RT/RF), and *cycsync*(3RT/RF).

Status information is invalid in the following two cases:

1. The process is initializing. In this case, **cycgetstatus** will return an error.
2. The process has not suspended since the last **cycsync**. In this case, the process has not had a chance to synchronize itself with other processes. When the process next suspends, it will notice that a **cycsync** has occurred; it will synchronize, and status information will again become valid.

5 Timer Services

5.1 Overview

UTX/32 real-time timer services allow the user to create alarms and to measure intervals with greater resolution than has formerly been available on UTX/32. These real-time services are based on a high-resolution clock distinct from the UTX/32 system clock.

5.2 User Interface

UTX/32 timer services use the **gethscvalue** and **sethsctimer** system calls; see *gethscvalue*(2RT/RF) and *sethsctimer*(2RT/RF). These calls respectively get the current value of the timer and set an alarm for the process. The user interface is almost exactly the same as that of the standard UTX/32 **getitimer** and **setitimer** calls. **gethscvalue** returns the number of seconds and nanoseconds since the last system boot. **sethsctimer** sets the high-resolution clock. Each process may have only one outstanding alarm. Setting an alarm cancels any existing alarm. When the timer expires, the process receives a SIGALRM signal. **sethsctimer** can be used to set three kinds of alarms. An *absolute* alarm is sent at a time measured from system boot time. A *relative* alarm is sent at a time measured from when it was set. A relative alarm can also be *repeating*, in which case a SIGALRM signal is sent periodically. The three types of alarms can be requested by giving the appropriate mode value to **sethsctimer**. An additional mode allows the user to cancel an outstanding alarm.

The mode values are defined in **<sys/rt_time.h>**. This file includes the following defines:

```
#define HSC_MAX_TICKS  2000000000  /* 20 min at 600 ns per tick */
#define HSC_RATE       600         /* nanosec per hsc tick */
```

It is recommended that the timer frequency be set to 600 nanoseconds per tick. This will yield a maximum high speed clock interval of about 1288.24 seconds. In addition to modifying the count rate, the interval timer hardware on the MFP or IOP must be jumpered to count at the same rate. Consult your hardware service representative to change the interval timer count rate.

For examples of using the timer services, refer to Section A.3, "Timer Services," Section A.4, "Shared Memory," Section B.2, "High-Resolution Interval Timers Model," and Section B.3, "High-Resolution Repeating Timers Model."

5.3 Special Considerations

- It is possible for an alarm to be delivered while a cancel request for it is in progress.
- Use of the repeating timer feature available through **sethsctimer** should be reserved for applications requiring its high-resolution accuracy. For most cases in which periodic scheduling is required, use of the UTX/32 cyclic scheduler is recommended for its high reliability.
- The high-speed timer services, as distributed, use the interval timer on the IOP. They expect the interval timer to be jumpered to 600 ns. It does not come distributed that way. If the user notices that only a few million "nanoseconds" are passing every second, this is the problem.

6 Prepage and Lockdown

6.1 Description

This facility permits a process to fix its virtual memory pages in physical memory, resulting in (1) the avoidance of nondeterministic delays as paging occurs, and (2) the ability to do I/O directly in user memory. For more information, refer to *plock*(2RT/RF). For more information on direct I/O, see Section 9.3, "The Direct I/O Facility."

6.2 User Interface

Prepage and lockdown are implemented by the **plock** system call. With this call, you can do the following:

- Lock an entire process with **plock**(PROCLOCK)
- Lock the text segment of a process with **plock**(TXTLOCK)
- Lock the data and stack segments of a process with **plock**(DATLOCK)
- Remove all locks with **plock**(UNLOCK)

The following code is a simple instance of using the **plock** system call in C.

```
main()
{
     if( plock(PROCLOCK) != 0 )
     {
            printf("Can't plock\n");
            exit(1);
     }
     /* Use direct I/O */
}
     /* plock implicitly released on exit from process */
```

For further examples of **plock** in use, refer to Section A.2, "Cyclic Scheduling," Section A.5, "Direct File System," Section A.6, "Direct I/O," and Appendix B, "Model Real-Time Applications."

6.3 Special Considerations

6.3.1 Managing Physical Memory

Because physical memory is a limited resource, **plock**ed processes cannot increase their stacks or data areas without limit. In the current UTX/32 implementation, the variable **max_total_plocks** in the kernel sets an upper limit on the amount of **plock**ed memory. By default, it is set at boot time to PLOCK_FRACTION (75%) of the physical memory available to the user. The actual amount of locked down memory is contained in the variable **total_plocks**. All of these are defined in **usr/include/sys/cmap.h**. Their units are *cmaps*, which are individual 8Kb pages of physical memory. Because of this granularity, the limit on the amount of locked down memory should be considered advisory only. The limit may be exceeded if multiple processes perform **plock**s simultaneously.

If you wish to change the limit for **plock**ed memory, proceed as follows:

1. In a source code system, change PLOCK_FRACTION and reboot the system.
2. In a binary system, patch the variable **max_total_plocks** into the running kernel using **adb**; see *adb*(1). This is shown in the following example. The user first invokes **adb**, including the **–k** option to use kernel memory and the **–w** option to enable write mode. Of course, someone doing this must have permission to write **/dev/kmem**. Note that **adb** does not return a prompt, and that it spaces out its responses. The user next examines the current value of **max_total_plocks** (24a in this example) and is returned a value. The user writes the new value for **max_total_plocks** and finally examines the variable to ensure that the new value is in place.

```
# adb -k -w /unix /dev/kmem
sbr af7c8 slr afbc8
p0br 0 p0lr 0 p1br 0 p1lr 0
max_total_plocks/X
_max_total_plocks:      24a
max_total_plocks/W7fffffff
_max_total_plocks:      24a      =       7fffffff
max_total_plocks/X
_max_total_plocks:      7fffffff
```

Note that, in changing the limit, care must be taken to avoid situations in which deadlock can occur involving other facilities that require locked-down memory.

6.3.2 Unlocking and Exiting

Memory must be locked down if direct I/O is being done into it and must remain locked until the I/O is completed. **plock** provides a facility inside the kernel by which the direct I/O driver or a directly connected interrupt can register a claim against locked-down memory. Attempts to unlock memory while this claim is outstanding are prevented, and facilities using locked-down memory must call cleanup routines before **plock** can exit.

6.3.3 Waiting to Load

plock waits for virtual memory pages to be paged in when locking. This means that **plock** may take a long time to return if very little physical memory is free.

6.3.4 Shared Text

In the current implementation, shared text that is **plock**ed remains locked down, even after the process that locked it exits, until all processes using that shared text have exited.

6.3.5 Shared Memory

plock is intended to lock only the principal data segment of a process; it does not necessarily lock down shared memory segments. In the current implementation, this distinction does not matter, because shared memory is always locked down, but this locking should not be relied on in the future. It is possible to lock down shared memory explicitly by using the **shmctl** system call. See *shmctl*(2RT) for more information.

7 Shared Memory

7.1 Description

The current release of UTX/32 includes an AT&T System V shared memory interface that allows two or more processes to share physical memory segments for data storage. The interface is intended to be used with semaphores or a simple message protocol to provide a high bandwidth interprocess communication (IPC) facility. In light of this, the System V shared memory interface uses standard IPC control and operation primitives and values in common with System V messages and semaphores.

Shared memory segments are dynamically created. Each segment is assignèd a unique identifier by its creator and, like UTX/32 files, a set of access permissions. Processes wishing to attach to the segment must use the same identifier and have appropriate permissions.

Shared memory segments need not occupy the same virtual address space in each sharing process. In normal usage, assignment of shared memory virtual addresses is left to the operating system. A process can request that a segment be attached at a specified address as long as that space is not already in the process image.

NOTE: The FORTRAN interface to shared memory is through global commons and datapool facilities provided by the Gould Common FORTRAN compiler. The System V shared memory facilities will provide the underlying mechanism for the implementation of these language facilities.

7.2 User Interface

Four functions are provided for manipulating shared memory segments:

shmget
: Gets a shared memory segment; see *shmget*(2RT).

shmctl
: Provides a variety of specifiable shared memory control operations; see *shmctl*(2RT). These include

- Setting permissions on a shared memory segment
- Placing the contents of a shared data structure into a buffer
- Locking a shared memory segment into memory and unlocking it
- Removing a shared memory identifier from the system and destroying the shared memory segment and data structure associated with that identifier

shmat

Attaches the shared memory segment associated with the shared memory identifier to the data segment of the calling process. Depending on the address specifier and other factors, the segment may be attached at the first available address selected by the system or at a specified address, and it may be read-only or read-write. See *shmop*(2RT).

shmdt

Detaches the shared memory segment located at a specified address from the calling process's data segment; see *shmop*(2RT).

The following header files should be included in any program which uses any of the C shared memory interface functions: **<sys/types.h>**, **<sys/ipc.h>**, and **<sys/shm.h>**.

For an example of using shared memory, see Section A.4, "Shared Memory."

7.3 Special Considerations

7.3.1 Configuration

Several limits on the number and size of shared memory segments exist as configuration parameters. See Chapter 7, "Reconfiguring the System," in the *UTX/32 Operations Guide*.

7.3.2 Synchronization

On PN6080 and PN9080 (CPU/IPU) multiprocessor systems, processes must use some method of explicit synchronization to ensure consistency of shared data.

7.3.3 Virtual Address Space Implications

A shared memory segment can be attached only at a virtual address greater than the highest valid address in the current process image.

The address of the first (lowest) shared memory segment attached to a process places an upper bound on the dynamic data region that can be allocated to a process.

7.3.4 Paging and Swapping

If a real-time process requires deterministic access to a shared memory segment, it is recommended that the **plock** system call prepage and lock down the shared segment. In the current implementation, shared memory is locked down, but this locking should not be relied on in the future; see *plock*(2RT/RF).

7.3.5 Shared Text

The System V shared memory interface does not support text sharing. Shared text is implemented in UTX/32 as a characteristic of the process specified at link time. (Text sharing is the default mode of **ld.**)

8 Direct File System

8.1 Description

The UTX/32 direct file system allows processes to perform disk I/O in a fast, efficient manner. It is a collection of library routines that allow the user to manipulate a disk volume using UTX/32 direct I/O capabilities. With this facility, you can

- Define contiguous file structures
- Create preallocated files
- Perform priority I/O requests with asynchronous notification
- Achieve fast and predictable I/O times
- Perform I/O without buffering
- Access a disk volume from multiple processes

These features are useful for real-time processing and applications that require high-speed, predictable disk data rates. Standard UTX/32 I/O does not support contiguous files, static priority I/O, or asynchronous I/O completion notification.

The structure of the direct file system makes single-revolution opens, reads, and writes possible for preexisting files. This implies that files can be created so that the directory entry can be located on the first disk access, and read/writes can be started without reading any disk information other than the data to be transferred. Only situations resulting from disk contention with other processes can slow down the direct file system.

The direct file system was implemented using the direct I/O interface, but if necessary, the standard UTX/32 raw I/O interface can be specified when mounting a direct file system. Note, however, that when using the UTX/32 I/O interface, asynchronous capabilities are lost and access is not guaranteed to be synchronous. See *dio*(7RT), *ioi*(7), and the *UTX/32 Input/Output Subsystem Guide* for further information.

The user commands for the direct file system are: **dfcreate**, **dfdelete**, **dfextend**, **dfls**, **dfread**, **dfrename**, and **dfwrite**. The system administrator commands are **dfmkfs**, **dffsck**, **dfmount**, and **dfumount**. These commands, and the direct file system library routines and system calls, are documented in the *UTX/32 BSD User's Reference Manual*, the *UTX/32 BSD Programmer's Reference Manual*, and the *UTX/32 System Administrator's Reference Manual*. All relevant manpages begin with *df*.

This is useful if the user wishes to have both a direct file system and a UNIX file system on the same volume and use them simultaneously. The direct file system demands exclusive access to a volume.

8.2 User Interface

8.2.1 File System Structure

Each direct file system resides on its own volume and must be mounted, using the **dfmount** utility, on a disk device. The direct file system is not accessible by the UTX/32 file system, but it is mounted and dismounted similarly.

Files are referenced by pathnames of the form **/volume/name**, where **volume** is the name by which the volume is mounted and **name** is the name of the file on the volume; see *dfmount*(8RT). Neither the volume name nor the filename can be longer than 31 characters.

The **dfmkfs** utility formats the direct file disk; see *dfmkfs*(8RT). It formats an empty directory on the disk and forms a free list of all the disk tracks that have not been marked "bad" by the verification diagnostic. Each volume contains one directory (i.e., a flat file system) that contains entries for all files on the volume. To give better hashing performance, twice the number of entries requested are created, but only the requested number of entries can be accessed.

Here is how the disk is organized:

- The first block contains a volume descriptor describing the disk geometry and the volume formatting.
- The free list bit map occupies the next set of sequential blocks. Each bit in the free list corresponds to a track on the disk. All tracks on the disk are represented, including the free list itself. A set bit denotes that the track is allocated. The number of blocks in the free list depends on the size of the disk.
- The disk directory occupies the next set of sequential blocks. It extends to a track boundary.
- All disk blocks following the directory are data blocks.

8.2.2 File Characteristics

Direct files have these characteristics:

Extents

Direct files are composed of groups of contiguous disk blocks called *extents*. Files may be composed of 1 to 16 extents, with each extent a multiple of disk tracks. The user controls the size of each extent and the distance between the extents (see "Gaps," below) at creation time. By creating a file with an extent size the same as the file size, you can create an entirely contiguous file.

Gaps

There will often be cases where bad blocks partition the disk so that large, contiguous files cannot be created. For this reason, you can specify a maximum gap size when you create a file. Maximum gap size is defined to be the number of bytes that can separate two adjacent extents in a file.

Since the minimum disk allocation unit is one track, the minimum allowable gap size is also one track. This implies that the start of the *n*th extent is within some user-defined number of tracks from the last track of the *n*-1th extent. Skipping a track or cylinder may be acceptable, but seeking to the front of the disk may be unacceptable in certain applications.

Disk geometry

Disk geometry (bytes per block, blocks per track, etc.) can vary from disk to disk. The direct file system can support any size the disk driver for the UTX/32 disk processor can initialize, as long as the block size is greater than the size of a single directory entry. Recommended sizes for transfers are multiples of the disk block size.

8.2.3 dffstab

The file **/etc/dffstab** describes the file systems used by the direct file system. It is read by **dfmount** to determine device address. This file must be created before a direct file system can be mounted. See *dffstab*(4RT).

8.2.4 File System Attributes

The direct file system, using the direct I/O interface, has the following attributes:

No buffering

All I/O is done directly into the caller's address space. Data is not buffered or blocked.

Priority I/O queuing by process

Each process has its I/O queued at the process' real-time execution priority. Within a process, I/O requests to a given device are queued in FIFO order.

Asynchronous I/O completion notification

When files are opened, the mechanism for I/O completion notification is specified. I/O can be performed as *wait*, where the process is blocked from execution until the I/O has completed, or *no-wait*, where the process continues executing after the I/O request has been queued. In either mode, the user supplies a pointer to a set of three event counters that are to be incremented, respectively, as follows:

- When an I/O request is received for that file

- When an I/O request has completed for that file
- When an I/O error occurs on that file

The user can poll the completion event counter to determine when I/O has completed. Optionally in either mode, the user can supply the address of a procedure to be called and executed when the I/O completes. The user-supplied procedure is passed two parameters that contain the file descriptor and a pointer to a structure that contains the IOCL, error status, and seek data. Users can use this data for error recovery. Or, they can copy the IOCL to perform their own direct I/O requests when the process has returned from the signal. If the IOCL data is to be sent back as an I/O request, it must be copied to a static structure before issuing the request. Note that static structures do not always exist in FORTRAN, except in commons.

In no-wait mode, calls to **dfread, dfwrite, dfexcp, dffstat, dfvstat**, or **dfseek** are guaranteed not to block the process while I/O is occurring. In wait mode, the process resumes execution only after the I/O has completed.

Certain operations cause the process to block, regardless of the wait/no-wait mode of operation:

- All **dfopen**, **dfcreate**, **dfrename**, **dfdelete**, **dfsetEOF**, and **dfextend** calls will cause the process to block until the call is completed.
- Calling **dfwrite** and autoextending a file will cause the process to block while the file is extended.
- Calling **dfclose** may cause the process to be blocked while waiting for access to shared memory.

Maximum transfer size

The maximum single transfer size is 26 tracks. (On a 300Mb disk, a cylinder is 19 tracks.) The maximum number of bytes per transfer depends on the number of bytes per track. Transfers larger than 26 tracks must be separated into two or more I/O requests.

Random access within a file

Files can be accessed randomly by disk blocks, using **dfseek**. When each read/write has completed, a file pointer is updated to the next unreferenced disk block. You can change the pointer by calling the **dfseek** routine. Note that the resolution of the pointer is a single disk block.

Logical EOF

Unless the file is truncated, the logical end of file denotes the largest block number of the file ever written using **dfwrite**. This block number is kept local to the process. It is not visible to other processes unless it is saved by calling the **dfsetEOF** function each time it is to be updated. (**NOTE:** The FORTRAN version of this function is called **dfseteof**.) When this function is called, the logical EOF is set to be the larger of (1) the largest block number written since the file was opened and (2) the previous logical end of file.

There is one copy of the logical EOF for each file, regardless of how many times the file is opened. It is updated whenever **dfsetEOF** is called. If the file is opened and the TRUNCATE flag is set, the logical EOF is reset to the start of the file. Each time **dfsetEOF** is called, the disk directory entry for the file is updated. The process blocks while this update is made. When an end of file is set via **dfsetEOF**, the date and time are saved in the directory.

Note that the logical EOF differs from the physical end of file, which denotes the end of the file's allocated space.

Extending files

Files may be extended manually or, when the user attempts to write past the physical end of file, automatically. When opening a file, you can specify, with the DF_AUTOEXT flag, that it can be extended automatically (autoextended).

When a file is autoextended, an effort is made to meet user-specified minimum extent size and the maximum gap size, if possible. If this is not possible, the file will be extended with the largest extents available. Note that, when autoextending, the process will be blocked from execution while the directory is being updated.

When a file is extended manually, user specifications for minimum extent size and maximum gap size are guaranteed, because if they cannot be met, the call will fail.

Users can optionally disregard extent and gap sizes when explicitly extending a file.

Multiple process access to the same file system

Any number of processes may access a file system simultaneously, with semaphores controlling directory access.

Files may be opened for exclusive or shared use. A user trying to open a file that is in exclusive use will fail, but anyone can access a shared file. It is the user's responsibility to ensure that use of a shared file is coordinated by using semaphores when necessary.

Error correction

Error correction is not performed on any disk errors encountered by the direct file system, with the exception of seek errors. Seek errors are recalibrated and retried once. All other errors are reported back to the user in the I/O completion notification procedure.

Access controls

There are no UTX/32 access controls on the direct file system. This implies that no owners or groups are associated with the files.

Reliability

The direct file system is reliable. The file structure will retain its integrity when processes abort or the system crashes. If the system crashes in a critical section, the only free space that can be lost is free space that can be recovered by rebuilding the free list when the disk is next mounted. The direct file system is not protected when users issue their own IOCLs.

Speed

The direct file system is capable of doing single-revolution opens. These are file opens in which the directory entry for the file is accessed on the first disk read of the directory. To achieve this, the file must be created as DF_FASTOPEN.

Due to the hashing scheme used to access the directory, occasionally a particular file name may not be creatable as DF_FASTOPEN. If this happens, an error condition will be returned. The best thing to do is to try again using a different file name.

Sizing information

All calls to the direct file system represent sizes in bytes by default. This promotes compatibility with the UNIX philosophy and encourages code that is less dependent on the physical characteristics of the disk medium.

User-definable limits

The following parameters are listed in **/usr/include/dfconfig.h**, and can be adjusted when configuring the direct file system. Default values are shown in brackets.

MAXFILES
: Maximum number of concurrently open files per process [20]

MAXVOLUMES
: Maximum number of concurrently mounted volumes [10]

MAXSYSFILES
: Maximum number of concurrently open (different) files for all processes [40]

MAXBSIZE
: Maximum disk block size [1024 bytes]

MAXIOREQ
: Maximum number of queued I/O requests per process [16]

MAXTRACKS
: Maximum number of tracks per disk [15637 tracks]

For examples of using the direct file system facility, refer to Section A.5, "Direct File System."

Here is an example user session that exercises some of the facility's user and system administrator commands:

```
# cat /etc/dffstab
disk0a:/dev/rdk0a
disk0b:/dev/rdk0b
disk1a:/dev/rdk4a
# dfmount disk0a
# dfcreate 200t /disk0a/file
# dfextend -z 16b /disk0a/file
# dfwrite /disk0a/file < /etc/dffstab
# dfread -e /disk0a/file
disk0a:/dev/rdk0a
disk0b:/dev/rdk0b
disk1a:/dev/rdk4a
# dfdelete /disk0a/file
# dfumount disk0a
```

In this example, the direct file volume **disk0a** is mounted. A file named **file** is created on this volume, with an initial size of 200 tracks. The file is extended by 16 bytes, and the extended portion of the file is zeroed after it is created. The contents of /etc/**dffstab**, a standard UTX/32 file, are written to the new direct file. The direct file is then opened for exclusive use, and its contents are read and printed to standard output. Finally, the direct file is deleted and the direct file volume is unmounted.

8.3 Special Considerations

The following restrictions apply to the direct file system:

Use of signals

When signals are used, be sure to observe the following guidelines:

- Signal 29 (SIGDIRF) is reserved for use by the direct file system library routines.
- Signals must always return properly.
- If a process receives a signal, requests from that process to the direct file system may be rejected while the process is in the signal handler. If the file system is interrupted by a signal while processing a request and the signal handler calls a direct file system function, the direct file system will reject the request and return EINTR as an error. To avoid this, do not mix direct file requests from the normal process environment with those from a signal handler.
- The procedure called for I/O completion notification is called within a signal handler. Therefore, all restrictions applying to signal handlers also apply to this procedure.

Lockdown

While using the direct file system, the process must have all its pages locked down.

Killing a process

If a process using the direct file system is killed, the file system may hang. A reboot will be necessary.

Disk formatting

Before **dfmkfs** can create a direct file system, the disk must be formatted using **prep**; see *prep*(8).

9 High-Speed Input/Output Support

9.1 Aspects of High-Speed Input/Output

UTX/32 support for high-speed I/O consists primarily of

- Class E I/O support, which includes I/O interface (IOI) extensions and a generic HSD device driver
- A direct I/O facility

Detailed information can be found in the *UTX/32 Input/Output Subsystem Guide*.

9.2 Support for Class E I/O

Many real-time applications use devices connected through the Gould high-speed device (HSD) interface. This interface is a SelBUS™ card that responds to class E machine instructions. UTX/32 support for the HSD consists of a generic device driver for a "typical" HSD-interfaced device and a set of modifications to the IOI to support class E operations similar to those for class F devices.

The generic driver, **ce**, is a simple UTX/32 device driver that uses the class E facilities of the IOI; see *ce*(7RT). Since the HSD supports a wide variety of devices, most users will have to write their own device drivers to suit the needs of their specific devices. The generic driver is intended to help in this process by demonstrating the form of a UTX/32 device driver and the use of the IOI facilities for low-level operations.

Refer to the *UTX/32 Input/Output Subsystem Guide* for a more extensive discussion of the IOI extensions, the generic HSD driver, and the writing of custom drivers.

9.3 The Direct I/O Facility

Many real-time tasks need faster access to devices and files than can be provided using standard UTX/32 I/O, which is designed for time-sharing. UTX/32 therefore includes facilities allowing real-time applications to establish direct attachments to devices and to perform I/O and control operations without going through the normal UTX/32 I/O and file system facilities. While the conveniences of standard UTX/32 I/O are lost, performance and control are greatly increased.

The direct I/O facility and its use are described in detail in the *UTX/32 Input/Output Subsystem Guide*. The direct I/O device driver, on which the facility is built, is described in detail in *dio*(7RT). The subroutines that support the system are documented by those manual pages beginning with *dio* in Section 3, "Subroutines," in the *UTX/32 BSD Programmer's Reference Manual.*

For examples of the use of direct I/O in code, refer to Section A.6, "Direct I/O," and Section B.1, "Header File for Model Program."

10 Connected Interrupts

10.1 Overview

Real-time applications must be able to respond quickly to interrupts, and response time is quickest when an application program is connected to one or more interrupt levels. *Connected interrupts* provide an easy and reliable way of connecting an application process to an interrupt level. With connected interrupts, an application may process device interrupts in user space or just choose to be notified when interrupts occur.

When the term *connected interrupt* is used, it is never meant to imply that an interrupt is connected to anything. Rather, it is a term commonly used to refer to the situation in which a connection exists between an application process and an interrupt level. In such situations, the connected process handles all interrupts that occur at that level.

Connected interrupts come in two varieties: directly connected and indirectly connected. Directly connected interrupts enable the user to process interrupts at interrupt time in user space. The user will usually be privileged in order to do this, since device control requires privileged instructions. Indirectly connected interrupts use an event mechanism to notify a number of processes when an interrupt occurs. Currently, signals are used as the events.

Connected interrupts have the following uses:

- Device control
- Performance evaluation
- Statistics gathering
- Scheduling

By using device control through connected interrupts, new device drivers can be written and tested without recompiling the kernel and rebooting for each test.

Performance can be evaluated by a separate process that connects to an interrupt from a Real-Time Option Module (RTOM) interval timer. A process can be coded to send an interrupt to the RTOM periodically. The interrupt routine would then increment a counter. Querying the counter at the beginning and at the end of the evaluated code provides a means of timing. Statistics can be gathered in much the same manner.

Although cyclic scheduling provides a better interface for process scheduling, certain processes will require better performance than the cyclic scheduler can provide and thus will use connected interrupts scheduling.

A user process installs a connected interrupt by making two system calls: **plant** and **graft**. The **plant** system call performs scratchpad initialization and saves context for later use during an interrupt. The **graft** system call installs the Interrupt Service Routine (ISR). The user supplies the ISR for directly connected interrupts. The kernel supplies the ISR if the interrupt level is indirectly connected. Several processes may indirectly connect to the same interrupt level by calling **graft** with appropriate arguments, although the same process cannot connect to the same interrupt level more than once. The same process must issue both the **plant** and the **graft** system calls for directly connected interrupts. The user process is responsible for enabling interrupts, whether it is directly or indirectly connected. Library routines are provided to do this.

10.1.1 Configuration

Use of connected interrupts must be configured. This release of UTX/32 is distributed with no connected interrupts. Attempts to use the connected interrupts functionality will result in an ENORESOURCE error. See Chapter 7, "Reconfiguring the System" in the *UTX/32 Operations Guide* for instructions on configuring connected interrupts.

10.2 User Interface

10.2.1 Indirectly Connected Interrupt Example

Section A.7.1, "Example 1," presents a simple example of an indirectly connected interrupt. The program connects to the IOP RTOM interrupt level 0x50 (80) at address 0x7f0d. The program sets up the connected interrupt by calling the **plant** and the **graft** system calls. Two signal handlers are set up: one for killing the process, and one for receiving interrupts. The process then enables the interrupt level and waits for interrupts. When an interrupt is received, the signal handler **sig_handler** is called as a result of the process being signaled. The process then enables the interrupt level and waits for interrupts.

Note that the indirectly connected process does not need to deactivate the interrupt level in the signal handler. This has already been done by the operating system. It is possible for multiple interrupts to occur before the process' signal handler runs. In this case, only one signal will be delivered, and the signal handler will execute only once.

It is not required that an indirectly connected process be privileged. None of its work requires the execution of privileged machine instructions. It is recommended the process not be privileged for greater protection of the kernel.

The kernel tries to remove connections when a process exits, although for indirectly connected interrupts, it cannot always do so. All programs that indirectly connect should contain code to remove the connection to avoid such failure. Since many processes can **graft** to one indirectly connected interrupt, each should remove that connection when it terminates. However, the connection created by **plant** will remain unless some process removes it. This connection cannot be removed until all indirect **grafts** have been removed. For this reason, there should be a governing process that waits for the indirectly connected processes to exit and then removes the connection using **uproot**. If, despite all caution, a connection remains, it may be removed by using the **cis** and **cirm** programs.

10.2.2 Directly Connected Interrupt Example

Section A.7.2, "Example 2," presents a simple example of a directly connected interrupt. The program connects to the IOP RTOM interrupt level 0x50 (80) at address 0x7f0d. The process begins by locking itself into memory and targeting itself to the CPU. A signal handler is set up so that the process will clean up after itself and exit when it receives a SIGINT interrupt from the terminal. The process initializes the connected interrupt using the **plant** and **graft** system calls. Interrupts are enabled using the **ei** library routine and the process waits for an interrupt. When an interrupt is received, the **intr_handler** routine is called. Note that the **intr_handler** deactivates interrupts just before returning.

This is different from indirectly connected processes, which do not have to deactivate the interrupt level. A directly connected process *must* deactivate the interrupt level in the interrupt handler.

The process may make no system calls within the interrupt service routine, including system calls from within library calls. For example, the user will not be able to debug the interrupt service routine by inserting **printf** statements. Debugging in this mode will be limited to console debugging (write stops, instruction stops, etc.) and to setting global variables that can be read by some other process or the same process at some later time. Attempts to make system calls will result in a signal (SIGSYS) as well as a failed return from the system call (ESYSUNAVAIL). The signal should terminate the process, although it may take awhile if the process is running on the IPU when the interrupt is taken. The signal will be noticed as soon as the process enters the kernel, either for a scheduling change or to make a system call.

A process can be running in both processors simultaneously on PowerNode™ machines. The process can be running on the IPU and its interrupt service routine can be running on the CPU. This will ordinarily not cause any problems, except perhaps with cache coherency. If a simultaneous read and write occur to the same doubleword of memory, then the processor that did the read will not get the updated write until the cache is flushed. It is therefore strongly suggested that users target to the CPU a process that is directly connected (see the *targetcpu*(2RT) manual page).

Even though system calls are traps, they are handled differently from other kinds of traps. If a directly connected process traps while in the interrupt routine, the process will trap in an ordinary way. However, the effect of the trap will not be felt until the process enters the kernel. This effect may be delayed if the process is mapped in the IPU and is doing computation-bound processing. It is also possible for the process to be trapping from elsewhere in the process when an interrupt is taken. In the rare instance that the interrupt is connected to the same process that is already trapping, the kernel will not proceed correctly. Behavior at this point is undefined, but a panic or machine halt is likely to occur in the near future.

The process must perform interrupt control. This means that the process must enable the interrupt and deactivate it during the interrupt service routine. Note that this has been done in the example program using the library routine **hwprivdai.**

It is guaranteed that the user's ISR will be entered with interrupts blocked. The user may choose not to unblock interrupts, but this means that the ISR is uninterruptible. If the ISR is a very long one, then the user may choose to unblock interrupts. This can be accomplished with the **hwprivuei** library routine. This allows interrupts at a higher priority level to interrupt the current interrupt service routine. The higher priority ISR will complete and then continue the interrupted ISR.

If interrupts are unblocked, then the current interrupt should not be deactivated until interrupts are again blocked. This action disallows receiving an interrupt from the same level that is currently being processed. If the user chooses to leave interrupts blocked, then no action in the interrupt routine is necessary. If the user chooses to unblock interrupts, then the interrupt routine should perform the following steps:

1. Begin interrupt
2. Unblock interrupts
3. Perform interrupt processing
4. Block interrupts
5. Deactivate interrupts
6. Return from interrupt

The user must not use the same interrupt stack for different interrupt levels, especially if interrupts are unblocked in the user's ISR. Such a situation would make it possible for one interrupt to interrupt the other and use the same stack, thus destroying the integrity of the values stored on the stack by the first ISR.

A user may change the default settings in the Program Status Doubleword (PSD), although this action is not expected. One possible setting change is that of the block/unblock mode bit. The **plant** system call currently sets this bit to be unblocked. That is, when the CPU accepts an interrupt, it blocks all interrupts at lower priority levels. The user must deactivate the interrupt to restore interrupt processing for that and lower priorities. Blocked mode automatically issues the deactivate interrupt command, but it blocks all interrupts. In this case, the user must be careful not to unblock interrupts, because another interrupt at the same priority level might come in immediately and destroy the integrity of the stack.

The file **/usr/include/sel/psd.h** contains the definition of the PSD structure. The **plant** system call sets the fields in an ICB's new PSD as follows:

- The privileged bit is set.
- The condition codes are all reset.
- The extended addressing option bit is reset.
- The base register mode bit is set.
- The arithmetic exception trap is disabled.
- The program counter is the address of a known function within the kernel. This function will later call the user's function.
- The mapped environment bit is set.
- The retain current map bit is reset.
- The interrupt control flags are all zero.
- The current process index is obtained from the current PSD.

Debuggers such as **adb** and **dbx** will not work in a directly connected interrupt routine. The user must use the console debugging commands such as IS (instruction stop) and WS (write stop). Another possible mode of debugging is to use shared memory by having the process write debugging information into shared memory using a protocol that is understood by some other process. The other process can then digest and print this information.

Some interrupt control is available through system calls, so some debugging can be done using indirectly connected interrupts. When the program works using indirectly connected interrupts, it will likely also work using directly connected interrupts. Note that the interrupt control system calls are available only to the superuser.

The kernel automatically removes direct connections when a process exits. This is useful when a program terminates unexpectedly. The program should, however, **prune** and **uproot** upon normal termination.

The process that uses directly connected interrupts will usually be privileged. This allows the use of privileged instructions for interrupt control and I/O. Note that this is a dangerous situation, because the user can now write and destroy parts of the operating system. Once privileged, the user can do considerable damage that might not be immediately apparent, and appropriate caution is urged.

10.3 Stack Addresses

Stacks on CPL hardware grow downwards, toward decreasing addresses. The address that is given to the **graft** system call should be the highest address of the memory allocated for the stack.

If the stack were declared in C as

```
int    Stack[500],
```

then the address given to **graft** should be

```
&Stack[499]
```

If the stack were declared in FORTRAN as

```
integer Stack(500),
```

then the address given to **graft** should be

```
Stack(500)
```

The **graft** system call will handle all requirements for alignment.

The user must provide a large enough stack. Stack overflow will generally not be detected, especially if the user is privileged. Stack overflow will result in writing to data structures that are near the stack (usually declared before the stack in the program). Many nested function calls and much storage for local variables increase the memory required for stacks. Five hundred words should be sufficient for most medium-sized interrupt routines. One hundred words is probably sufficient for a minimal interrupt routine. An interrupt routine that just calls **hwprivdai** and returns requires 16 words (24 words including enough for file alignment). Because stacks must be file (32 byte) aligned, enough memory must be reserved for the worst case alignment, that is, in the worst case, enough memory must be reserved for the minimum requirement plus 32 bytes. If memory is at a premium, then the user should determine exactly how much memory the interrupt routine and all of its called functions need for stack, being watchful of recursive routines.

10.4 Helpful Programs

The **cis** and **cirm** programs allow the user to get status on outstanding connections and remove them if possible. **cis** prints the status of the requested connections, including the interrupt level, the class and address of the device, and the type of connection, if connected at all. Given this information, the user may remove selected interrupt levels using **cirm**. **Cis** will display all connections that have been made, including those interrupt levels in use by the kernel. This information is useful when trying to determine what level to use for a given program. The **cistatus** system call will also return this information. See *cis*(1RT) and *cirm*(1RT) for more details.

11 Suspend and Resume

11.1 Overview

The **suspend** and **resume** system calls provide a fast, simple method for process context switching. By invoking **suspend**, an active process relinquishes the CPU without the overhead of sending and handling signals. By invoking **resume**, a process causes one or more suspended processes to be immediately placed on the run queue.

11.2 User Interface

11.2.1 suspend

A suspended process remains suspended until it is resumed or until it receives a signal. If the suspended process is resumed, **suspend** returns a value of 0. If a suspended process receives a signal, execution continues, **suspend** returns a value of −1, and **errno** is set to equal EINTR. For more information, see *suspend*(2RF/RT).

A suspended process has the same characteristics as a process that was stopped by sending a SIGSTOP via **signal**.

The following examples show how to call **suspend**. (For more detailed examples on how to use **suspend**, refer to Section A.8, "Suspend and Resume.") In the first example, a process suspends itself. If it receives a signal, the process executes, but it immediately suspends itself again. Its execution continues normally only after it is resumed by another process.

```
/* suspend repeatedly until resumed */
while ((ret = suspend() == -1) && (errno == EINTR))
{
            /* NULL statement */
}

/* suspend failed or process was resumed */
if (ret != 0) /* if suspend failed */
{
            printf("unexpected return from suspend =%d, errno = %d",
                     ret, errno);
            exit(-1);
}

/* suspend and resume succeeded so continue with program */
```

In the following example, the process that called **suspend** will continue executing, either when the process is resumed via **resume** or when it receives a signal.

```
if ((ret = suspend() != 0) && (ret != -1) || (errno != EINTR)) {
          printf("unexpected return from suspend =%d, errno = %d",
                 ret, errno);
          exit(-1);
          }
/* process was resumed or received a signal so
 * continue with program
 */
```

11.2.2 resume

The **resume** system call awakens one or more suspended processes and adds them to the run queue. It optionally suspends the calling program after resuming the specified processes.

If only one process is to be resumed, the calling program invokes **resume** with two arguments. The first argument is a flags field. When the calling program resumes only one process, none of the bits in this field need to be set. The second argument is the process id of the process to be resumed. If **resume** fails to resume the specified process, it returns a value of -1 and sets **errno** to indicate the reason for the failure.

If more than one process is to be resumed, the calling program invokes **resume** with four arguments. The first argument is a flags field in which the SR_MULTIPLEPIDS bit must be set. The second argument is the address of an array containing the process ids of the processes to be resumed, and the third argument is the number of processes to be resumed. The fourth argument is the address of an array into which **resume** copies the return status for the processes, as each process may have a different return status.

Two examples of the use of **resume** are as follows:

```
resume(flags, pid)

resume(flags|SR_MULTIPLEPIDS, pidlist, numb_pids_to_resume, statuslist)
```

By default, **resume** resumes the specified processes and returns control to the calling program. If the calling program sets the SR_SUSPEND bit in the first argument, it is suspended after the specified processes are resumed. The **resume** system call returns control to the calling program only when another program awakens it by invoking **resume** with the calling program's process id.

See *resume*(2RT/RF), and refer to Section A.8, "Suspend and Resume," for detailed examples on the use of **resume**.

In the following example, the calling program resumes a suspended process.

```
/* set flags field to specify to continue with execution
 * after resuming suspended process */
flags = 0;

/* pid is the process id of a suspended process and
 * has been set somewhere else in the program
 */
if (resume(flags, pid) == -1) {
            printf("resume failed errno = %d",errno)
            exit(-1)
            }
```

In the following example, the calling program resumes a suspended process, then immediately suspends itself.

```
/* set flags field to specify that caller should
 * suspend itself after resuming the suspended process
 */
flags = SR_SUSPEND;

/* pid is the process id of a suspended process and
 * has been set somewhere else in the program
 */
if (resume(flags, pid) == -1) {
            printf("resume failed errno = %d",errno);
            exit(-1);
            }
```

In the following example, the calling program resumes multiple processes.

```
/* declare space for pidlist and statuslist */
int pidlist[25];
int statuslist[25];
                .
                .
    set pidlist[0] through pidlist[24] to
    equal the process ids of 25 suspended
    processes
                .
                .

/* set flags field to continue execution
 * after resuming suspended process and to specify
 * that multiple processes should be resumed
 */
flags = SR_MULTIPLEPIDS;
numb_pids = 25;

if (resume(flags, pidlist, numb_pids, statuslist) == -1) {
          printf("resume failed errno = %d",errno)
          for(counter = 0; counter <numb_pids; counter++)
                     printf("statuslist[%d] = %d\n",
                             counter, statuslist[counter]);
          exit(-1);
          }
```

11.3 Special Considerations

- A process resumed via **resume** is not guaranteed to execute immediately. Although such a process is immediately placed in the run queue, it will not execute until it becomes the process with the highest priority.
- When a child process suspends itself, the parent process does not automatically receive notification, via SIGCHILD, that the child has changed state.
- A child of **vfork** cannot be suspended until it has done an **exec**, because such a suspension may cause deadlock.
- Not all signals can be handled by a suspended process. For more information, see *signal*(3).

12 Memory Classes

12.1 Overview

Memory classes provide the user with a mechanism for controlling physical memory for special purposes.

The conventional model of memory use rests on two assumptions: that all memory is alike and that a process does not depend on the particular physical addresses of the memory it uses. Given those assumptions, the physical memory attached to a machine can be treated as a single pool of indistinguishable pages. When a process is given a page by the operating system—at process startup, when expanding memory use at runtime, or because of paging—any physical page may be chosen.

In real-time systems, the basic assumptions do not hold, for these reasons:

1. Not all physical memory is alike. For example, some physical memory may be faster but more expensive than the rest. Real-time applications may require that such memory be held in a separate pool so that it can be allocated when needed.
2. Some memory may be shared among several machines. To allow processes to communicate with their counterparts, each must be able to attach precisely the memory that is shared. Hence, a process must be able to request pages of memory at particular physical addresses.

The UTX/32 implementation of memory classes introduces two related ways to group memory for special purposes, i.e., to create *special memory*:

Memory extents

A *memory extent* describes a single, continuous range of memory that is set aside from general use. Memory outside a memory extent is allocated to processes in the usual ways; memory in an extent is set aside from general use and must be specially allocated. Generally, an extent is made up entirely of a particular kind of memory, such as shadow (fast) memory or reflective (shared) memory.

Memory extents are never paged or swapped, and they cannot overlap.

Memory regions

A *memory region* is a subset of a memory extent. It may cover the entire extent or be as small as a single page. The memory region is the unit of allocation; processes request special memory by requesting particular regions.

The UTX/32 implementation of memory classes is upwardly compatible with System V shared memory. The System V implementation is based on *shared memory objects*, which are sets of pages with some associated information. Two processes share memory when both allocate the same shared memory object.

UTX/32 extends the System V implementation in two ways:

1. In System V, shared memory objects are referred to by numeric keys. In UTX/32, shared memory objects may also have string names; they may be created with names, keys, or both.
2. In System V, the creator of a shared memory object may specify precisely which pages it contains by giving it pages from a memory extent. In UTX/32, shared memory objects whose pages come from a memory extent are called memory regions.

12.2 An Example

The FORTRAN language provides shared memory between processes through global commons and datapools.

1. A FORTRAN program **F** declares a global common **X**, which is a set of data with a group name.
2. When **F** begins execution, the global common is placed into the shared memory object named **X**. If no such object exists, one is created in general memory. If the shared memory object is a memory region, the global common is placed in the special memory named by the region.

   NOTE: In the current implementation, global commons and datapools remain uninitialized on process startup.
3. When another program, **G**, using global common **X** begins execution, it shares memory with **F**.
4. Program **G** may be moved to a completely new machine. If the two machines are connected with shared memory, and if both have regions **X** defined to be at the same offset within the shared memory, **G** will run without recompilation.

12.3 Contiguous and Noncontiguous Memory Extents

When special memory is shared with another machine, a process must be able to allocate a particular range of physical pages. But when special memory is simply faster than general memory, any page of fast memory is as good as any other. This distinction is captured by two types of memory extents: contiguous and noncontiguous. They differ in the type of regions they contain.

Contiguous memory extents

A region within a *contiguous extent* is a set of pages that begins at a specified offset from the start of the extent and extends without gaps for a specified number of pages. Any process allocating that region will receive the same set of pages.

Noncontiguous memory extents

Within a *noncontiguous extent*, there is typically only one region, called the *template region*. This region contains all otherwise unused pages of the extent. It is created when the memory extent is created and has the same name as the memory extent. (It is, in actuality, simply the name by which memory in the extent can be allocated.)

A process may allocate the template region, but instead of receiving all the pages in the extent, it receives a set of pages of the requested size. These pages are private to the process. The template region thus serves as a pool of pages to be allocated in much the same way as general memory pages are allocated by the **sbrk** system call (see *sbrk*(2)).

Named regions (other than the template region) within a noncontiguous memory extent are still necessary for sharing memory. For example, if two processes want to share a global common in fast memory, they must make a region within the memory extent. That region will be permanently allocated pages the processes can share; these pages are not contained in the template region.

12.4 Permissions

Shared memory objects obey the same rules for ownership and permission that files do. See *intro*(2) for details. In brief, shared memory objects have a specified owner and group. They also have permission modes that specify whether the owner, the group, or other users may read or write the pages in the object. When a process attempts to attach to the shared memory object, it is classified as being either the owner, in the owning group, or other. The requested access type (read-only or read/write) is checked against the access permissions.

The ability to create and destroy regions is subject to the restrictions imposed by the real-time access control mechanism. A noncontiguous memory extent's template region is owned by the superuser (user ID 0) and group 0. A template region's mode is always 0666, which allows any process to allocate pages for reading or writing.

12.5 Reflective Memory

Reflective memory is a form of shared memory between machines. Writes into reflective memory on one machine may be ''reflected'' over a special bus to reflective memory on other machines. All reads are from local memory.

The operating system may specify the range of reflective memory that is to reflect writes and receive reflected writes. The reflected locations are referred to as the *window*. UTX/32 supports this feature by providing a system call **mem_reflect** (see *mem_reflect*(2RT)), which sets the reflective memory window to a particular region.

12.6 Summary of Special Memory Support

12.6.1 Creation

Memory extents are created at system startup. They are defined in the system configuration file. Instructions on editing the system configuration file to create memory extents may be found in the *UTX/32 Operations Guide*.

Memory regions may be defined in the system configuration file. They can also be created with the **mkregion** utility or within a program by the **mkregion** system call. See *mkregion*(8RT), *mkregion*(2RT), or *mkregion*(3RF).

Shared memory objects that are not memory regions (that is, not within memory extents) can be created with the **shmget** or **shmgetbyname** system calls. See *shmget*(2RT) and *shmgetbyname*(2RT).

Shared memory objects may also be created by the execution of a FORTRAN program containing a named global common or datapool. If no shared memory segment or region with that name exists, a shared memory segment will be created in general memory (never in special memory).

12.6.2 Allocation

Regions are allocated in two steps. In the first step, the region is allocated to the process with **shmget** or **shmgetbyname**. In the second step, the region is attached to the process's virtual address space with the **shmat** system call (see the *shmop*(2RT) manual page). The same sequence is used for shared memory objects that are not regions.

Allocation of pages from a noncontiguous memory extent requires the same two steps. **shmget** or **shmgetbyname** is used on the extent's template region. The result is a private shared memory object with the requested number of pages. That private object is then attached using **shmat**.

The two steps take place automatically when a FORTRAN program naming a shared global common or datapool is placed into execution.

12.6.3 Destruction

A shared memory object created during process startup is destroyed when the last process using that object exits. For example, a global common created by a FORTRAN program will be destroyed when the last program using it exits.

In no other case is a shared memory object or region automatically destroyed. In particular, a private shared memory object created by **shmget** or **shmgetbyname** will not be destroyed when the process exits, even though no other process may attach to it.

Regions may be destroyed with the **rmregion** utility or with the **rmregion** system call. See *rmregion*(8RT), *rmregion*(2RT), or *rmregion*(3RF).

Shared memory segments that are not regions may be destroyed with the **ipcrm** utility or the **shmctl** system call. See *ipcrm*(1RT) or *shmctl*(2RT).

12.6.4 Inspection

The **ipcs** utility (see *ipcs*(1RT)) can be used from the shell to inspect shared memory segments and regions. There is currently no defined way to inspect regions from within a program.

13 Instruction Execution Modes

13.1 Overview

The Gould CONCEPT Product Line (CPL) architecture supports two modes of instruction execution, *privileged* and *unprivileged.* These two modes are necessary because certain instructions in the CPU instruction set can be executed only by programs that have special privileges. On UNIX-based operating systems such as UTX/32, machines typically execute in privileged mode only within the kernel, and the existence of two distinct modes is transparent to user programs. (Kernel mode thus mistakenly became synonymous with privileged execution, although privileged mode is not really equivalent to kernel mode.)

However, with the introduction of connected interrupts (see Chapter 10, "Connected Interrupts"), a user program must be able to execute privileged instructions that affect an interrupt level to which it has established a connection. Two new system calls, **hwpriv** and **hwunpriv**, allow a user program to run in privileged instruction execution mode and unprivileged instruction execution mode, respectively. Because these system calls regulate the execution of CPU instructions, i.e., the execution of instructions at the hardware level, the two modes are often referred to as *hardware privileged mode* and *hardware unprivileged mode*, respectively, and hence the names of the system calls.

13.2 Instructions Requiring Privileged Mode

A process begins execution in hardware unprivileged mode. However, the process must switch to hardware privileged mode before it can execute the following kinds of instructions:

- Instructions related to interrupt processing, such as Enable Interrupt (EI) and Disable Interrupt (DI)
- Instructions that can modify a machine's memory mapping registers, such as Load Program Status Doubleword and Change Maps (LPSDCM)
- Input/Output instructions, such as Start I/O (SIO), Halt I/O (HIO), and Command Device (CD)
- Instructions that can place the CPU in a state that requires an operator action at the console, such as HALT
- Instructions that can change the state of the CPU, such as WAIT and Load Program Status Doubleword (LPSD)

These instructions should be used with extreme caution because of the potential for compromising the integrity of the system. (See *adb*(1) for a complete listing of the CPU instruction set.) Their improper use can result in corruption of data in main memory or secondary storage, or even a crash of the entire system. When all privileged instructions have been executed, a process should usually invoke **hwunpriv** to revert back to hardware unprivileged mode.

Processes executing in these two modes are differentiated by a bit in the Program Status Doubleword (PSD). When the bit is set to one, the process is executing in hardware privileged mode. Any attempt to execute a privileged instruction while in hardware unprivileged mode causes an exception trap.

13.3 Ways to Execute Privileged Instructions

Once a process has executed the **hwpriv** system call, there are three ways to execute a privileged instruction:

1. A C program may pass the privileged instruction as an argument to an *asm* statement, causing the compiler to insert the instruction directly into its assembly language output. This method carries the least overhead, but because Gould Common C will not support *asms*, its use is discouraged for portability reasons.
2. A C or FORTRAN program may invoke a library routine that executes the privileged instruction. The real-time C library contains one routine for each of the privileged instructions, as does the real-time FORTRAN library. The library routines are named by adding the **hwpriv** prefix to the name of the privileged instruction. For example, the **hwprivei** library routine executes the EI instruction. For more details, see the following manpages:
   - *hwpriv_intr_control*(3RT)
   - *hwpriv_intr_control*(3RF)
   - *hwpriv_io_control*(3RT)
   - *hwpriv_io_control*(3RF)
3. A privileged instruction may be embedded in an assembler routine, and a C or FORTRAN program may make a call to that routine. This is the method of choice only when a code segment is written in assembly language for speed and efficiency considerations, and it is natural for the segment to contain a privileged instruction. If an assembler routine would contain nothing but the privileged instruction, use a library routine instead.

Some of the privileged instructions may be executed by invoking a system call while in unprivileged instruction execution mode. This mechanism is provided for debugging purposes only, because system call overhead makes it undesirable for regular use. A program may invoke the **intctl** system call directly, or it may use the library routines that invoke the system call after generating its instruction-specific arguments. See *intctl*(2RT), *intctl*(2RF), *io_control*(3RT), *io_control*(3RF), *interrupt_control*(3RT), and *interrupt_control*(3RF) for more information.

Use of the **hwpriv** and **hwunpriv** system calls is subject to the restrictions imposed by the real-time access control mechanism.

See the *Gould V6/V9 CPU Reference Manual* that accompanies Gould CPL hardware for more information about execution modes, and refer to *hwpriv*(2RT), *hwunpriv*(2RT), *hwpriv*(2RF), and *hwunpriv*(2RF) for details about the use of the system calls.

Appendix A
General Examples

This appendix contains simple examples written in C and in FORTRAN. These examples illustrate how to use many of the UTX/32 real-time features.

A.1 Real-Time Scheduling

This section contains two examples of real-time scheduling, one in FORTRAN and one in C.

A.1.1 FORTRAN Real-Time Scheduling Example

The following FORTRAN test program exercises the real-time FORTRAN library routines that control real-time priority and processor targeting, such as **setrealpriority** and **settargetcpumask**.

```
c
c      Compile with:  fort -o rtschedtest rtschedtest.f -lrtf -lrt
c
       program rtschedtest

       integer cpumask
       integer newprio
       parameter (newprio = 5)
       integer newmask,mask,prio,oldprio,ret
       integer i
       integer settargetcpumask, gettargetcpumask, getactivecpumask
       integer unixscheduling, setrealpriority, getrealpriority

       cpumask (n) = 2**n

c
c      Request real time priority 5
c
       oldprio = setrealpriority (0,newprio)
       if (oldprio .lt. 0) then
                         write (6,*) 'setrealpriority failed ret=', oldprio
                         call exit (-1)
       endif

c
c      Get the current priority
c
       prio = getrealpriority (0)
       if (prio .lt. 0) then
                         write (6,*) 'getrealpriority failed ret=', prio
                         call exit (-1)
       endif

c
c      Did it really work?
c
       if (prio .ne. newprio) then
                         write (6,*) 'priority not correct: new=',newprio,
      +                                  ' returned=',prio
                         call exit (-1)
       endif

c
c      Bias the process to the IPU
c
       write (6,*) 'biased to the ipu'
       newmask = cpumask (1)
       mask = settargetcpumask (0,newmask)
       if (mask .lt. 0) then
```

```
                        write (6,*) 'settargetcpumask failed ret=', mask
                        call exit (-1)
      endif

      mask = gettargetcpumask (0)
      if (mask .lt. 0) then
                        write (6,*) 'gettargetcpumask failed ret=', mask
                        call exit (-1)
      endif
      write (6,*) 'mask=',mask

c
c     Did it really work?
c
      if (mask .ne. newmask) then
                        write (6,*) 'mask not correct: new=',newmask,
     +                                ' returned=',mask
                        call exit (-1)
      endif

c
c     Waste time.  Real processing should go here if this were a real
c     program.
c
      do 10 i=1,10000000
                        continue
   10 continue

c
c     What are the active CPUs that can be used?
c
      mask = getactivecpumask ()
      if (mask .lt. 0) then
                        write (6,*) 'getactivecpumask failed ret=', mask
                        call exit (-1)
      endif

c
c     Return to normal priority
c
      ret = unixscheduling (0)
      if (ret .lt. 0) then
                        write (6,*) 'unixscheduling failed ret=', ret
                        call exit (-1)
      endif

c
c     Waste some more time
c
      do 20 i=1,10000000
                        continue
   20 continue

      write (6,*) 'test passes'

      stop
      end
```

A.1.2 C Real-Time Scheduling Example

The following C program uses the real-time scheduling library routines. This program gives the user a convenient way to control the real-time priority or processor-targeting of a process.

```
/*
 * Compile with:  cc -o rtsched rtsched.c -lrt
 */

#include <stdio.h>
#include <sys/types.h>
#include <errno.h>

extern int errno;

main(argc,argv)
int argc;
char **argv;
{
    int pid;
    int tmp_rtprio, set_rtprio, old_rtprio, flag_rtprio;
    int tmp_targetcpumask, set_targetcpumask, old_cpumask, flag_targetcpumask;

    if( argc <= 1 )
        Usage(argv[0]);
    else
    for(;*++argv;) {
        /*
         * Get the Real Time priority
         */
        if( sscanf(*argv,"-rtprio=%d",&tmp_rtprio) == 1 )  {
            set_rtprio = tmp_rtprio;
            flag_rtprio = 1;
        }
        /*
         * Get the cpu mask
         */
        else
        if( sscanf(*argv,"-cpu=0x%x",&tmp_targetcpumask) == 1 ) {
            set_targetcpumask = tmp_targetcpumask;
            flag_targetcpumask = 1;
        }
        /*
         * Must be an argument.  Do the requested operations on the process id.
         */
        else
        if( sscanf(*argv,"%d",&pid) == 1 ) {
            if( flag_rtprio )
            {
                if( (old_rtprio = setrealpriority(pid,set_rtprio)) < 0) {
                    fprintf (stderr, "Couldn't set RT priority:  pid=%d  ",
                        pid);
                    perror ("");
                }
                else {
                    printf("Pid %d:  Old prio=%d  New prio=%d\n",
                        pid, old_rtprio, set_rtprio);
                }
            }
            if( flag_targetcpumask ) {
                if ((old_cpumask = settargetcpumask(pid,set_targetcpumask)) < 0) {
                    fprintf (stderr, "Couldn't set cpu masK:  pid=%d  ",
                        pid);
                    perror ("");
                }
                else {
                    printf("Pid %d:  Old cpu mask=%d  New cpu mask=%d\n",
                        pid, old_cpumask, set_targetcpumask);
                }
            }
        }
        else {
            fprintf (stderr, "Unrecognized argument:  '%s'\n", *argv);
            Usage (argv[0]);
        }
    }
}

Usage(name)
char *name;
{
    fprintf(stderr,
        "Usage: %s [-rtprio[=val]] [-cpu[=val]] [-active] pid...\n", name);
}
```

A.2 Cyclic Scheduling

Many of the applications of the cyclic scheduler will be for sets of processes. The example in this section is a template for a slave process that will be run by a master process.

A.2.1 Summary of the Template Example

The template example begins on the next page.

The particular process in this example sets up a signal handler for the SIGINT signal, which tells the process to restart from the beginning. The SIGINT signal can be used to restart a set of processes by sending a signal to all of them.

After the signal handler is set up, the actual cyclic scheduling parameters are set up. The process in the example specifies a cycle with 60 frames and sets every third frame. If the process overruns a frame, it will continue with a clean start on the next set frame.

To ensure good response, the process is locked into memory and real priority of zero is requested. The entire process (text and data) is locked. The restarting place is then set with the **setjmp** call.

At this point, the initialization procedure is complete. The process requests cyclic scheduling with the **cycsetdata** call and waits for the next **cycsync**, which the master process is expected to issue. Some form of communication should exist between the master process and its slaves. Shared memory might be a good vehicle for this; see *shmctl*(2RT), *shmget*(2RT), and *shmop*(2RT) for a discussion of shared memory. When the master process notices that all slaves have completed their initialization, it issues a **cycsync**, and the slave starts cyclic execution.

The slave process will run in the first frame after the **cycsync**. When it is finished, it suspends with the **cycsuspend** system call. The process must be careful to note the return code from the system call. **cycsuspend** returns an error and sets **errno** to EINTR if the system call is interrupted. This is not a fail-safe procedure since the system call might return with EINTR in the frame in which the process should run. This problem can be avoided by blocking or ignoring signals.

A.2.2 Slave Process Template

```
#include <sys/cyclic.h>
#include <sys/lock.h>
#include <errno.h>
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>

#define FR_LEN  60               /* I want to be 60 frames long */

extern int errno;

void    restart();

jmp_buf         Restart;         /* The restart environment */

/*
 *   M A I N
 */
main (argc, argv)
int     argc;
char    *argv[];
{
        int             i;
        int             ret;
        Tcyclicdata     cd;
        int             frame;


        /*
         * Set up the signal handler for SIGINT.
         */
        signal (SIGINT, restart);

        /*
         * Initialize the cyclic data structure.  The cycle length is
         * FR_LEN frames.  The process expects to run every third frame.
         * If the process is late in suspending, that's too bad.  Don't
         * look back on frames and try to reschedule immediately.
         */
        for (i=0; i<FR_LEN; i++)
                cd.cycle [i] = !(i % 3);
        cd.cycle_length = FR_LEN;
        cd.frames_lookback = 0;

        /*
         * Lock the process into memory.
         */
        if (plock (PROCLOCK) < 0)
        {
                printf ("plock failed,  errno=%d\n", errno);
                exit (1);
        }

        /*
         * Set the real priority to the highest priority possible.
         * This is an important process.
         */
        if (setrealpriority (0, 0) < 0)
        {
                printf ("setrealpriority failed,  errno=%d\n", errno);
                exit (1);
        }

        /*
         * This is the place to return to when the process restarts.
         */
        setjmp (Restart);

        /*
         * Set the cyclic scheduling parameters for the process.
         * Wait for the next cyclic sync.
         */
        if ((ret = cycsetdata (&cd, 1)) < 0)
        {
                printf ("setdata failed,  errno=%d\n", errno);
                exit (1);
        }

        for (;;)
        {
                /*
                 * This is where the useful work of the process gets done.
                 * Every time the process resumes, it will come here and
                 * execute whatever needs to be done.
                 */

                /*
                 * Notice that the return code is checked.  If cycsuspend()
                 * returns with an error and errno is EINTR, then we have
                 * received a signal
                 */
                while (((frame = cycsuspend ()) < 0) && (errno == EINTR))
                {
                        printf ("interrupted\n");
                }
        }
}


/*
 * This is the signal catcher for SIGINT.
 * Just restart the process from the beginning.
 */
void
restart ()
{
        longjmp (Restart, 1);
}
```

A.3 Timer Services

This section contains four examples in which the enhanced timer services are used. Two are in FORTRAN, and two are in C.

A.3.1 FORTRAN Test Programs

The first FORTRAN test program, **hscbad**, tests the **sethsctimer** system call for bad input values. The second, **hscreadtimer**, reads from the high-speed clock and prints out the information as fast as possible.

Example 1

```
c
c       Compile with:  fort -o hscbad hscbad.f -lrtf -lrt
c
        program hscbad

        parameter (PROCLOCK=1)
        parameter (EINVAL=22)
        parameter (HSCCANCEL=0, HSCABSOLUTE=3, HSCCYCLE=2)

        integer ret
        integer failed
        integer hval1(2), hval2(2)
        integer hsc1secs, hsc1ns, hsc2secs, hsc2ns
        integer sethsctimer

        equivalence (hval1(1), hsc1secs)
        equivalence (hval1(2), hsc1ns)
        equivalence (hval2(1), hsc2secs)
        equivalence (hval2(2), hsc2ns)

        failed = 0

        ret = plock (PROCLOCK)
        if (ret .ne. 0) then
                        write (6,*) 'plock failed,  ret = ', ret
                        call exit (1)
        endif

        ret = setrealpriority (0, 0)
        if (ret .ne. 0) then
                        write (6,*) 'setrealpriority failed,  ret = ', ret
                        call exit (1)
        endif

c
c       Give sethsctimer a bad opcode.
c
        ret = sethsctimer (HSCCANCEL-1, hval1, hval2)
        if (ret .ne. -EINVAL) then
                        write (6,*) 'bad sethsctimer arg test 1 failed,',
       +                                        '  ret = ', ret
                        failed = failed + 1
        endif

        ret = sethsctimer (HSCABSOLUTE+1, hval1, hval2)
        if (ret .ne. -EINVAL) then
```

```
                     write (6,*) 'bad sethsctimer arg test 2 failed,',
     +                                  '   ret = ', ret
                     failed = failed + 1
      endif

c
c     Give sethsctimer invalid data for the second argument.
c     All combinations.
c
      hsclsecs = -1
      hsclns = 0
      ret = sethsctimer (HSCCYCLE, hval1, hval2)
      if (ret .ne. -EINVAL) then
                     write (6,*) 'sethsctimer invalid data test 1 failed,',
     +                                  '   ret = ', ret
                     failed = failed + 1
      endif

      hsclsecs = 0
      hsclns = -1
      ret = sethsctimer (HSCCYCLE, hval1, hval2)
      if (ret .ne. -EINVAL) then
                     write (6,*) 'sethsctimer invalid data test 2 failed,',
     +                                  '   ret = ', ret
                     failed = failed + 1
      endif

      hsclsecs = -1
      hsclns = -1
      ret = sethsctimer (HSCCYCLE, hval1, hval2)
      if (ret .ne. -EINVAL) then
                     write (6,*) 'sethsctimer invalid data test 3 failed,',
     +                                  '   ret = ', ret
                     failed = failed + 1
      endif

      hsclsecs = 2147483647
      hsclns = 0
      ret = sethsctimer (HSCCYCLE, hval1, hval2)
      if (ret .ne. -EINVAL) then
                     write (6,*) 'sethsctimer invalid data test 4 failed,',
     +                                  '   ret = ', ret
                     failed = failed + 1
      endif

      hsclsecs = 0
      hsclns = 1000000001
      ret = sethsctimer (HSCCYCLE, hval1, hval2)
      if (ret .ne. -EINVAL) then
                     write (6,*) 'sethsctimer invalid data test 5 failed,',
     +                                  '   ret = ', ret
                     failed = failed + 1
      endif

      if (failed .eq. 0) then
                     write (6,*) 'hscbad test passed'
      else
                     write (6,*) 'hscbad test had', failed, ' errors'
      endif

      stop
      end
```

Example 2

```
C
C     Compile with:  fort -o hscreadtimer hscreadtimer.f -lrtf -lrt
C
      program hscreadtimer

      parameter (PROCLOCK=1)

      integer ret
      integer hval(2)
      integer hscsecs, hscns
      integer gethscvalue

      equivalence (hval(1), hscsecs)
      equivalence (hval(2), hscns)


      ret = plock (PROCLOCK)
      if (ret .ne. 0) then
            write (6,*) 'plock failed,  ret = ', ret
            call exit (1)
      endif

      ret = setrealpriority (0, 0)
      if (ret .ne. 0) then
            write (6,*) 'setrealpriority failed,  ret = ', ret
            call exit (1)
      endif

C
C     Forever loop
C
  100 continue
      ret = gethscvalue (hval)
      if (ret .ne. 0) then
            write (6,*) 'gethscvalue failed,', '  ret = ', ret
            call exit(1)
      endif
      write (6,*) 'secs=', hscsecs, ' nanosecs=', hscns
      go to 100

      stop
      end
```

A.3.2 C Test Programs

The first C example demonstrates how to measure the interval between two events. The **hscval_before** structure contains the value of the timer before the event. **hscval_after** contains the value of the timer after the event. The nanosecond value is guaranteed to be smaller than 1,000,000,000.

The second example shows how to set a timer and wait for it to expire.

Example 1

```
/*
 * Compile with:  cc -o interval interval.c -lrt
 */

#include <sys/rt_time.h>
#include <sys/lock.h>
#include <errno.h>
#include <stdio.h>

extern int errno;

/*
 *   M A I N
 */
main()
{
        struct hscval   hscval_before;  /* time before work */
        struct hscval   hscval_after;   /* time after work */

        /*
         * Lock memory and set us up at real
         * time priority 0 (highest).
         */
        if (plock (PROCLOCK) < 0)
        {
                printf ("plock failed,  errno=%d\n", errno);
                exit (1);
        }
        if (setrealpriority (0, 0) < 0)
        {
                printf ("setrealpriority failed,  errno=%d\n", errno);
                exit (1);
        }

        gethscvalue (&hscval_before);

        /*
         * Do some work that you want to measure.
         */

        gethscvalue (&hscval_after);
}
```

Example 2

```
/*
 * Compile with:  cc -o timer timer.c -lrt
 */

#include <sys/rt_time.h>
#include <sys/lock.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>

int alrm();

extern int errno;

int             Timerexpired;           /* If true, the timer has expired */
struct hscval   hscval;                 /* Value to program timer */

/*
 *   M A I N
 */
main()
{
        /*
         * Set up the signal handler for the timer signal.
         */
        signal (SIGALRM, alrm);

        /*
         * Plock memory and request real-time priority.
         */
        if (plock (PROCLOCK) < 0)
        {
                printf ("plock failed,  errno=%d\n", errno);
                exit (1);
        }
        if (setrealpriority (0, 0) < 0)
        {
                printf ("setrealpriority failed,  errno=%d\n", errno);
                exit (1);
        }
        /*
         * Set the timer to expire 8.325 milliseconds from now.
         * Note that the second argument to sethsctimer is 0.
         * We don't care about the previous timer,
         * whether it was set or not.
         */
        hscval . hsc_seconds = 0;
        hscval . hsc_nanosecs = 8325000;
        Timerexpired = 0;
        if (sethsctimer (HSC_RELATIVE, &hscval, 0) < 0)
        {
                printf ("error: errno=%d\n", errno);
                exit (1);
        }

        /*
         * If timer has not already expired, then wait for it.
         */
        if (!Timerexpired)
```

```
                sigpause (0);
}

/*
 * Signal handler for the SIGALRM signal.  All we do is set
 * the Timerexpired value to true so it can be checked when
 * the timer was started.
 */
alrm ()
{
        Timerexpired = 1;
}
```

A.4 Shared Memory

The following C program uses shared memory and timer services. This program tests that multiple processes scheduled to run at the same absolute time will run in the proper order.

```
/*
 * Process Order Test
 *      This program tests the orderly execution of
 *      multiple processes scheduled to run at the same absolute time.
 *      When each process runs, it determines if it is running
 *      in sequence by looking in shared memory for the
 *      (test specific) process number of the last process to
 *      run. It then stores its own process number in shared memory
 *      and schedules itself to run on the next absolute time increment.
 */

#include <signal.h>
#include <machine/cpu.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include <setjmp.h>
#include <stdio.h>
#include <errno.h>
extern int      errno;

#include <sys/rt_time.h>

#define MAXDEL          30
#define MAXNUMPROCS     10
#define TESTTIMEOUT     (10*60)

int             TDelay = 3;
struct hscval   hscval;
int             ProcList[MAXNUMPROCS];  /* PIDs of child processes */
int             NumProc = 5;            /* Number of procs actually running */
int             ProcNum;                /* Process number in the set
                                         * of child processes: a loop
                                         * counter in main, different
                                         * per-process in the children */

/* Shared memory for last ran process number */
struct IPCbuff
{
    int         LastProcNum;            /* last process that ran (used
                                         * to check sequencing) */
};
int             ProcShmid;              /* Shared memory identifier */
struct IPCbuff *Procbuff;               /* Shared memory address */

/* Shared memory for test information communication */
struct GenericBuff
{
    int         failedtests;
    int         successfulpasses;
};
int             FTShmid;                /* Generic Shared mem id     */
struct GenericBuff *Generic;            /* Generic Shared mem addr.  */
```

```
/* test functions */
void            enditall ();            /* Kills all the child proc. */
void            CheckProcessOrdering ();
void            TestTimeout ();
void            alarm_handler ();

/* shared memory functions */
int             shmget ();
char           *shmat ();
void            shmgetat ();

int Done = 0;

main (argc, argv)
    int             argc;
    char          **argv;
{
    int         ret;            /* return code for sys calls */
    int         i;              /* counter                   */

    HandleArguments (argc, argv);

    /* Ends test at 10 mins */
    signal (SIGALRM, TestTimeout);
    alarm (TESTTIMEOUT);

    /* cleanup if signal to interrupt test */
    signal (SIGTERM, enditall);

    /*
     * Attach to shared memory for Generic
     * Note that shmgetat combines shmget and shmat
     */
    shmgetat (sizeof (*Generic), &FTShmid, &Generic);
    Generic->failedtests = 0;
    Generic->successfulpasses = 0;

    /* Attach to shared memory for Procbuff */
    shmgetat (sizeof (*Procbuff), &ProcShmid, &Procbuff);
    Procbuff->LastProcNum = NumProc - 1;

    /* Get starting absolute time */
    if ((ret = gethscvalue( &hscval)) == -1)
    {
        printf("Gethscvalue failed errno=%d\n",errno);
        Generic->failedtests++;
        proccleanup();
    }

    /*
     * Create child processes
     * Children inherit the data space of the parent process,
     * so they get a copy of 'hscval' above
     * They also inherit the shared memory segments, so
     * the children do not need to do individual shmat's.
     */
    for (ProcNum = 0; ProcNum < NumProc; ProcNum++)
    {
        if ((ProcList[ProcNum] = fork ()) == 0)
        {
            Child ();
            proccleanup();
        }
    }

    /* Wait for completion of all children */
    for (i = 0; i < NumProc; i++)
    {
        wait (0);
    }

    printf("Parent exiting\n");
    printf ("Errors so far: %d\n", Generic->failedtests);
    printf ("Successful passes so far: %d\n", Generic->successfulpasses);
    exit (0);
}

/*
 * Child
 *      Child process code for test.
 * Uses
 *      extern ProcNum (per-process) - process number in test set.
 *
 */
Child ()
{
    int             i;
    int             ret;
    struct hscval   hscval2;
    cpumask_t       mask;

    /*
     * Child initialization code. Set up ordering
     * amongst processes (eg. by using RT priority).
     */
    signal (SIGALRM, alarm_handler);

    /*
     * In this test, done by setting RT priority, everybody waking up
     * at same time. RT priorities start at 1, not 0, so that we can
     * regain control of machine.
     */
    if ((ret = setrealpriority (0, ProcNum + 1)) < 0)
    {
        printf (" Setrealpriority failed, errno=%d\n", errno);
        Generic->failedtests++;
        proccleanup();
    }
    if ((mask = settargetcpumask(0, P_CPUMASK(0))) == -1)
    {
        printf("targetcpu failed, errno=%d\n",errno);
        Generic->failedtests++;
        proccleanup();
    }

    /* Set new absolute timer and pause until it expires*/
    for (i = 0; i < 100; i++)
    {
        hscval.hsc_seconds = hscval.hsc_seconds + TDelay;
        if ((ret = sethsctimer(HSC_ABSOLUTE, &hscval, &hscval2)) == -1)
        {
            Generic->failedtests++;
            printf("Sethsctimer failed, errno=%d\n",errno);
            proccleanup();
            }
            pause ();
        }
        Done = 1;
        proccleanup();

}

/*
 * alarm_handler - timer has gone off
 */
void
alarm_handler ()
{
            CheckProcessOrdering();
}

/*
 * enditall
 *          cleanup routine
 *          set LastProcNum to -1.
 *          Other processes will exit when they see this.
 */
void
enditall ()
{
    int             i;

    Procbuff->LastProcNum = -1;

    /* remove shared memory (also detaches) */
    shmctl (ProcShmid, IPC_RMID, 0);
    shmctl (FTShmid, IPC_RMID, 0);

    exit(0);
}

/*
 *  CheckProcessOrdering
 *      Check that the test processes execute in order.
 *      Order is determined by the test, usually using RT prio in some form.
 *      Ordering is strictly linear-circular.
 *  Uses
 *      extern ProcNum (per process) - process number in test set
 */
void
CheckProcessOrdering ()
{
    int lastproc = Procbuff->LastProcNum;

    if ( lastproc == -1)        /* parent says go away */
    {
        exit(2);
    }
    else        /* normal case */
    {
        if ( ((lastproc + 1) % NumProc)  != ProcNum)
        {
            Generic->failedtests++;
            printf (" Illegal Process sequence\n");
            printf (" LastProcNum = %d, ProcNum=%d, Successful=%d\n",
            lastproc, ProcNum, Generic->successfulpasses);
```

```
                proccleanup();
                /* NOT REACHED */
            }
            Generic->successfulpasses++;
            Procbuff->LastProcNum = ProcNum; /* indicate this proc ran */
        }
}


/*
 * Wrapper around combined calls to shmget and shmat,
 * printing and exiting on error.
 */
void
shmgetat (size, idp, basep)
    int             size;
    int            *idp;
    char          **basep;
{
    if ((*idp = shmget (IPC_PRIVATE, size, IPC_CREAT | 0660)) < 0)
    {
        printf ("shmget failed,  errno=%d\n", errno);
        proccleanup();
    }
    if ((*basep = shmat (*idp, 0, SHM_LOCK)) < 0)
    {
        printf ("shmget failed,  errno=%d\n", errno);
        proccleanup();
    }
}

/*
 * HandleArguments -- Is where test specific argument
 *          handling code goes.
 *
 */
HandleArguments (argc, argv)
    int             argc;
    char          **argv;
{
    int             arg;

    for (; *++argv;)
    {
        /* Allows passing in number processes to create */
        if (sscanf (*argv, "-np=%d", &arg) == 1)
        {
            if (1 <= arg && arg <= MAXNUMPROCS)
            {
                NumProc = arg;
            } else
            {
                printf ("-np=<NumProc> must be within [1,%d]\n", MAXNUMPROCS);
                proccleanup();
            }
        }
        else
        {
            printf ("Unrecognized argument %s\n", *argv);
            proccleanup();
        }
    }
```

```
}

/*
 * TestTimeout
 *        Stops execution after TESTTIMEOUT minutes
 */
void
TestTimeout ()
{
    printf ("Test timed out\n");
    printf ("Errors so far: %d\n", Generic->failedtests);
    printf ("Successful passes so far: %d\n", Generic->successfulpasses);
    proccleanup();
}


/*
 * proccleanup -- cleans up the child processes
 *
 */
proccleanup()
{
    int i;

    /* Kill all child processes.  A -1 in LastProcNum tells the other
     * processes that they should just exit. */
    Procbuff->LastProcNum = -1;

    exit(1);
}
```

A.5 Direct File System

The following FORTRAN program uses the direct file system calls. This test program simply exercises all direct file system calls in the real-time FORTRAN library.

```
c
c      Compile with:  fort -o direct direct.f -lrtf -lrt
c
       program dftest

       integer PROCLOCK, UNLOCK
       parameter (PROCLOCK=1,UNLOCK=0)
       integer plock,dfcreate,dfopen,dfwrite,dfclose, dfdelete,dfextend
       integer ret,fd,error
       integer eventctrs(3)
       integer wbuff(100)
       character*60 pathname
       data pathname/'/disk0a/file'/

       error = 0
       ret = plock (PROCLOCK)
       if (ret .ne. 0) then
           write (6,*) 'plock failed, ret=', ret
           call exit (1)
       endif

       ret = dfcreate (pathname,1,1,-1,0)
       if (ret .lt. 0) then
           write (6,*) 'dfcreate failed, ret=', ret
           error = 1
           goto 600
       endif

       ret = dfextend (pathname,1,0)
       if (ret .lt. 0) then
           write (6,*) 'dfextend failed, ret=', ret
           error = 1
           goto 500
       endif

       fd = dfopen (pathname,0,eventctrs,0)
       if (fd .lt. 0) then
           write (6,*) 'dfopen failed, ret=', ret
           error = 1
           goto 400
       endif

       ret = dfwrite (fd,wbuff,30)
       if (ret .lt. 0) then
           write (6,*) 'dfwrite failed, ret=', ret
           error = 1
           goto 400
       endif

400    ret = dfclose (fd)
       if (ret .lt. 0) then
           write (6,*) 'dfclose failed, ret=', ret
           error = 1
       endif

500    ret = dfdelete (pathname)
```

```
      if (ret .lt. 0) then
          write (6,*) 'dfdelete failed, ret=', ret
          error = 1
          goto 600
      endif

600   ret = plock (UNLOCK)
      if (ret .ne. 0) then
          write (6,*) 'plock failed, ret=', ret
          error = 1
      endif

      if (error .eq. 0) then
          write (6,*) 'test passes'
      endif

      stop
      end
```

A.6 Direct I/O

The following FORTRAN program uses the direct I/O facility. This example is of special interest because it involves calls to C code. The FORTRAN program is **diotest.** The C utilities follow.

If you run this program, be sure that you run it on a disk that doesn't contain important data. This test will write the first 1024 words on the disk at address 0x800. Note that the disk geometry given to the **createiocds** routine changes for various types of disk drives.

To compile the program, execute the following commands:

```
% cc -c createIOCD.c
% fort -o diotest diotest.f createIOCD.o -lrtf -lrt
```

A.6.1 DIO FORTRAN Program

```
program diotest

integer PROCLOCK, UNLOCK
parameter (PROCLOCK=1,UNLOCK=0)
parameter (READCMD=0,WRITECMD=1)
parameter (NWORDS=1024)
integer plock,dioconnect,dioconvert,dionotify,diorelease
integer dioreserve,diosiolog
integer notify(2),statbf(2),conID
integer ret,error
integer wbuff(NWORDS*4),rbuff(NWORDS*4)
integer wlogiocl(4),rlogiocl(4),wphysiocl(6),rphysiocl(6)
integer seekdata(1)
integer diskaddr
data    diskaddr/x'800'/

error = 0
ret = plock (PROCLOCK)
if (ret .ne. 0) then
    write (6,*) 'plock failed, ret=', ret
    call exit (1)
endif

ret = dioreserve (diskaddr)
if (ret .lt. 0) then
    write (6,*) 'dioreserve failed, ret=', ret
    error = 1
    goto 600
endif

conID = dioconnect (diskaddr,0,4,notify)
if (conID .lt. 0) then
    write (6,*) 'dioconnect failed, ret=', conID
    error = 1
    goto 500
endif

ret = dionotify (conID,29,0,0)
if (ret .lt. 0) then
    write (6,*) 'dionotify failed, ret=', ret
```

```
          error = 1
          goto 400
      endif

      call createiocds(WRITECMD,0,NWORDS*4,16,5,seekdata,wbuff,wlogiocl)

      ret = dioconvert (conID,wlogiocl,16,wphysiocl,24)
      if (ret .lt. 0) then
          write (6,*) 'dioconvert failed, ret=', ret
          error = 1
          goto 400
      endif

c
c     Fill up wbuff with a known test pattern.
c
      do 100 i = 1,NWORDS
                wbuff(i) = i
  100 continue

c
c     Write the pattern to the disk.
c
      ret = diosiophys (conID,wphysiocl,statbf,0,10)
      if (ret .lt. 0) then
          write (6,*) 'diosiophys failed, ret=', ret
          error = 1
          goto 400
      endif

      call createiocds (READCMD,0,NWORDS*4,16,5,seekdata,rbuff,rlogiocl)

c
c     Read the pattern from the disk into rbuff.
c
      ret = diosiolog (conID,rlogiocl,16,rphysiocl,24,statbf,0,10)
      if (ret .lt. 0) then
          write (6,*) 'diosiolog failed, ret=', ret
          error = 1
          goto 400
      endif

c
c     Buzz loop until the I/O completes.   Instead of doing this, we
c     could have requested wait I/O in the dionotify.
c     Note:  no timeout, so if the I/O never completes, the program
c     never leaves here.
c
  150 if (notify(1) .ne. notify(2)) goto 150

c
c     Test the pattern that was written with the pattern that was read.
c
      do 200 i = 1,NWORDS
          if (wbuff(i) .ne. rbuff(i)) then
              write (6,*) 'bad read or write; buffer corrupted'
              error = 1
              goto 400
          endif
  200 continue

  400 ret = diodisconnect (conID)
```

```
      if (ret .lt. 0) then
          write (6,*) 'diodisconnect failed, ret=', ret
          error = 1
      endif

500   ret = diorelease (diskaddr)
      if (ret .lt. 0) then
          write (6,*) 'diorelease failed, ret=', ret
          error = 1
      endif

600   ret = plock (UNLOCK)
      if (ret .ne. 0) then
          write (6,*) 'plock failed, ret=', ret
          error = 1
      endif

      if (error .eq. 0) then
          write (6,*) 'test passes'
      endif

      stop
      end
```

A.6.2 C Utilities for DIO FORTRAN Program

```
/*
 *  dio test utility routines...
 */
#include "sys/ioctl.h"
#include "selio/iocmd.h"

#define READ_CMD        0
#define WRITE_CMD       1

/*                                                                    */
/*   Function:                                                        */
/*       This procedure creates an IOCD pair for the Gould Disk       */
/*       Processor to seek and read/write on the disk.  The 2 IOCDs   */
/*       are command chained.  The address fields in the IOCD are all */
/*       logical addresses.  The output of this procedure is suitable */
/*       to pass to the Direct I/O interface as a LOGICAL IOCL.  This */
/*       routine may be called outside the Direct Files environment,  */
/*       but is heavily dependent upon Gould CONCEPT/32 Class F hardware. */
/*                                                                    */


createiocds_ (ReadWrite, Block, Size, Trksize, Cylsize, Seek, Addr, Dest)

int *ReadWrite;                /* A flag indicating a read or a write cmd */
int *Block;                    /* The disk block number to start I/O at   */
int *Size;                     /* The number of bytes to read/write       */
int *Trksize;                  /* # of blocks per track                   */
int *Cylsize;                  /* # of tracks per cylinder                */
TSeekWord *Seek;
char *Addr;                    /* The logical address of where the data is
                                  to be read/written to/from              */
TIOCD *Dest;                   /* A pointer to where the 2 IOCD structures
                                  are to be stored                        */

{
        register int rem;      /* remainder after division */


/* compute seek data word... */


        Seek->s_SeekCyl = *Block / (*Cylsize * *Trksize);
        rem = *Block % (*Cylsize * *Trksize);
        Seek->s_SeekTrack = rem / *Trksize;
        Seek->s_SeekSector = rem % *Trksize;


/* format seek IOCD... */


        Dest->i_IOCmd = IO_SEEK;
        Dest->i_Address = (unsigned) Seek;       /* put addr of seek word */
        Dest->i_IOflags = IO_CMD_CHAIN;
        Dest->i_XferCount = 4;                   /* 4 bytes of seek data */

        Dest++;                                  /* increment to next IOCD */


/* format read or write IOCD... */
```

```
        if (*ReadWrite == READ_CMD)
        {
                Dest->i_IOCmd = IO_READ;        /* read data command */
        }
        else
        {
                Dest->i_IOCmd = IO_WRITE;       /* write data command */
        }

        Dest->i_Address = (unsigned) Addr;
        Dest->i_IOflags = 0;
        Dest->i_XferCount = *Size;

        return;
}
```

A.7 Connected Interrupts

The following examples are explained in Section 10.2 of Chapter 10, "Connected Interrupts." The first example shows a simple indirectly connected interrupt. The second example shows a simple directly connected interrupt.

A.7.1 Example Using Indirectly Connected Interrupt

```
/*
 * Compile with:  cc -o indircon indircon.c -lrt
 */

#include <stdio.h>
#include <signal.h>
#include <sys/rt_ci.h>
#include <sys/lock.h>
#include <sys/errno.h>
#include <sel/machparam.h>

extern int      errno;


/*
 * These define the device and interrupt level of the connection.
 */
#define CHAN    0x7f
#define SUBCHAN 0x0d
#define CLASS   0x03
#define PRI     0x50

int     sig_handler ();
int     cleanup ();

int     Nsigs = 0;          /* Number of signals received */
int     Key;                /* Identifying key for the connection */



/*
 * main         This program makes a direct connection to an interrupt
 *              level and counts the number of interrupts it receives.
 *
 *              The program must be run as superuser.
 */

main(argc, argv)
int argc;
char **argv;
{
        int     ret;    /* Return value of system calls */

        /*
         * Set up a signal handler so that the process will clean up and
         * exit when it is signaled.
         */
        if (signal (SIGINT, cleanup) < 0)
        {
                fprintf (stderr, "signal (SIGINT) failed errno=%d\n", errno);
                exit (1);
        }

        /*
         * Set up the signal handler that will be called when an interrupt
         * is received.
         */
        if (signal (SIGUSR1, sig_handler) < 0)
        {
                fprintf (stderr, "signal (SIGUSR1) failed errno=%d\n", errno);
                exit (1);
        }

        Nsigs = 0;

        /*
         * Install the indirectly connected interrupt handler.
         */
        if ((ret = plant (&Key, CHAN, SUBCHAN, CLASS, PRI, CI_INDIRECT)) < 0)
        {
                fprintf (stderr, "plant failed errno=%d  intr level=0x%lx\n",
                                        errno, PRI);
                exit (1);
        }
        if ((ret = graft (Key, SIGUSR1, 0, CI_INDIRECT)) < 0)
        {
                uproot (Key);
                fprintf (stderr, "graft failed errno=%d  intr level=0x%lx\n",
                                        errno, Key);
                exit (1);
        }

        /*
         * Enable the interrupt level.
         */
        ei (Key);

        /*
         * Wait forever.  The work of this program is done in the signal
         * handler.
         */
        for (;;)
                pause ();
}

/*
 * sig_handler -        This function is the signal handler.
 */
sig_handler ()
{
        Nsigs++;

        /*
         * The interrupt level for an indirectly connected interrupt
             * is deactivated by the kernel.
         */
}


/*
 * cleanup        - This function cleans up the process when terminated.
 */
cleanup ()
```

```
{
        prune (Key);
        uproot (Key);
        exit (0);
}
```

A.7.2 Example Using Directly Connected Interrupt

```
/*
 * Compile with:  cc -o dircon dircon.c -lrt
 */
#include <stdio.h>
#include <signal.h>
#include <sys/rt_ci.h>
#include <sys/lock.h>
#include <sys/types.h>

extern int      errno;


#define CHAN    0x7f
#define SUBCHAN 0x0d
#define CLASS   0x03
#define PRI     0x50

int     intr_handler ();
int     cleanup ();

int     Stack [500];            /* Interrupts run on this stack */
int     Nints = 0;              /* Number of interrupts taken so far */
int     Key;                    /* Key to identify connection */




/*
 * main         This program makes a direct connection to an interrupt
 *              level and counts the number of interrupts it receives.
 *
 *              The program must be run as superuser.
 */

main(argc, argv)
int argc;
char **argv;
{
        int     ret;

        /*
         * Lock the process into memory.
         */
        if (plock (PROCLOCK) < 0)
        {
                fprintf (stderr, "plock failed:  errno=%d\n", errno);
                exit (1);
        }

        /*
         * Target the process to the CPU.
         */
        if (settargetcpumask (getpid (), P_CPUMASK (P_CPU)) < 0)
        {
                fprintf (stderr, "settargetcpumask failed:  ");
                fprintf (stderr, "errno=%d  pid=0x%lx  cpu mask=0x%lx\n",
                                        errno, getpid (), P_CPUMASK (P_CPU));
                exit (1);
        }

        /*
         * Set up a signal handler so that the process will clean up and
         * exit when it is signaled.
         */
        if (signal (SIGINT, cleanup) < 0)
        {
                fprintf (stderr, "signal failed:  errno=%d\n", errno);
                exit (1);
        }

        Nints = 0;

        /*
         * Install the directly connected interrupt handler.
         */
        if ((ret = plant (&Key, CHAN, SUBCHAN, CLASS, PRI, CI_DIRECT)) < 0)
        {
                fprintf (stderr, "plant failed errno=%d  intr level=0x%lx\n",
                                        errno, PRI);
                exit (1);
        }
        if ((ret = graft (Key, &Stack[499], intr_handler, CI_DIRECT)) < 0)
        {
                uproot (Key);
                fprintf (stderr, "graft failed errno=%d  Key=0x%lx\n",
                                        errno, Key);
                exit (1);
        }


        /*
         * Enable the interrupt level.
         */
        ei (PRI);


        /*
         * Wait forever.  The work of this program is done in the interrupt
         * routine.
         */
        for (;;)
                pause ();
}


/*
 *
 * intr_handler -       This function is the interrupt handler.  It is called
 *                      when an interrupt is received.
 *
 */
intr_handler (level, status)
register int    level;
register int    status;
```

```
{
        Nints++;

        /*
         * Deactivate the interrupt.
         */
        hwprivdai (level);
}


/*
 *
 * cleanup -            This function cleans up when the process exits.
 *                      It is expected that the process will exit after
 *                      receiving a SIGINT.
 *
 */
cleanup ()
{
        prune (Key);
        uproot (Key);
        exit (0);
}
```

A.8 Suspend and Resume

A.8.1 Ping-Pong

In this example the parent process creates one child process. The child process immediately suspends itself. The parent process then resumes the child and suspends itself. Upon being resumed, the child process resumes the parent and suspends itself. This causes a ping-pong effect between the two processes.

```
/*
 * ping-pong - This program demonstrates how two processes can
 *       suspend and resume each other in sequence (ping-pong
 *       back and forth).
 *
 *       Compile with:  cc -o pingpong pingpong.c -lrt
 */

/* header files */
#include <errno.h>
#include <stdio.h>
#include <sys/rt_suspres.h>

/* externally declared variables */
extern int errno;

/* constant declarations */
#define ITERATIONS      5

/* Global Variables */
int child1;             /* process list containing child pid */
int parent;             /* process list containing parent pid */


main()
{

        /* create process list which contains parent pid */
        parent = getpid();

        /* fork off child process and create
         * process list which contains child pid
         */
        if((child1 = fork()) == 0)
        {
                /* suspend child process
                 * in order to achieve proper sequencing
                 * do not care if suspend interrupted by signal
                 * if suspend fails test will fail
                 */
                if (suspend() == -1)
                {
                        printf("suspend failed errno = %d\n",errno);
                        exit(-1);
                }
                Child1();
        }

        /* wait for a while to assure child process
         * is suspended  before starting ping pong effect
         * The 2 passed to sleep was chosen at random
         */
        sleep(2);
        Parent();
}

/*
 * Child1 --  resumes the parent process and immediately suspends
 *            itself.  Repeats this for ITERATIONS times.
 */
Child1()
{
        int i = 0;                  /* counter variable */

        while (i++ < (ITERATIONS -1))
        {
                /*  resume parent process and immediately suspend */
                if (resume(SR_SUSPEND,parent) == -1)
                {
                        printf("resume in Child 1 failed i=%d\n",i);
                        printf(" errno = %d\n",errno);
                        exit(-1);
                }
                printf("Child 1 loop = %d\n",i);
        }
        /* resume last parent process but do not suspend
         * because parent will never resume this process
         */
        if (resume(0,parent) == -1)
        {
                printf("resume in Child 1 failed i=%d\n",i);
                printf(" errno = %d\n",errno);
                exit(-1);
        }
        printf("exiting child 1\n");
        exit(0);
}

/*
 * Parent --  resumes child process and immediately suspends
 *        itself repeats this for ITERATIONS
 */
Parent()
{
        int i = 0;                  /* counter variable */

        while (i++ < ITERATIONS)
        {
               /* resume child process and then suspend */
               if (resume(SR_SUSPEND,child1) == -1)
               {
                        printf("resume in Parent failed i=%d\n",i);
                        printf(" errno = %d\n",errno);
                        exit(-1);
               }
               printf("Parent loop = %d\n",i);
        }
        wait(0);
        printf("exiting Parent\n");
        exit(0);
}
```

A.8.2 Order

In this example, the parent process creates several child processes. Each child process sets its real-time priority to be one greater than the last child that was forked. Each child process then suspends itself and waits to be resumed. After all of the child processes are suspended, the parent process resumes the list of child processes. The child processes should then execute according to their real-time priorities.

```
/*
 *   order -- This program sets up several child processes, assigns
 *            them each a real-time priority, and then suspends them.
 *            The parent process simply resumes all of the children.
 *            Upon being resumed the child processes should execute
 *            according to their real-time priorities.
 *
 *   Compile with:  cc -o order order.c -lrt
 */

/* HEADERS */
#include <signal.h>
#include <machine/cpu.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <setjmp.h>
#include <stdio.h>
#include <errno.h>
#include <sys/rt_suspres.h>

extern int      errno;


#define MAXNUMPROCS     10                  /* Number of child procs to create */
#define TESTTIMEOUT     (10*60)

int             ProcList[MAXNUMPROCS];   /* PIDs of child processes */
int             return_status[MAXNUMPROCS]; /* return status of
                                                child processes */
int             NumProc = 5;            /* Number of procs actually
                                         * running */
int             ProcNum;                /* Process number in the set
                                         * of child processes: a loop
                                         * counter in main, different
                                         * per-process in the
                                         * children */
 int                 Parent;            /* pid of parent process */

/* Shared memory for checking process execution order */
struct IPCbuff
{
    int         LastProcNum;            /* last process that ran (use
                                         * to check sequencing */
};
int             ProcShmid;              /* Shared memory identifier */
struct IPCbuff *Procbuff;               /* Shared memory address */

/* Shared memory for intercommunication between processes */
struct GenericBuff
{
    int         failedtests;
    int         successfulpasses;
```

```
};

int                 FTShmid;                    /*Generic Shared mem id      */
struct GenericBuff *Generic;                    /*Generic Shared mem addr.   */

/* process test ordering functions */
void            enditall ();                    /* Kills all the child proc. */
void            CheckProcessOrdering ();
void            TestTimeout ();

/* declaration for shared memory functions */
int             shmget ();
char           *shmat ();
void            shmgetat ();


main (argc, argv)
    int         argc;
    char        **argv;
{
    int         ret;                /* return code for sys calls */
    int         i;                  /* counter */
    int         j;                  /* counter */


    HandleArguments (argc, argv);

    /* Ends test at 10 mins */
    signal (SIGALRM, TestTimeout);
    alarm (TESTTIMEOUT);

    /* cleanup on signals */
    signal (SIGTERM, enditall);

    /* Attach to shared memory Generic */
    shmgetat (sizeof (*Generic), &FTShmid, &Generic);
    Generic->failedtests = 0;
    Generic->successfulpasses = 0;

    /* Attach to shared memory Procbuff*/
    shmgetat (sizeof (*Procbuff), &ProcShmid, &Procbuff);
    Procbuff->LastProcNum = NumProc - 1;

    /* get pid of parent process */
    Parent = getpid();

    /* fork off child processes */
    for (ProcNum = 0; ProcNum < NumProc; ProcNum++)
    {
        if ((ProcList[ProcNum] = fork ()) == 0)
        {
            Child ();
            exit(0);
        }
    }

    /* if parent process */
    if (Parent == getpid()) {
        /*  Wait for completion of all children
         *  2 seconds is just a time chosen at random
         */
        sleep(2);
```

```
            /* resume child processes */
            ProcList[ProcNum++] = 0;
            if (resume(SR_MULTIPLEPIDS, ProcList, NumProc, return_status) == -1) {
                printf(" resume failed\n");
                for (j=0;j<NumProc;j++)
                    printf("return_status[%d]= %d\n",j,return_status[j]);
                Generic->failedtests = 0;
            }

            /*  Wait for completion of all children
             *  5 seconds is just a time chosen at random
             */
            sleep(5);
            printf("Parent exiting\n");
            printf ("Errors so far: %d\n", Generic->failedtests);
            printf ("Successful passes so far: %d\n", Generic->successfulpasses);
            exit (0);
        }
}



/*
 * Child
 *          Child process code for test.
 * Uses
 *          extern ProcNum (per-process) - process number in test set.
 *
 */
Child ()
{
    int         i;
    int         ret;
    cpumask_t   mask;

    /*
     * In this test, done by setting RT priority, everybody waking up
     * at same time. RT priorities start at 1, not 0, so that we can
     * regain control of machine.
     */
    if ((ret = setrealpriority (0, ProcNum + 1)) < 0)
    {
        printf (" Setrealpriority failed, errno=%d\n", errno);
        Generic->failedtests++;
        proccleanup();
    }
    if ((mask = settargetcpumask(0, P_CPUMASK(0))) == -1)
    {
        printf("targetcpu failed, errno=%d\n",errno);
        Generic->failedtests++;
        proccleanup();
    }

    /* suspend child process */
    if (suspend() == -1) {
        printf(" suspend failed in ProcNum = %d\n",ProcNum);
        Generic->failedtests++;
        proccleanup();
    }
    CheckProcessOrdering();
}
```

```
/*
 * TestTimeout
 *       Stops execution after TESTTIMEOUT minutes
 */
void
TestTimeout ()
{
    printf ("Test timed out\n");
    printf ("Errors so far: %d\n", Generic->failedtests);
    printf ("Successful passes so far: %d\n", Generic->successfulpasses);
    proccleanup();
}

/*
 * enditall
 *          cleanup routine
 *          Kills child processes and removes shared memory.
 */
void
enditall ()
{
    int         i;

    /* kill all child processes */
    printf("enditall called\n");
    Procbuff->LastProcNum = -1;

    /* remove shared memory */
    shmctl (ProcShmid, IPC_RMID, 0);
    shmctl (FTShmid, IPC_RMID, 0);

    exit(0);
}

/*
 *  CheckProcessOrdering
 *      Check that the test processes execute in order.
 *      Order is determined by the test, usually using RT prio in some form.
 *      Ordering is strictly linear-circular.
 *  Uses
 *      extern ProcNum (per process) - process number in test set
 */
void
CheckProcessOrdering ()
{
    int lastproc = Procbuff->LastProcNum;

    if ( lastproc == -1)        /* parent says go away */
    {
        exit(2);
    }
    else        /* normal case */
    {
        /* check execution order */
        if ( ((lastproc + 1) % NumProc)  != ProcNum)
        {
            Generic->failedtests++;
            printf (" Illegal Process sequence\n");
            printf (" LastProcNum = %d, ProcNum=%d, Successful=%d\n",
            lastproc, ProcNum, Generic->successfulpasses);
            proccleanup();
            /* NOT REACHED */
        }
        Generic->successfulpasses++;
        Procbuff->LastProcNum = ProcNum; /* indicate this proc ran */
    }
}


/*
 * wrapper around combined calls to shmget and shmat,
 * printing and exiting on error.
 */
void
shmgetat (size, idp, basep)
    int             size;
    int            *idp;
    char          **basep;
{
    if ((*idp = shmget (IPC_PRIVATE, size, IPC_CREAT | 0660)) < 0)
    {
        printf ("shmget failed,  errno=%d\n", errno);
        proccleanup();
    }
    if ((*basep = shmat (*idp, 0, SHM_LOCK)) < 0)
    {
        printf ("shmget failed,  errno=%d\n", errno);
        proccleanup();
    }
}

/*
 * HandleArguments -- Is where test specific argument handling code goes.
 */
HandleArguments (argc, argv)
    int             argc;
    char          **argv;
{
    int             arg;

    for (; *++argv;)
    {
        /* Allows passing in number processes to create */
        if (sscanf (*argv, "-np=%d", &arg) == 1)
        {
            if (1 <= arg && arg <= MAXNUMPROCS)
            {
                NumProc = arg;
            }
                else
            {
                printf ("-np=<NumProc> must be within [1,%d]\n", MAXNUMPROCS);
                proccleanup();
            }
        }
        else
        {
            printf ("Unrecognized argument %s\n", *argv);
            proccleanup();
        }
    }
}

/*
```

```
 * proccleanup -- cleans up the child processes
 *
 */
proccleanup()
{
    int i;

    /* Kill all child processes.  A -1 in LastProcNum tells the other
     * processes that they should just exit. */
    Procbuff->LastProcNum = -1;

    exit(1);
}
```

Appendix B
Model Real-Time Applications

This appendix provides some simple examples of real-time programming under UTX/32. One simple model application has been written in C in three versions. The programs model a cyclic process that periodically writes information to a device. The three different methods provide various degrees of accuracy. They are presented in increasing order of accuracy.

The first version uses high-resolution interval timers. This version performs the work and then sets a timer. When the timer expires, the work is repeated. There is no way to determine how accurate this is, because the execution time between timers is unknown. In addition, the time for the timer system call can vary somewhat and increase the inaccuracy. This version is good for low resolution cyclic scheduling.

The second version uses high-resolution repeating timers. This is a more accurate revision of the first model. The system takes care of sending the process a signal periodically, so the user doesn't have to take the time. This removes the system call overhead. Even though this method is more accurate, it still cannot provide the greatest accuracy. This is an inherent problem of the interval timer. Time can be skewed by small amounts when programming the interval timer. The more the interval timer is used, the more it can skew.

The final version uses cyclic scheduling. This is the most accurate, but also offers the lowest resolution. The process requests cyclic scheduling and then suspends until it needs to run. The cyclic scheduler completely removes the scheduling burden from the process. All the process needs to do is suspend itself. The cyclic scheduler will wake it up at the appropriate time and the process then executes.

This model application involves most of UTX/32's major real-time facilities, such as timers or cyclic scheduling, real-time scheduling (including processor targeting), lockdown, and direct I/O.

The files and their contents are:

model.h	The header file for the model program
model.c	The version using high-resolution interval timers
model2.c	The version using high-resolution repeating timers
model3.c	The version using cyclic scheduling

B.1 Header File for Model Program

```
#define NO_OF_PROCESSES 8                /* # of processes executing */
#define DEVICE_ADDR      0x0800          /* device to use for Direct I/O */
#define MAXSVC           5               /* max # of SVCs per process */

/*
 * SVC types (one for each different SVC made in the model not including
 * SVCs made in initialization, direct I/Os, or the scheduling)...
 */

enum SVCtypes { NoSVC, HiSpeedClock, ReadClock };

int        Processor[NO_OF_PROCESSES] = {
                                         0, 0, 0, 0, 0, 0, 0, 0
                                         };
int        Priority[NO_OF_PROCESSES] = {
                                         1, 4, 5, 9, 2, 6, 8, 10
                                        };
/*
 * Process_schedule is the # of 60HZ frames to suspend before starting the
 * next execution of the process...
 */

int        Process_schedule[NO_OF_PROCESSES] = {
                                                 0, 2, 2, 2, 2, 2, 2, 2
                                                 };

enum        SVCtypes SVClist[NO_OF_PROCESSES][MAXSVC] = {
                 { NoSVC, NoSVC, NoSVC, NoSVC, NoSVC },
                 { HiSpeedClock, ReadClock, NoSVC, NoSVC, NoSVC },
                 { HiSpeedClock, ReadClock, NoSVC, NoSVC, NoSVC },
                 { HiSpeedClock, ReadClock, NoSVC, NoSVC, NoSVC },
                 { HiSpeedClock, ReadClock, NoSVC, NoSVC, NoSVC },
                 { HiSpeedClock, ReadClock, NoSVC, NoSVC, NoSVC },
                 { HiSpeedClock, ReadClock, NoSVC, NoSVC, NoSVC },
                 { HiSpeedClock, ReadClock, NoSVC, NoSVC, NoSVC }
                                                     };

/*
 * The DirectIOs struct contains the count of direct I/O operations to be
 * done on a per process basis.  The "count_physical" is the number of
 * physical (no translation) IOCLs to be executed.  The "count_logical" is the
 * number of logical (translation by direct i/o) IOCLs to be executed.  The
 * "count_IOCDS" is the number of IOCDS in the logical IOCL.
 */

struct        {
                int        count_physical;
                int        count_logical;
                int        count_IOCDS;
              } DirectIOs[NO_OF_PROCESSES] = {
                                                  { 0, 0, 0 },
                                                  { 1, 0, 2 },
                                                  { 0, 0, 0 },
                                                  { 0, 0, 0 },
                                                  { 0, 0, 0 },
                                                  { 0, 0, 0 },
                                                  { 0, 0, 0 },
                                                  { 0, 0, 0 }
```

```
                                              };

/*
 * Other globals for each process...
 */

int Cid;                /* connection ID for direct I/O */
TIOCD P_iocl[2];        /* physical IOCL list */
TIOCD L_iocl[100];      /* logical IOCL list */
```

B.2 High-Resolution Interval Timers Model

```
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/lock.h>
#include <selio/di.h>
#include <sys/rt_time.h>
#include <directfiles.h>
#include <machine/cpu.h>
#include <sys/types.h>

#include "model.h"                        /* model head that defines
                                           * the model */


#define TRUE                1             /* logical True */
#define FALSE               0             /* logical False */

extern          errno;
int             totalIOs;                 /* total # of IOs for dio */
int             check = TRUE;

void childalarm_handler();                /*alarm signal handler for child procs*/

main (argc, argv)

    int         argc;
    char        *argv[];

{
    register int    process;              /* process number              */
    int             seconds;              /* seconds to run tests        */
    int             passes;               /* # of passes to run tests    */
    union wait      waitstat;             /* status of "wait" call
                                           * return                      */
    register int    k;
    static TIOCD    C_iocl[200];          /* where converted IOCL goes   */
    static int      stat_buf[2];          /* status buffer               */
    int             err;                  /* return values from sys calls */
    int             clockrate;            /* the clock rate (e.g. 60)    */
    int             rate;                 /* clock rate in nanoseconds   */

    struct hscval   hscval, hscval2;

    /*
     * declarations for system calls...
     */
    int                 which;
    struct itimerval    value;


    if (argc < 2)
    {
        fprintf (stderr, "\nUsage: %s seconds\n", argv[0]);
        exit (1);
    }
    seconds = atoi (argv[1]);             /* get number of seconds to
                                           * run model */
    /*
```

```
     *  Get clock rate
     */
    clockrate = cycgetrate();

    /*
     *  Convert clock rate to nanoseconds
     */
    rate = (int) NS_PER_SEC / clockrate;

    /*
     *  Establish child processes
     */
    process = init_processes (seconds, &passes, argv[0]);

    /*
     *  Process = 0 is the parent process.  Processes >= 1 are
     *  the child processes.
     *  Wait for child processes to complete.
     */
    if (process == 0)
    {
        for(;;)
        {
            if (wait (&waitstat) == -1)
                break;
        }
        /* exit from parent process */
        exit(0);
    }
    else
    {
        /*
         * Set up alarm handler for child process
         */
        signal(SIGALRM, childalarm_handler);

        /*
         * Execute model for this process...
         */
        for (; passes > 0; passes--)
        {
            check = TRUE;

            /*
             * Do SVCs for this pass...
             */
            for (k = 0; k < MAXSVC; k++)
            {
                switch (SVClist[process][k])
                {
                case NoSVC:
                    break;

                case HiSpeedClock:
                    err = gethscvalue (&hscval);
                    if (err == -1)
                    {
                        fprintf (stderr,"gethscval failed errno=%d\n",
                                errno);
                        exit (1);
                    }
                    break;
    case ReadClock:
        getitimer (which, &value);
        break;

    default:
        fprintf (stderr, "%s: invalid SVC specified\n",
                 argv[0]);
        break;
    }

}

/*
 * Do Direct I/Os for this pass...
 */
for (k = 0; k < DirectIOs[process].count_physical; k++)
{
    err = diosiophys (Cid, &P_iocl[0], stat_buf, 0, 2);
    if (err == -1)
    {
        fprintf (stderr, "%s: sio physical failed, errno was: %d\n",
                 argv[0], errno);
        exit (1);
    }
}

/*
 * Turn off all IOCDs after the number for this
 * process have been completed
 */
L_iocl[DirectIOs[process].count_IOCDS - 1].i_IOflags &=
    ~IO_CMD_CHAIN;

for (k = 0; k < DirectIOs[process].count_logical; k++)
{
    err = diosiolog (Cid, &L_iocl[0],
             DirectIOs[process].count_IOCDS * 8, &C_iocl[0],
    DirectIOs[process].count_IOCDS * 8 * 2, stat_buf, 0, 2);
    if (err == -1)
    {
        fprintf (stderr, "%s: sio logical failed, errno was: %d\n",
                 argv[0], errno);
        exit (1);
    }
}

/*
 * Turn on IOCDs again
 */
L_iocl[DirectIOs[process].count_IOCDS - 1].i_IOflags |=
    IO_CMD_CHAIN;

check = FALSE;

/*
 * set up timer for next alarm
 */
hscval2.hsc_seconds = 0;
hscval2.hsc_nanosecs = Process_schedule[process] * rate;
if ((err = sethsctimer (HSC_RELATIVE, &hscval2, &hscval)) == -1)
{
    fprintf (stderr, "Sethsctimer failed errno=%d\n",
                    errno);
                exit (1);
            }

            /*
             * suspend process till alarm goes off
             */
            pause();
        }

        /*
         * if process used dio then disconnect dio
         */
        if (totalIOs > 0)
        {
            if ((err = diodisconnect (Cid)) == -1)
            {
               fprintf(stderr,"Diosiconnect failed errno=%d\n",errno);
               exit(1);
            }
        }

        /* exit from child process */
        exit(0);
    }
}

/*
 * init_processes -- sets up the child processes.
 *                  1) forks the child process off
 *                  2) Locks process in memory
 *                  3) Targets process to CPU
 *                  4) and Sets up dio operations if necessary
 *
 *      RETURNS ->  The process number
 *
 */
init_processes (seconds, passes, modelname)

    int              seconds;
    int             *passes;
    char            *modelname;

{
    int              i;                  /* counter variable             */
    register int     process;            /* process #                    */
    static int       seekword,
                     buffer;
    static int       notify[2];
    cpumask_t        mask;         /* used to target process to CPU or IPU*/

    /*
     * process 0 is the parent and processes >= 1 are the children
     */
    process = 0;

    /*
     * fork off all processes...
     */
    for (i = 0; i < NO_OF_PROCESSES - 1; i++)
    {
        if (fork () == 0)
        {
            process = i + 1;
            break;
        }
    }

    /*
     * if parent process return
     */
     if (process == 0)
         return(process);

    /*
     * compute # of passes...
     */
    *passes = 60 * seconds / Process_schedule[process];

    /*
     * lock down process in memory...
     */
    if (plock (PROCLOCK) == -1)
    {
        fprintf (stderr, "%s: Unable to lock down process #%d, errno was: %d\n",
                       modelname, process, errno);
        exit (1);
    }

    /*
     * schedule process at a real-time priority...
     */
    setrealpriority (0, Priority[process]);

    /*
     * target process to CPU
     */
    if ((mask = settargetcpumask(0, P_CPUMASK(Processor[process]))) == -1)
    {
        fprintf(stderr,"targetcpu failed, errno=%d\n",errno);
        exit(1);
    }

    totalIOs = DirectIOs[process].count_physical +
        DirectIOs[process].count_logical;

    if (totalIOs > 0)
    {
        /*
         * connect to direct i/o...
         */
        Cid = dioconnect (DEVICE_ADDR, 0, totalIOs, notify);
        if (Cid == -1)
        {
            fprintf (stderr,"%s:unable to connect to dio, errno =%d\n",
                           modelname, errno);
            exit (1);
        }

        /*
         * This example uses disk I/O
         * Create the IOCL
         */
        L_iocl[0].i_IOCmd = IO_SEEK;
```

```
            L_iocl[0].i_Address = (unsigned) &seekword;
            L_iocl[0].i_IOflags = IO_CMD_CHAIN;
            L_iocl[0].i_XferCount = 4;

            L_iocl[1].i_IOCmd = IO_READ;
            L_iocl[1].i_Address = (unsigned) &buffer;
            L_iocl[1].i_IOflags = 0;
            L_iocl[1].i_XferCount = 4;

            /*
             * convert logical IOCL to physical IOCL
             */
            if (dioconvert (Cid, &L_iocl[0], 16, &P_iocl[0], 16) == -1)
            {
                fprintf (stderr, "%s: Error converting logical to physical IOCL, errno was: %d\n", modelna
                exit (1);
            }

            /*
             * complete IOCL
             */
            L_iocl[1].i_IOflags = IO_CMD_CHAIN;
            for (i = 0; i < 97; i = i + 2)
            {
                L_iocl[i + 2] = L_iocl[i];
                L_iocl[i + 3] = L_iocl[i + 1];
            }
            L_iocl[99].i_IOflags = 0;
        }
        return (process);
}


/*
 * childalarm_handler
 */
void
childalarm_handler()
{

        /*
         * check to see if system calls and dio's
         * completed before alarm received
         */
        if (check)
        {
            fprintf(stderr,"SVC and DIO code not completed\n");
        }

}
```

B.3 High-Resolution Repeating Timers Model

```
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/lock.h>
#include <selio/di.h>
#include <sys/rt_time.h>
#include <directfiles.h>
#include <machine/cpu.h>
#include <sys/types.h>

#include "model.h"                          /* model head that defines
                                             * the model    */


#define TRUE            1                   /* logical True */
#define FALSE           0                   /* logical False*/

extern int              errno;              /* error number                 */
int                     totalIOs;           /* total # of IOs for dio       */
int                     check = TRUE;

void childalarm_handler();                  /* alarm signal handler for child
                                             * child processes
                                             */

main (argc, argv)

    int             argc;
    char           *argv[];

{
    register int    process;                /* process number               */
    int             seconds;                /* seconds to run tests         */
    int             passes;                 /* # of passes to run tests     */
    union wait      waitstat;               /* status of "wait" call
                                             * return                       */
    register int    k;
    static TIOCD    C_iocl[200];            /* where converted IOCL goes    */
    static int      stat_buf[2];            /* status buffer                */
    int             err;                    /* holds sys call return values */
    int             clockrate;              /* the clock rate (e.g. 60)     */
    int             rate;                   /* clock rate in nanoseconds    */


    struct hscval   hscval, hscval2;

    /*
     * declarations for system calls...
     */
    int                 which;
    struct itimerval    value;


    if (argc < 2)
    {
        fprintf (stderr, "\nUsage: %s seconds\n", argv[0]);
        exit (1);
    }
```

```
seconds = atoi (argv[1]);            /* get number of seconds to
                                      * run model */
/*
 * get clock rate
 */
clockrate = cycgetrate();

/*
 * convert clock rate to nanoseconds
 */
rate = (int) NS_PER_SEC / clockrate;

/*
 * Establish child processes
 */
process = init_processes (seconds, &passes, argv[0]);

/*
 * if parent process wait for child processes to complete
 */
if (process == 0)
{
    for (;;)
    {
        if (wait (&waitstat) == -1)
            break;
    }

    /* exit from parent */
    exit(0);
}
else
{
    /*
     * Execute model for this process...
     */
    signal(SIGALRM, childalarm_handler);
    hscval2.hsc_seconds = 0;
    hscval2.hsc_nanosecs = Process_schedule[process] * rate;
    if ((err = sethsctimer (HSC_CYCLE, &hscval2, &hscval)) == -1)
    {
        fprintf (stderr, "Sethsctimer failed errno=%d\n",
             errno);
        exit (1);
    }
    for (; passes > 0; passes--)
    {
        check = TRUE;

        /*
         * Do SVCs for this pass...
         */
        for (k = 0; k < MAXSVC; k++)
        {
            switch (SVClist[process][k])
            {
            case NoSVC:
                break;

            case HiSpeedClock:
                err = gethscvalue (&hscval);
                if (err == -1)
                {
                    printf ("gethscval failed errno=%d\n",
                            errno);
                    exit (1);
                }
                break;

            case ReadClock:
                getitimer (which, &value);
                break;

            default:
                fprintf (stderr, "%s: invalid SVC specifed\n",
                         argv[0]);
                break;
            }

        }

        /*
         * Do Direct I/Os for this pass...
         */
        for (k = 0; k < DirectIOs[process].count_physical; k++)
        {
            err = diosiophys (Cid, &P_iocl[0], stat_buf, 0, 2);
            if (err == -1)
            {
                fprintf (stderr, "%s: sio physical failed, errno was: %d\n",
                         argv[0], errno);
                exit (1);
            }
        }

        L_iocl[DirectIOs[process].count_IOCDS - 1].i_IOflags &=
            ~IO_CMD_CHAIN;

        for (k = 0; k < DirectIOs[process].count_logical; k++)
        {
            err = diosiolog (Cid, &L_iocl[0],
                      DirectIOs[process].count_IOCDS * 8, &C_iocl[0],
            DirectIOs[process].count_IOCDS * 8 * 2, stat_buf, 0, 2);
            if (err == -1)
            {
                fprintf (stderr, "%s: sio logical failed, errno was: %d\n",
                         argv[0], errno);
                exit (1);
            }
        }
        L_iocl[DirectIOs[process].count_IOCDS - 1].i_IOflags |=
            IO_CMD_CHAIN;

        /* sleep(1); do this until cyclic suspend is working */
        check = FALSE;
        pause();
    }

    /*
     * if dio's used in this process disconnect dio
     */
    if (totalIOs > 0)
    {
        if ((err = diodisconnect (Cid)) == -1)
```

```
              {
                  fprintf(stderr,"Diodisconnect failed errno=%d\n",errno);
                  exit(1);
              }
          }
          /* exit from child process */
          exit(0);
     }
}

/*
 * init_processes -- sets up the child processes.
 *                   1) forks the child process off
 *                   2) Locks process in memory
 *                   3) Targets process to CPU
 *                   4) and Sets up dio operations if necessary
 *
 *      RETURNS ->  The process number
 *
 */
init_processes (seconds, passes, modelname)

    int              seconds;
    int             *passes;
    char            *modelname;

{
    int              i;                 /* counter variable        */
    register int     process;         /* process #               */
    static int       seekword,
                     buffer;
    static int       notify[2];
    cpumask_t       mask;         /* mask used to target process to CPU or IPU*/

    /*
     * process = 0 is the parent process process >= 1 are the child
     *    processes
     */
    process = 0;

    /*
     * fork off all processes...
     */
    for (i = 0; i < NO_OF_PROCESSES - 1; i++)
    {
        if (fork () == 0)
        {
            process = i + 1;
            break;
        }
    }

    /*
     * if parent process return
     */
    if (process == 0)
        return(process);

    /*
     * compute # of passes...
     */
```

```
*passes = 60 * seconds / Process_schedule[process];

/*
 * lock down process in memory...
 */
if (plock (PROCLOCK) == -1)
{
    fprintf (stderr, "%s: Unable to lock down process #%d, errno was: %d\n",
                              modelname, process, errno);
    exit (1);
}

/*
 * schedule process at a priority...
 */
setrealpriority (0, Priority[process]);

/*
 * target process to CPU
 */
if ((mask =settargetcpumask(0,P_CPUMASK(Processor[process]))) == -1)
{
    fprintf(stderr,"targetcpu failed, errno=%d\n",errno);
    exit(1);
}

/*
 * connect to direct i/o...
 */
totalIOs = DirectIOs[process].count_physical +
    DirectIOs[process].count_logical;

if (totalIOs > 0)
{
    Cid = dioconnect (DEVICE_ADDR, 0, totalIOs, notify);
    if (Cid == -1)
    {
        fprintf (stderr,"%s:unable to connect to dio, errno =%d\n",
                       modelname, errno);
        exit (1);
    }

    /*
     * create iocd structures (very machine dependent)... ...its
     * also for the disk only, since thats the only device I've
     * got...
     */
    L_iocl[0].i_IOCmd = IO_SEEK;
    L_iocl[0].i_Address = (unsigned) &seekword;
    L_iocl[0].i_IOflags = IO_CMD_CHAIN;
    L_iocl[0].i_XferCount = 4;

    L_iocl[1].i_IOCmd = IO_READ;
    L_iocl[1].i_Address = (unsigned) &buffer;
    L_iocl[1].i_IOflags = 0;
    L_iocl[1].i_XferCount = 4;

    if (dioconvert (Cid, &L_iocl[0], 16, &P_iocl[0], 16) == -1)
    {
        fprintf (stderr, "%s: Error converting logical to physical IOCL, errno was: %d\n",
                              modelname, errno);
        exit (1);
        }
        L_iocl[1].i_IOflags = IO_CMD_CHAIN;

        for (i = 0; i < 97; i = i + 2)
        {
            L_iocl[i + 2] = L_iocl[i];
            L_iocl[i + 3] = L_iocl[i + 1];
        }
        L_iocl[99].i_IOflags = 0;
    }
    return (process);
}

/*
 * childalarm_handler
 */
void
childalarm_handler()
{

    if (check)
    {
        fprintf(stderr,"SVC and DIO code not completed\n");
    }

}
```

B.4 Cyclic Scheduling Model

```
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/lock.h>
#include <selio/di.h>
#include <sys/rt_time.h>
#include <directfiles.h>
#include <machine/cpu.h>
#include <sys/types.h>

#include <sys/param.h>
#include <sys/cyclic.h>
#include <sys/proc.h>
#include <errno.h>
extern int      errno;

#include "model.h"                      /* model head that defines
                                         * the model         */



int             totalIOs;               /* total # of IOs for dio         */

Tcyclicdata     TestCycle;

main (argc, argv)

    int         argc;
    char        *argv[];

{
    register int        process;                /* process number             */
    int                 seconds;                /* seconds to run tests       */
    int                 passes;                 /* # of passes to run tests   */
    union wait          waitstat;               /* status of "wait" call
                                                 * return         */
    register int        k;
    static TIOCD        C_iocl[200];            /* where converted IOCL goes */
    static int          stat_buf[2];            /* status buffer             */
    int                 err;
    int                 curframe;

    struct hscval   hscval;

    /*
     * declarations for system calls...
     */
    int                 which;
    struct itimerval    value;


    if (argc < 2)
    {
        fprintf (stderr, "\nUsage: %s seconds\n", argv[0]);
        exit (1);
    }
    seconds = atoi (argv[1]);                   /* get number of seconds to
                                                 * run model */
```

```
/*
 *  Establish child processes
 */
process = init_processes (seconds, &passes, argv[0]);


/*
 * process 0 is the parent process  process >= 1 are child processes
 * in order for cyclic scheduled processes to work properly
 * they must be cycsynced
 */
if (process == 0)
{
     /*
      *  All processes must be established before cycsync can be
      *  performed. So sleep for a while
      */
     sleep(4);

     if ((err = cycsync ()) < 0)
     {
          fprintf (stderr,"sync failed,  errno=%d\n", errno);
          exit (1);
     }

     /*
      * wait for child processes to complete
      */
     for (;;)
     {
          if (wait (&waitstat) == -1)
               break;
     }

     /* exit from parent process */
     exit(0);
}
else
{

    /*
     * Execute model for this process...
     */
    for (; passes > 0; passes--)
    {
         /*
          *  suspend process till its next set frame
          */
         cycsuspend();

         /*
          * Do SVCs for this pass...
          */
         for (k = 0; k < MAXSVC; k++)
         {
              switch (SVClist[process][k])
              {
              case NoSVC:
                   break;

              case HiSpeedClock:
                   err = gethscvalue (&hscval);
```

```
                if (err == -1)
                {
                    fprintf (stderr,"gethscval failed errno=%d\n",
                             errno);
                    exit (1);
                }
                break;

            case ReadClock:
                getitimer (which, &value);
                break;

            default:
                fprintf (stderr, "%s: invalid SVC specified\n",
                         argv[0]);
                break;
            }

        }

        /*
         * Do Direct I/Os for this pass...
         */
        for (k = 0; k < DirectIOs[process].count_physical; k++)
        {
            err = diosicphys (Cid, &P_iocl[0], stat_buf, 0, 2);
            if (err == -1)
            {
                fprintf (stderr, "%s: sio physical failed, errno was: %d\n",
                         argv[0], errno);
                exit (1);
            }
        }

        /*
         * Turn off all IOCDs after the numbers for this process
         * have been completed.
         */
        L_iocl[DirectIOs[process].count_IOCDS - 1].i_IOflags &=
            ~IO_CMD_CHAIN;

        for (k = 0; k < DirectIOs[process].count_logical; k++)
        {
            err = diosiolog (Cid, &L_iocl[0],
                     DirectIOs[process].count_IOCDS * 8, &C_iocl[0],
            DirectIOs[process].count_IOCDS * 8 * 2, stat_buf, 0, 2);
            if (err == -1)
            {
                fprintf (stderr, "%s: sio logical failed, errno was: %d\n",
                         argv[0], errno);
                exit (1);
            }
        }

        /*
         * Turn on IOCDs again
         */
        L_iocl[DirectIOs[process].count_IOCDS - 1].i_IOflags |=
            IO_CMD_CHAIN;
    }

    /*
```

```
         * if this process previously connected dio,
         * then disconnect dio
         */
        if (totalIOs > 0)
        {
            if ((err = diodisconnect (Cid)) == -1)
            {
               fprintf(stderr,"Diosiconnect failed errno=%d\n",errno);
               exit(1);
            }
        }

        /* exit from child process */
        exit(0);
    }
}

/*
 * init_processes -- sets up the child processes.
 *                1) forks the child process off
 *                2) Locks process in memory
 *                3) Targets process to CPU
 *                4) Sets up cyclic scheduler
 *                5) and Sets up dio operations if necessary
 *
 *     RETURNS ->  The process number
 *
 */
init_processes (seconds, passes, modelname)

    int              seconds;
    int             *passes;
    char            *modelname;

{
    int              i;                   /* counter variable                */
    register int     process;             /* process #                       */
    static int       seekword,
                     buffer;
    static int       notify[2];
    cpumask_t        mask;                /* target process to CPU or IPU    */
    int              no_of_frames;        /* no of frames in a cycle         */

    process = 0;                          /* process = 0 is parent process   */

    /*
     *  Get Number of frames per cycle
     */
    no_of_frames = cycgetrate();

    /*
     * fork off all processes...
     */
    for (i = 0; i < NO_OF_PROCESSES - 1; i++)
    {
        if (fork () == 0)
        {
            process = i + 1;
            break;
        }
    }
```

```
/*
 * if parent process, return
 */
if (process == 0)
    return(process);

/*
 * compute # of passes...
 */

*passes = 60 * seconds / Process_schedule[process];

/*
 * lock down process in memory...
 */
if (plock (PROCLOCK) == -1)
{
    fprintf (stderr,"%s: Unable to lock down process #%d, errno was: %d\n",
                   modelname, process, errno);
    exit (1);
}

/*
 * schedule process at a real-time priority...
 */
setrealpriority (0, Priority[process]);

/*
 * target process to CPU
 */
if ((mask = settargetcpumask(0, P_CPUMASK(Processor[process]))) == -1)
{
    fprintf(stderr,"targetcpu failed, errno=%d\n",errno);
    exit(1);
}

/* Make cyclically scheduled */
TestCycle.cycle_length = no_of_frames;
TestCycle.frames_lookback = 0;
for (i = 0; i < no_of_frames; i++)
{
    if ((i % Process_schedule[process]) == 0)
        TestCycle.cycle[i] = 1;
    else
        TestCycle.cycle[i] = 0;
}
if (cycsetdata (&TestCycle, 1) == -1)
{
    fprintf (stderr,"cycsetdata failed process %d pid %d errno %d\n",
            process, getpid (), errno);
    exit (1);
}

/*
 * connect to direct i/o...
 */
totalIOs = DirectIOs[process].count_physical +
    DirectIOs[process].count_logical;

if (totalIOs > 0)
{
    Cid = dioconnect (DEVICE_ADDR, 0, totalIOs, notify);
```

```
        if (Cid == -1)
        {
            fprintf (stderr,"%s:unable to connect to dio, errno =%d\n",
                           modelname, errno);
            exit (1);
        }

        /*
         * this example uses disk I/O
         */
        L_iocl[0].i_IOCmd = IO_SEEK;
        L_iocl[0].i_Address = (unsigned) &seekword;
        L_iocl[0].i_IOflags = IO_CMD_CHAIN;
        L_iocl[0].i_XferCount = 4;

        L_iocl[1].i_IOCmd = IO_READ;
        L_iocl[1].i_Address = (unsigned) &buffer;
        L_iocl[1].i_IOflags = 0;
        L_iocl[1].i_XferCount = 4;

        /*
         * convert logical dio to physical dio
         */
        if (dioconvert (Cid, &L_iocl[0], 16, &P_iocl[0], 16) == -1)
        {
            fprintf (stderr, "%s: Error converting logical to physical IOCL, errno was: %d\n",
                                 modelname, errno);
            exit (1);
        }

        /*
         * Create IOCL
         */
        L_iocl[1].i_IOflags = IO_CMD_CHAIN;
        for (i = 0; i < 97; i = i + 2)
        {
            L_iocl[i + 2] = L_iocl[i];
            L_iocl[i + 3] = L_iocl[i + 1];
        }
        L_iocl[99].i_IOflags = 0;
    }
    return (process);
}
```

Gould Inc., Computer Systems Division
6901 W. Sunrise Blvd.
P. O. Box 409148
Fort Lauderdale, FL 33340-9148
Telephone (305) 587-2900

GOULD
Electronics

Users Group Membership Application

USER ORGANIZATION: ______________________________

REPRESENTATIVE(S): ______________________________

____

____

ADDRESS: ______________________________

____

TELEX NUMBER: ______________ PHONE NUMBER: ______________

NUMBER AND TYPE OF GOULD CSD COMPUTERS: ______________________________

____

OPERATING SYSTEM AND REV. LEVEL: ______________________________

____

APPLICATIONS (Please Indicate)

1. EDP
   - A. Inventory Control
   - B. Engineering & Production Data Control
   - C. Large Machine Off-Load
   - D. Remote Batch Terminal
   - E. Other

2. Communications
   - A. Telephone System Monitoring
   - B. Front End Processors
   - C. Message Switching
   - D. Other

3. Design & Drafting
   - A. Electrical
   - B. Mechanical
   - C. Architectural
   - D. Cartography
   - E. Image Processing
   - F. Other

4. Industrial Automation
   - A. Continuous Process Control Op.
   - B. Production Scheduling & Control
   - C. Process Planning
   - D. Numerical Control
   - E. Other

5. Laboratory and Computational
   - A. Seismic
   - B. Scientific Calculation
   - C. Experiment Monitoring
   - D. Mathematical Modeling
   - E. Signal Processing
   - F. Other

6. Energy Monitoring & Control
   - A. Power Generation
   - B. Power Distribution
   - C. Environmental Control
   - D. Meter Monitoring
   - E. Other

7. Simulation
   - A. Flight Simulators
   - B. Power Plant Simulators
   - C. Electronic Warfare
   - D. Other

8. Other

Please return to:

Users Group Representative

Date: ______________

243-06-1 (1/86)

Gould Inc., Computer Systems Division Users Group. . .

The purpose of the Gould CSD Users Group is to help create better User/User and User/Gould CSD communications.

There is no fee to join the Users Group. Simply complete the Membership Application on the reverse side and mail to the Users Group Representative. You will automatically receive Users Group Newsletters, Referral Guide and other pertinent Users Group activity information.

Fold and Staple for Mailing

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 947 FT. LAUDERDALE, FL

POSTAGE WILL BE PAID BY ADDRESSEE

GOULD INC., COMPUTER SYSTEMS DIVISION
ATTENTION: USERS GROUP REPRESENTATIVE
6901 W. SUNRISE BLVD.
P. O. BOX 409148
FT. LAUDERDALE FL 33340-9970

Fold and Staple for Mailing

(Detach Here)

GOULD
Electronics