Data General

Customer Documentation

Programming in the DG/UX™ Kernel Environment

AViiON®
PRODUCT LINE

Programming in the DG/UX™ Kernel Environment

093-701083-00

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Programming in the DG/UX™ Kernel Environment

093-701083-00

Revision History:	Effective with:
Original Release – March 1991	DG/UX Release 5.4
Addendum 083-000426 – Feb. 1992	DG/UX Release 5.4.1

Updating Instructions

This addendum updates *Programming in the DG/UX Kernel Environment* (093-701083-00) with a new appendix, Appendix D, that describes how to take advantage of the symmetric multiprocessing environment with the DG/UX STREAMS facility.

To update your copy of 093-701083-00, please remove the manual pages listed below and replace them with addendum pages as follows:

REMOVE	INSERT
Title/Notice Page	Title/Notice Page
~~xii~~	~~xii~~ xv
-	D-1/~~D25~~ D14

Insert this instruction sheet immediately behind the new Title/Notice page.

Preface

This manual provides the basic information necessary for programming in the DG/UX™ kernel environment. It includes an overview to the major features of kernel-level programming on the DG/UX system. It is also a reference manual for kernel utility routines that you can use to perform various programming tasks.

Who Should Read This Manual?

Readers should be programmers who are generally knowledgeable about operating system design topics. We presume a basic understanding of concepts such as virtual memory, synchronization, mutual exclusion, locking, and interrupts. Readers should also be familiar with the standard UNIX® concept of character special devices, block special devices, and the difference between the two.

We also assume familiarity with the C programming language, because the interfaces presented in this document are written in C.

Manual Organization

The manual is organized into parts as follows:

Part 1: Chapters 1 through 3	Part 1 of the manual gives the general lay of the land by describing kernel features and programming facilities. Chapter 1 reviews general kernel programming concepts with an eye to any differences caused by the fully-symmetric multiprocessing environment. Chapter 2 covers the DG/UX kernel's special facilities for passing information between kernel-level code and the user. Chapter 3 looks at other kernel-level facilities and their associated documentation.
Part 2: Chapters 4 through 8	Part 2 is a reference section covering kernel utility routines used in process synchronization and timing. Chapters 4 though 8 cover respectively: locks, eventcounters, clock routines, signals and process groups, and interrupt handling. Each chapter contains a brief overview of its topic area and a description of how and when you use the various routines.
Part 3: Chapters 9 through 11	Part 3 is a reference section covering kernel utility routines used in data and memory management. Chapters 9 though 11 cover respectively: memory allocation and deallocation, user data access validation,

and buffer vector management. Each chapter contains a brief overview of its topic area and a description of how and when you use the various routines.

Part 4: Chapters 12 through 16	Part 4 is a reference section covering kernel utility routines used in various driver operations. Chapters 12 though 16 cover respectively: configuration, handling the driver and generic daemons, error encoding and logging, select operations, miscellaneous. Each chapter contains a brief overview of its topic area and a description of how and when you use the various routines.
Appendix A	Lists standard peripherals and their default device codes, interrupt levels, and memory-mapped I/O addresses.
Appendix B	Describes how to set up system and master file entries.
Appendix C	Describes how to build a new kernel and check your driver's configuration.

Other Organizations' Documents

The following manuals and papers provide information from other organizations that you may find useful. To obtain a document contact the listed organization directly.

The primary method of synchronization provided by the kernel is eventcounters and sequencers. These were first described in the paper: "Synchronization with Eventcounts and Sequencers," David P. Reed and Rajendra K Kanodia, *Proceedings of the Sixth Symposium on Operating System Principles*, Purdue University, West Lafayette, IN, November 1977. They are also described in: "Synchronization with Eventcounts and Sequencers," David P. Reed and Rajendra K. Kanodia, *Communications of the ACM*, Vol. 22, Number 2, February 1979, pp. 115-123.

A detailed discussion of locks on the DG/UX system can be found in the paper: "Multiprocessor Aspects of the DG/UX Kernel", Kelley, Michael H., *Proceedings of the Winter 1989 USENIX Conference*. The USENIX Association, Berkeley, CA., 1989, 85-99.

Readers, Please Note

Data General manuals use certain symbols and styles of type to indicate different meanings. The Data General symbol and typeface conventions used in this manual are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

This manual also presumes the following meanings for the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt and is followed by a delimiter such as the curved arrow symbol for the New Line key. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	All DG/UX commands, system calls, pathnames, names of files, directories, and manual pages also use this typeface.
`constant width monospace`	Syntax lines and examples of code use this font.
italic	Represents variables for which you supply values; for example, arguments to routines. In text, italics are also used to emphasize a term that is used for the first time.

Contacting Data General

Data General wants to assist you in any way it can to help you use its products. Please feel free to contact the company as outlined below.

Manuals

If you require additional manuals, please use the enclosed TIPS order form (United States only) or contact your local Data General sales representative. A list of related documents appears at the end of this manual with the TIPS order form.

For a complete list of AViiON® and DG/UX™ manuals, see the *Guide to AViiON® and DG/UX™ System Documentation* (069-701085). The on-line version of this manual found in **/usr/release/doc_guide** contains the most current list.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, free telephone assistance is available with your hardware warranty and with most Data General software service options. If you are within the United States or Canada, contact the Data General Service Center by calling 1-800-DG-HELPS. Lines are open from 8:00 a.m. to 5:00 p.m., your time, Monday through Friday. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

Joining Our Users Group

Please consider joining the largest independent organization of Data General users, the North American Data General Users Group (NADGUG). In addition to making valuable contacts, members receive FOCUS monthly magazine, a conference discount, access to the Software Library and Electronic Bulletin Board, an annual Member Directory, Regional and Special Interest Groups, and much more. For more information about membership in the North American Data General Users Group, call 1-800-877-4787 or 1-512-345-5316.

End of Preface

Contents

Contents

Figures

Tables

Figures

Chapter 1 How Things Work in the DG/UX Kernel

This chapter provides a general overview of the DG/UX operating system environment. We assume that you have a working knowledge of programming and operating systems in general. Documents covering special topics such as eventcounters and sequencers are listed in the Preface.

What's Special About the DG/UX Kernel?

There are two major ways in which the DG/UX kernel is special:

- It is designed for fully symmetric multiprocessor operation throughout the kernel.

  Multiprocessing means that the architecture has multiple CPUs operating out of common shared memory. Fully symmetric means that all CPUs are equal; there are no master or central CPUs. The DG/UX kernel's fully symmetric multiprocessor implementation allows any part of the kernel to run on any CPU and in parallel. This full concurrency means full use of the CPUs. Processes really do run in parallel on separate processors and you don't have to do anything to take advantage of this.

- It is a re-implementation of the UNIX kernel (not a modified AT&T or Berkeley kernel) with greatly improved internal structure and modularity.

  The re-implementation and increased modularity means a better software development environment. Modularity means well specified interfaces that: 1) allow multiple developers to work independently; 2) allow the implementation of one part of the kernel to change without affecting another; 3) simplify integration of new software; and 4) meet government modularity requirements for secure systems.

These two elements — full-symmetry and a cleanly designed kernel — are at the heart of what makes the DG/UX system a powerful software environment especially at the kernel level.

How Does a Special Kernel Fit in the World of Industry Standards?

The DG/UX system is a UNIX system that "aggressively" adheres to industry standards such as BCS, POSIX, SVID, ABI, and BSD that define user-level interfaces. Thus, user-level portability is assured on the multiprocessor DG/UX kernel.

Even within the kernel, most of the basic concepts, operations, and data structures are the same as in traditional UNIX kernels. However, fully symmetric multiprocessing does change the kernel-level programming environment. Primarily it means more sophisticated synchronization (fine-grained locks and program-definable events) and scheduling (fully pre-emptive scheduling). It also means some shifts in the standard assumptions about what level of exclusivity and control a kernel-level process has while its running. This manual is designed to describe these kernel-level programming differences.

This manual is also designed to give you tools to program effectively on the DG/UX kernel. The DG/UX kernel is designed to simplify integration of new kernel-level software. We've already mentioned the first major simplification: modular engineering and well-defined interfaces between modules. The DG/UX system also simplifies kernel programming by providing a wide range of utilities covering all aspects of kernel programming from locks and interrupt handling to data and memory management. These routines both reduce the programming effort and hide the intricacies of full parallelism. If you use the kernel-supplied utilities, for the most part you don't need to know how full-symmetry works and you run less risk of having problems using it.

Kernel-level Programming Tasks and Documentation

There are three kernel-level programming manuals: an introduction and reference manual, and two manuals each describing a particular facility for integrating code into the kernel. Figure 1-1 outlines the structure of kernel-level documentation.

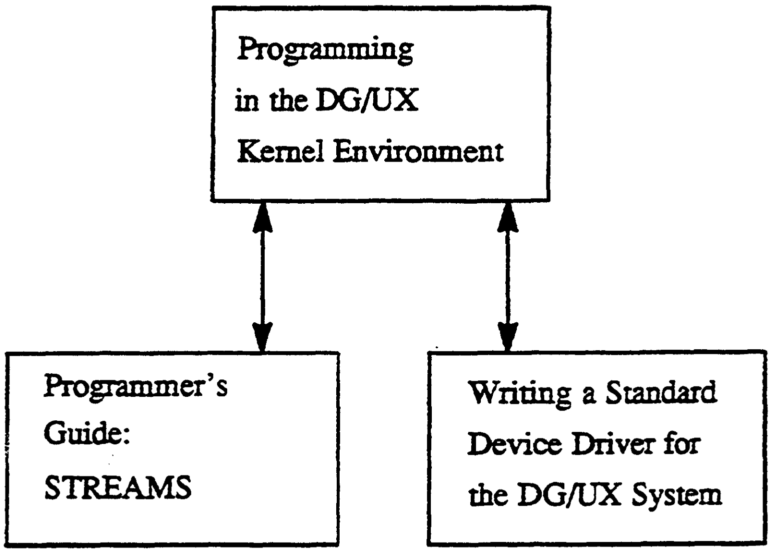

Programming in the DG/UX Kernel Environment
Programmer's Guide: STREAMS
Writing a Standard Device Driver for the DG/UX System

Figure 1-1 *Kernel-level Documentation*

This manual provides the entry point for anyone wanting to do kernel-level programming on the DG/UX system. It provides an overview of the DG/UX kernel-level programming environment and contains reference pages for the many kernel-supplied utilities you will need to perform support operations.

The DG/UX kernel provides two well-defined entry points for integrating foreign code: through the STREAMS facility or through the Standard (non-STREAMS) DG/UX Device Driver Interface. The two other manuals listed above describe these facilities and give detailed information on how you use them to integrate your code into the kernel.

The Standard DG/UX Device Driver Interface is the original and most basic interface to the DG/UX kernel. To create a device driver using this interface you supply a set of routines that conform to the interface specification. The specification describes the functionality of required and optional routines and gives the exact calling sequence you must supply for the main kernel code to call your driver.

The STREAMS facility provides special functions designed to make mixing and matching modules of code easier. It defines a standard interface between modules, ways to send messages between them, tools for managing common buffers, and easy ways to connect modules. The STREAMS facility is particularly handy for communications protocols and line disciplines that often consist of layers in combinations that may vary from system to system. For example, you might have a TCP/IP layer on an X.25 base on one system and an ISO layer on an X.25 base on another system. By standardizing flow of control, the STREAMS facility allows you to switch modules in and out to fit your system without redesigning the whole stream. To create a stream using the STREAMS interface, you write one or more modules and/or drivers that are connected and exchange information using the the STREAMS facility.

In theory you can change any part of an existing kernel, but in practice doing so requires extensive knowledge of the entire system and is beyond the scope of this document. Rather, this document focuses on the most common kernel-programming task: writing various types of I/O-related code such as device drivers, communications protocols, and line disciplines. To simplify discussion, we will speak in terms of "drivers" when referring to all such code. The goal of the DG/UX kernel-level documentation is to allow you to write drivers without having a Source License for kernel code.

The Structure of the DG/UX Kernel

This section describes the organization of code inside the kernel. The DG/UX kernel has the same basic layout as traditional UNIX kernels. It consists first of a separation between user-level and kernel-level with system calls acting as the interface or path between the two levels. In some cases a user program may call a library routine that calls a system call, but, whether directly or not, system calls are the user's path into the kernel.

The DG/UX kernel was designed with two guiding themes: 1) hierarchy, and 2) information hiding. Hierarchy means that a large software entity is divided into smaller entities. At each level, division is done on the basis of related functions to create meaningful software elements. As a result, the DG/UX system has four levels of hierarchy: 1) the kernel as a whole; 2) subsystems; 3) modules; 4) individual functions and data structure definitions. The kernel has about 50 subsystems each containing a set of related functions that support a major area of kernel operation. Subsystems consist of multiple modules, where a module consists of a C language source file.

Information hiding means that each software entity is separated into an interface and an implementation. The interface is visible; it forms the contract between the two entities. The implementation is hidden from outside entities. This means that an entity's implementation can change without its users being affected. The interface/implementation principle applies to all levels of the hierarchy.

The interface to each subsystem consists of a set of function calls and data structure definitions. The kernel-supplied utility routines mentioned earlier are calls to functions in different subsystems. Figure 1-2 shows some of the subsystems, modules and other components relevant to drivers. It shows general flow, though not all connections between subsystems. For example, a driver may call routines in the **vp** subsystems, and the **lm** subsystem is used by virtually all other code.

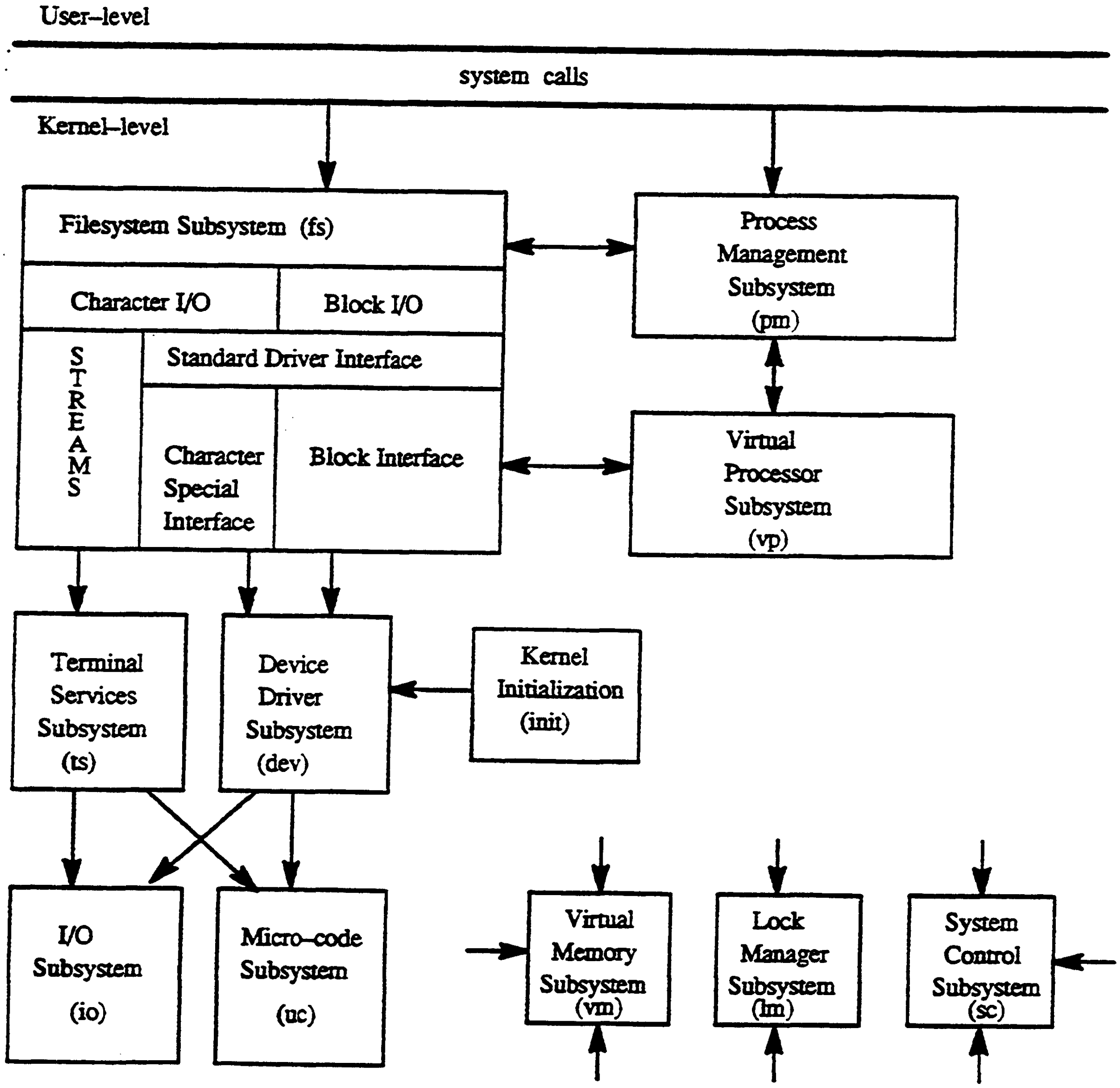

User–level
system calls
Kernel–level
Filesystem Subsystem (fs)
Character I/O
Block I/O
STREAMS
Standard Driver Interface
Character Special Interface
Block Interface
Process Management Subsystem (pm)
Virtual Processor Subsystem (vp)
Terminal Services Subsystem (ts)
Device Driver Subsystem (dev)
Kernel Initialization (init)
I/O Subsystem (io)
Micro-code Subsystem (uc)
Virtual Memory Subsystem (vm)
Lock Manager Subsystem (lm)
System Control Subsystem (sc)

Figure 1-2 *Layout of the DG/UX Kernel*

The following list gives a brief description of some of the important subsystems.

File system subsystem (fs) — The file system subsystem is a utility subsystem that contains routines of general use to the kernel. For our purposes, the important thing about the fs subsystem is that it is the entry point for kernel I/O. The fs subsystem channels processing requests to the correct lower-level subsystem.

Device Driver subsystem (dev) — The device driver subsystem contains device drivers for disk, tape, and line-printer devices and for non-Data General drivers. If you write a device driver, the driver will be part of this subsystem.

I/O subsystem (io) — The I/O subsystem implements a variety of I/O utility routines that are used by all device drivers. It defines the standard device interface and the general I/O architecture.

Micro-code subsystem (uc) — The micro-code subsystem isolates the rest of the kernel from variation in AViiON hardware platforms, allowing other kernel code to work on different machines without coding changes. It provides routines to generalize I/O interrupts, device codes, and hardware queries.

Kernel Initialization subsystem (init) — The kernel initialization subsystem initializes the kernel. It calls each subsystem's initialization routine, mounts the root file system and invokes the first user process. You will not need to call functions in this subsystem; rather, if you write a device driver, this subsystem will call the driver.

Lock management subsystem (lm) — The lock management subsystem handles all locking operations.

Process management subsystem (pm) — The process management subsystem manages processes including process attributes such as **uid** and **gid**, signals, and the **fork** and **exec** system calls.

System control subsystem (sc) — The system control subsystem handles various low-level system functions including: panic and dump code; hardware exception handlers; and kernel-code accessing of User space.

Virtual memory subsystem (vm) — The virtual memory subsystem implements the UNIX address space and virtual memory, and supports the shared memory mechanism.

Virtual process subsystem (vp) — The virtual process subsystem handles the virtual processor abstraction used to create the fully symmetric environment. As "virtual" processor, this subsystem manages processor related operations including: interrupt masking/unmasking; clock routines; and eventcounters.

Miscellaneous subsystem (misc) — The miscellaneous subsystem provides utility functions that do not have dependencies on other parts of the kernel; for example, generic string functions, 64-bit integer arithmetic, and indivisible counters.

Note that the path to a device driver goes through the fs subsystem first. The divisions within the fs subsystem show elements of particular interest to drivers. They are as follows:

Character and Block I/O — When an I/O-related system call comes in, the file subsystem separates requests by whether the device involved is a character type or block type I/O. Block I/O uses the buffer cache to collect and transfer data in large blocks while character or raw I/O operate on a character-by-character basis. Character versus block is the most basic breakdown of driver type.

STREAMS versus Standard Driver Interfaces — The two driver interfaces are logically below the separation between block and character I/O. A Standard driver can be either a character or block type, and drivers of each type are grouped under their respective I/O type. All STREAMS drivers/modules are considered a type of character special I/O and hence come in through the character channel.

Each subsystem makes its calls and structures available to other entities via an include file. To invoke a subsystem's functionality, you include its include file and call one or more of its functions. The subsystem's include file is named **i_*acronym*.h** where *acronym* is a two to four letter acronym for the subsystem (for example, **vm** for Virtual Memory subsystem). Table 1-1 shows some of the major subsystems, their acronyms,include files, and responsibilities.

Table 1-1 Subsystems and Their Include Files

Subsystem	Acronym	Include File
File system	fs	**i_fs.h**
I/O	io	**i_io.h**
Lock management	lm	**i_lm.h**
Miscellaneous	misc	**i_misc.h**
Process management	pm	**i_pm.h**
System control	sc	**i_sc.h**
Virtual memory	vm	**i_vm.h**
Virtual process	vp	**i_vp.h**
Micro-code	uc	**i_uc.h**

Subsystem include files are stored in **/usr/src/uts/aviion/ii**. The last 13 chapters of this manual list some of the function calls (kernel-supplied utilities) that you can use.

Using Conventions and Language Tools

A symmetric multiprocessing environment requires a high degree of coordination among kernel-level entities. In addition to hierarchy and information hiding, the principles of conventions and language tool usage also help promote coordination and orderly operation in the multiprocessor environment.

Conventions are used heavily throughout the DG/UX kernel in order to increase coordination and uniformity in the resulting product. A number of naming conventions are used to ensure consistency, enhance clarity and usability, and to reduce the chance of name collision. For example, prefixes and suffixes are used on C source file names to denote various attributes such as the owning subsystem or the category of data items contained. The conventions for identifier names and function names are similar but have the data type indicated as a suffix (for example, the suffix **ptr** is added to pointer names). Verb-object syntax is recommended for function names (for example **dm_create_link** instead of **dm_link_create**). Such conventions are very useful when reading and debugging code.

To avoid common coding errors that may prove difficult to debug, we also encourage the use of a number of coding conventions and language tools. For example, everything in the DG/UX kernel is declared specifically even when the C compiler default might produce the correct result. Similarly, compiler built-in data types such as **int** and **short** are not used because the realization of the types can vary on different machines. Instead kernel types are used that are defined in terms of the compiler built-in types. These kernel types are defined in the kernel include files, **c_generics.h** and **os_generics.h**. These files are found in **/usr/src/uts/aviion/ext**.

The use of **lint** type checking also helps flag type-mismatch errors early on in the development process. **Lint** makes heuristic checks for unused variables, variables that are set but not used, and variables that are used without being set. The checking done by this tool is similar to that done using function prototypes as specified by ANSI C and may be available on some compilers.

We will discuss the use of specific conventions throughout kernel-level documentation. We strongly encourage you to adopt the indicated conventions though in most cases they are not mandatory. Conventions should not be substituted for good judgement.

Interfacing to the Kernel

You can add a driver to the DG/UX kernel either through the STREAMS interface or through the Standard Device Driver interface. The details of each interface are given in their respective manuals. This section gives an overview of the basic structure of a kernel interface. The major components of an interface are:

- **A set of routines** — All drivers (and/or modules in the case of STREAMS) consist of a set of routines each of which performs a particular I/O or support operation. For example, the Standard Device Driver interface requires routines for open, close, read, and write operations and the STREAMS interface requires routines for open, close, read-put, read-service, write-put and write-service. To write a driver you write an appropriate set of routines for your device.

- **A data structure identifying the driver's routines** — The routines provide entry points for requests to the driver but the requestors must be able to find these entry points. Standard DG/UX drivers must supply a data structure, called a *routines vector*, that contains the names of its routines. The main kernel code invokes the driver by calling the appropriate routine in the driver's routines vector.

  Note that the kernel must have a routine supplied for every possible driver routine. However, in many cases you will be able to use a kernel-supplied routine stub and sometimes a kernel-supplied default routine instead of writing your own. Chapter 15 lists these routine stubs, called "nodevice" routine stubs.

  STREAMS drivers (and modules) have a variation on the routines vector theme. Access to STEAMS modules are achieved through a driver supplied data structure called a **streamtab** as described by the STREAMS documentation. STREAMS drivers can also supply several DG/UX routines including routines for initialization and configuration. STREAMS drivers need a routines vector to allow access to these routines. However, STREAMS modules/drivers have the option of using a default kernel-supplied routines vector with default initialization and configuration routines. You get the default routines vector by using an option in the master file entry described in Chapter 2.

- **A configuration entry for the driver including an identifying mnemonic** — All drivers must have a unique 2- to 8-letter mnemonic by which they are identified. For example, **sd** is the mnemonic for the DG/UX SCSI disk driver. The kernel uses the driver mnemonic to locate your driver during initialization and configuration. To let the kernel know your driver exists, you must create an entry with this mnemonic (and possibly other driver data) in the **master file** configuration file, as discussed in Chapter 2.

  Figure 1-3 shows how the three interface elements tie to one another. In the figure the driver mnemonic is **Foo**.

```
                              Foo                          Foo
  Master File         Driver's Rtn.Vector          Driver Routines
+----------+      +----------------------+       +---------------+
|  Foo --------> | Open Routine Address --> | Open Routine  |
|          |      +----------------------+       +---------------+
|          |      |Close Routine Address -->  |Close Routine  |
+----------+      +--.-------------------+       +---.-----------+
                  |  .    Routine Address -->  |    .  Routine |
                  +--.-------------------+       +---.-----------+
```

Figure 1-3 *The Three Steps in Locating a Driver Routine*

As explained in Chapter 2, the actual routing of control is a slightly more complicated version of this basic picture.

Programming Implications of the Fully Symmetric Environment

For the most part, you will not need to deal with the implementation of the fully symmetric environment itself. However, a few of the basic concepts will help you understand programming in a fully symmetric environment.

The kernel makes full symmetry transparent to kernel-level processes by running them on abstractions of physical processors, called *virtual processors* (VP). By using a VP abstraction, the hardware implementation (actual number of physical processors) becomes irrelevant to the higher levels of the kernel. The actual physical processor is called a *job processor* (JP).

A given configuration of the kernel will have a fixed number of VPs that is usually greater than the number of physical processors but less than the number of processes wanting to execute. A two-level scheduling scheme is used to balance between processes, VPs, and JPs. The lower level of scheduling multiplexes VPs onto physical processors so that the VPs appear to be active entities that execute code. This short-term scheduling is performed by the dispatcher. A higher level of scheduling multiplexes processes onto VPs so that the processes may execute. This higher scheduling is performed by the medium-term scheduler (MTS) using the operations defined on VPs. Figure 1-4 shows the general design.

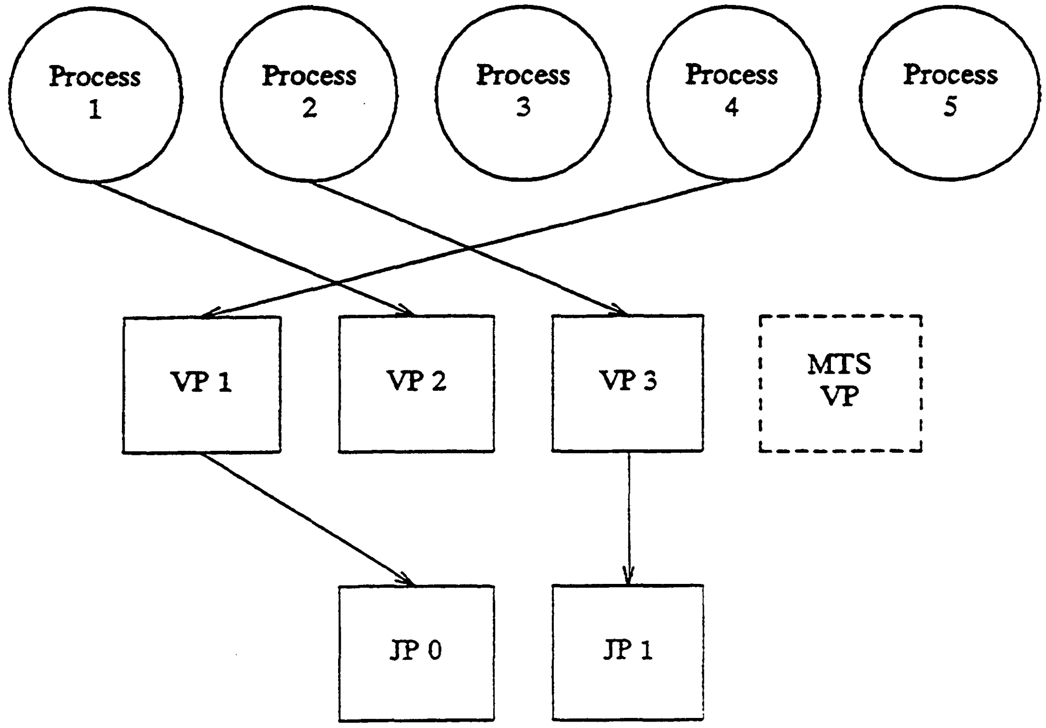


Figure 1-4 *Overview of the Fully Symmetric Design*

There are several general features of a fully symmetric environment that you must keep in mind as you do kernel-level programming. The following list gives an overview of the major features and their programming implications:

- **Many processes may be running concurrently on different processors.** Do not presume that, when your code has control, it is the only process running at kernel-level. Another process might be running on another processor and both this process and yours will have access to kernel data bases stored in common memory. This means that locking issues are very important. Because another process might seek access to a critical data structures, kernel-level processes must protect (lock) access to critical structures while they are modifying them.

- **You must lock critical sections of code or data and you can do so at a fine-grained level.** In order to maximize efficiency and because locking is so important when processes run concurrently, the DG/UX kernel provides special facilities to allow you to lock individual data structures. We discuss the lock facilities and the routines you use to access them in Chapter 4.

- **Interrupt handling is spread over multiple processors.** In the fully symmetric environment, a device's interrupts connect to all processors and the system picks which processor handles an interrupt on the basis of availability. Which processor handles a particular device's interrupt may even vary from interrupt to interrupt.

  Because of this arrangement of interrupt handling, you can enable and disable interrupts in two distinct ways: for a processor or for a device. Disabling a device's interrupts (masking its interrupt) prevents the device from interrupting any processor. Disabling a processor's interrupts prevents the processor from receiving interrupts from any device. Note that disabling a processor's interrupts does not affect the device; the device can still interrupt and the interrupt service routine can still run on another processor. A driver cannot disable processor interrupts to prevent collision with its own interrupt service routine; both could still access the same data structure at the same time. Similarly, note that masking does not guarantee that an interrupt is not already in progress.

Because more than one processor can service an interrupt, it is possible for a device to have two interrupt requests being serviced at the same time.

- **Scheduling even at the kernel-level is fully pre-emptive.** Fully pre-emptive scheduling is necessary to fully utilize multiple processor power. Fully pre-emptive scheduling means that the scheduler allocates process run-time through a combination of priority and round robin time-slice usage. Essentially, your process runs if no higher priority processes are waiting, and then the scheduler lets it run for a specified period of time and then gives control to another process at the same priority level.

  Though fully pre-emptive scheduling is standard in many operating systems, traditional UNIX kernels have been based on a simple, single-threaded control principle at the kernel level. Such kernels allow only one process at a time to run at kernel-level. This process could then assume that it would not lose control and that it had exclusive access to all kernel-level data bases. Such implementations are simple but very unresponsive to real-time interactions.

- **The eventcounter facility allows you to define and await multiple events.** Fine-grained locks and an eventcounter facility which supports waits on multiple process-defined events help many concurrently executing processes to coordinate their activity without collision. The eventcounter facility is discussed in Chapter 5.

Fully symmetric multiprocessing means that some programming practices that worked for the traditional UNIX kernels no longer hold. For example, we've mentioned that in single-threaded UNIX kernels, a process could assume that it would not lose control and that it had exclusive access to all kernel-level data bases. Similarly, in uni-processor UNIX kernels, disabling interrupts would guarantee that the current process has exclusive control (no interrupts) and exclusive access to all kernel-level data bases does not work in the fully symmetric environment. These assumptions do not hold on the DG/UX kernel.

General Hardware Concepts

This manual applies to drivers for all AViiON series machines that are running the DG/UX operating system. The different AViiON machines will have many common architectural features (for example, memory-mapped I/O) and some differences (for example, types and organization of primary and secondary buses). To encourage development of architecture-independent code, the DG/UX kernel was developed around a general hardware model that helps define the basic terms and concepts needed for drivers on all AViiON machines. The general hardware model has two basic peripheral levels: controller/adapters, and units (devices). Figure 1-5 shows how peripherals are connected to processors and memory.

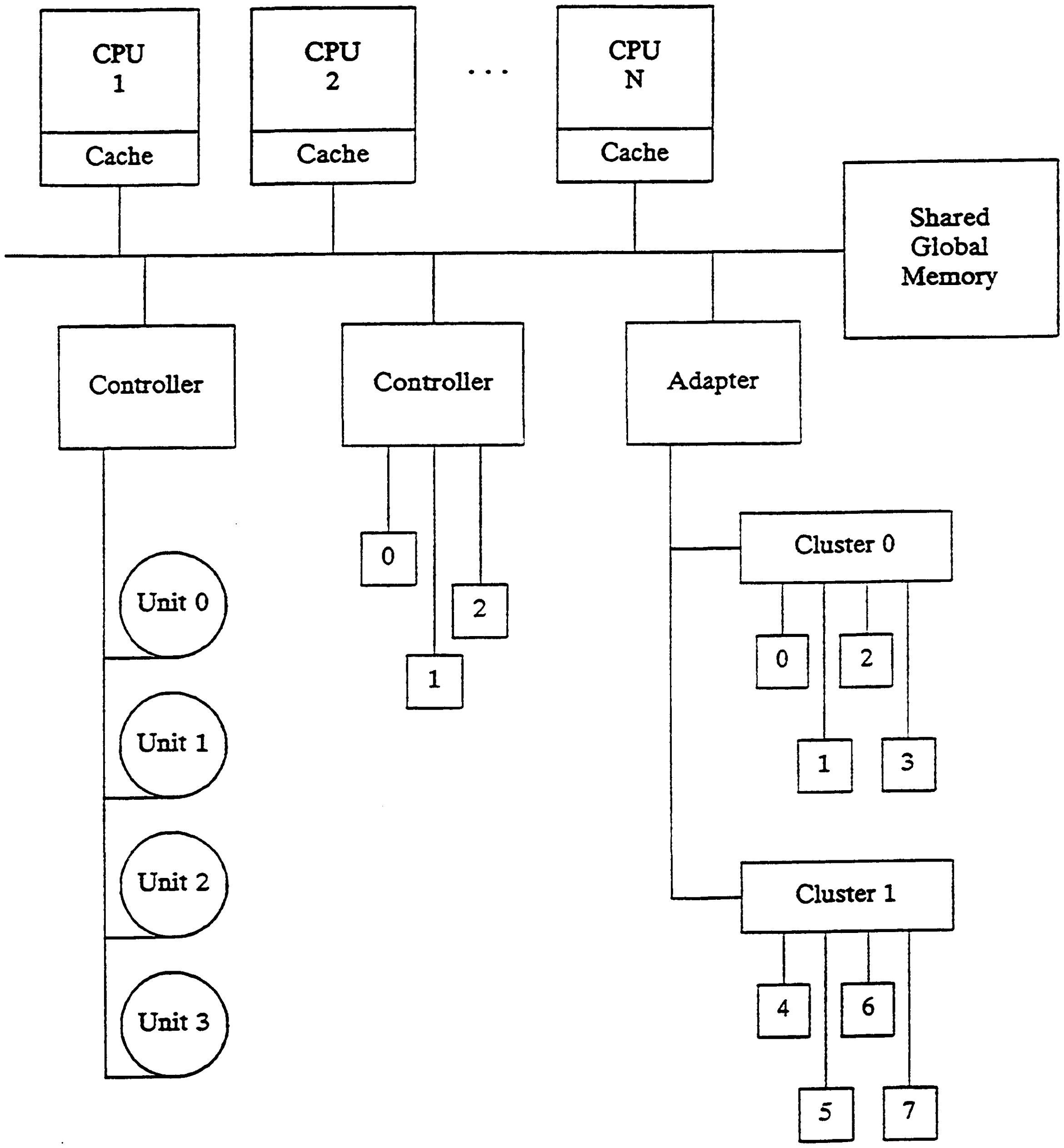


Figure 1-5 *A General Hardware Model*

Controllers and adapters — Controllers/adapters act as intermediate interfaces between the system's primary I/O bus and one or more I/O devices or units.

Within this general class, the term adapter refers to an I/O intermediary that manages an independent secondary bus. An adapter bridges between the primary system bus and the secondary bus and serves as a conduit between the CPU and devices attached to the secondary bus. An SCSI adapter supporting an SCSI bus with SCSI devices is an example of an adapter.

The term controller refers to an I/O intermediary that directly manages several lower level peripherals (usually all of the same type) without an independent secondary bus. A line controller supporting several asynchronous I/O lines is an example of a controller.

Only controllers and adapters can directly interrupt the system. A controller or adapter interrupt may be handled by any processor. A combination of hardware and kernel software determines which processor will receive a given interrupt. Drivers do not know which processor will be chosen and the assignment may change from interrupt to interrupt.

All processors have equal access to any controller or adapter to read status, to start I/O operations, or otherwise manipulate its state. Drivers do not know which processor will be accessing the controller/adapter and the processor may change from access to access.

Units (devices) — The term unit or device refers to a lower level peripheral attached to either a controller or adapter. Units do not interrupt the CPU directly.

For the most part, a controller's devices are simply considered to be sub-units of the controller. On the other hand, an adapter's devices have a degree of independence from the adapter because they are off a secondary independent bus.

Memory-Mapped I/O — On AViiON machines, drivers access their devices via memory-mapped I/O. This means you will read and write to specific areas of physical address space that are dedicated to your device. With memory-mapped I/O, assembly language programming becomes unnecessary because you can access your device using simple memory reference instructions.

For most devices, you set the device's memory-mapped I/O address by setting jumpers on the device itself. For devices it supplies, Data General pre-assigns and jumpers the memory-mapped I/O addresses to standard default addresses. However, if you add a non-standard device or a second instance of a standard device, you will have to jumper the I/O address on your hardware. More importantly, you will have to choose an address that is not already used by another device. Appendix A shows conventions and restrictions for choosing a memory-mapped I/O address. Appendix A also lists standard devices and their default addresses.

Direct Memory Access (DMA) — In memory-mapped I/O, a CPU accesses physical address space for the driver. In DMA, the controller transfers data between itself and main memory. Such transfers are based on physical main memory addresses and are cache coherent (that is, proper results are assured without performing any special operations on the cache memory). Controllers will typically have to perform scatter/gather addressing because a transfer may cross logical page boundaries and hence be in non-contiguous physical pages.

Keeping Interrupt Handling Machine Independent

Traditionally, a machine's interrupt structure is a major cause of machine-dependencies in device drivers. If your driver masks an interrupt using an interrupt identifier that is particular to one machine, you will have to modify code to run on different machines. The micro-code subsystem (**uc**) helps make drivers machine independent by doing most interrupt manipulations itself. The driver code calls micro-code routines to manipulate interrupts passing parameters that allow those routines to determine what interrupt is being addressed. To understand how the micro-code subsystem works, let us first look at examples of how machine differences affect interrupt determination in two AViiON machines.

Adapter, Controller, and Device Layouts on Different Machines

The definitions of adapter, controller, and device apply across AViiON machines. However, the layout of the different levels of peripheral will vary between machines. The type of bus by which a peripheral is attached is particularly important because it is the bus architecture that determines how interrupt identifiers are encoded and how interrupts are handled. For example, Figure 1-6 and Figure 1-7 show adapters, controllers and devices on the AViiON 5000 series systems and the AViiON 300 series stations, respectively.

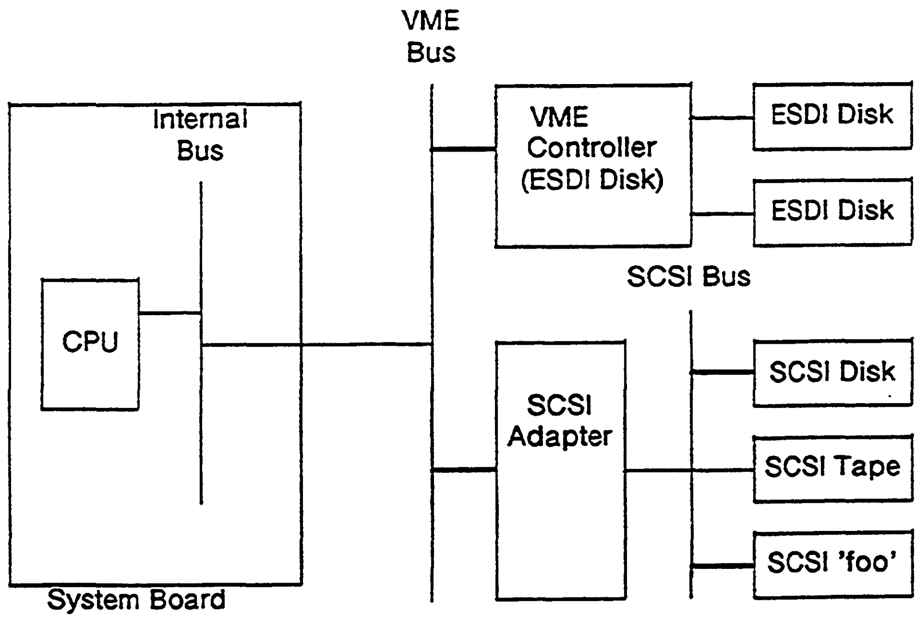


Figure 1-6 *Diagram of the AViiON System I/O Architecture*

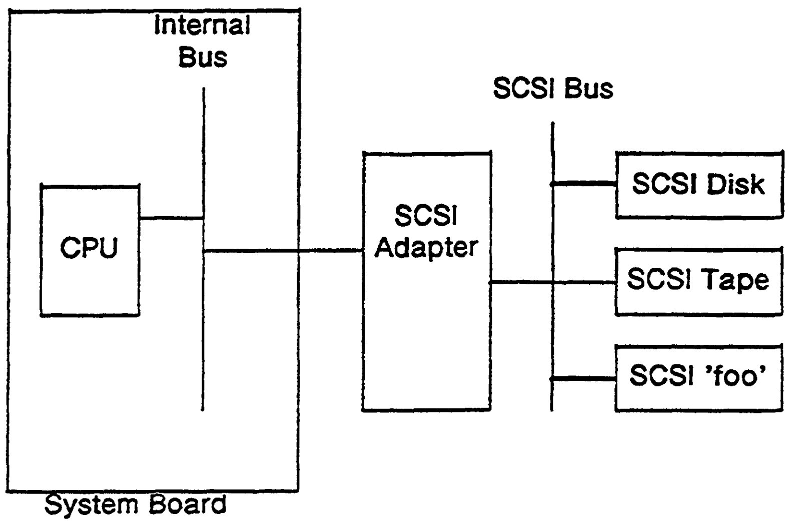


Figure 1-7 *Diagram of the AViiON Station I/O Architecture*

Note that while both machines support SCSI adapters, the SCSI adapter attaches to an integrated bus on the the AViiON 300 station and to an external VMEbus on the AViiON 5000 machine. Peripherals on an integrated bus have a fixed interrupt identifier while a peripheral's interrupts on a VMEbus is defined by its interrupt vector (see Chapter 8). Thus, an SCSI adapter on these two different machines would have two different interrupt identifiers.

Identifying Interrupts

The kernel helps drivers avoid machine-specific code by providing interrupt handling routines that manipulate (particularly mask and unmask) interrupts. Chapter 8 describes these routines. You specify the interrupt to be addressed via two parameters. One parameter gives the class of interrupts and the other gives a unique device identifier within that class. The interrupt class and device code parameters are defined as follows:

- **Interrupt (or Device) Class**

  The interrupt's class is defined by the bus on which the device is located. If the device is attached to a Data General proprietary bus integrated on the system board or a bus expansion slot off the system board, it belongs to the **Integrated** class of device interrupts. If it is attached to any bus other than these, the device class is defined by the particular bus. For example, the AViiON 5000 series systems support an external VME bus. All devices on this VME bus would belong to the **VMEbus** class of device interrupts. The kernel supplies an enumeration type that defines device class literals. Use these literals to specify your device's interrupt class. Chapter 8 discusses device class enumeration types.

NOTE: A device's class is defined by the bus to which it is attached. Thus, if a device is attached to a secondary bus serviced by an adapter, it is the adapter's bus that will define the device's class. For example, SCSI devices (those on an SCSI bus) are serviced by an SCSI adapter. On an AViiON 5000 series system, the SCSI adapter will be attached to a VME bus. Because only the adapter interrupts the CPU, the device class for the SCSI devices and the SCSI adapter is **VMEbus**.

- **Device Identifiers (or Device Codes)**

  Unique device identifiers within a device class are called device codes. Device code definitions vary with the device class.

  For devices in the **Integrated** class, the kernel supplies a set of literals for all possible types of devices found in the **Integrated** class. For example, you use the literal **UC_DUART_DEVICE_CODE** to identify any integrated duart device. Chapter 8 describes the device code literals for the **Integrated** class of devices.

  For other device classes, the device code is generally defined by a unique identifier jumpered on the board or set by the software. For example, on the AViiON 5000 series machines, controllers on the VME bus are jumpered to a particular vector number. The bus passes this vector number to the CPU and, when the device interrupts, it appears in the Interrupt Acknowledge register as the device identifier. Chapter 8 describes conventions for the **VMEbus** class device codes and Appendix A lists the standard VME interrupt vectors used on the DG/UX system.

  Note that your driver will be specific to a particular device class (for example, integrated or **VMEbus**). Thus, you will need different drivers for SCSI adapters in the integrated class (as on AViiON stations) and for SCSI adapters in the **VMEbus** class (as on AViiON systems). Drivers' dependency on device class results from the fact that device codes are interpreted differently in the different device interrupt classes.

End of Chapter

Chapter 2
How Information and Control Gets Passed Between Levels

This section describes a number of elements of kernel-level programming that relate to how needed information gets passed between levels.

Kernel-level code is written to serve user-level code. For portability reasons, user-level code needs a clean unchanging interface into the kernel. The system calls provide this clean interface. When a user wants to invoke a kernel function, it does so by calling the appropriate system call with arguments containing the necessary information.

This is the simple story of how information and control gets from the user-level to the kernel-level but it is not complete. There are three major additional aspects to the story:

1. **How the kernel gets the background information that it needs to handle the request.** For example, if the user is making an I/O request, the main kernel code must figure out what device driver should handle the request. And, once it gets control, the device driver must know various background facts about the device (like its hardware address and whether the operation is character or block I/O) to handle the request correctly. This information is not passed as part of the system call because the user-level code does not and should not have to know about such things.

   There are two primary tasks that must be done for the kernel to get the necessary driver/device background information: 1) preparing background device and driver information prior to run-time; and 2) registering necessary device and driver information with the kernel at run-time.

   Preparing background device information is a configuration task done by a system administrator prior to system initialization. It involves making sure that two files (master file and system file) accurately describe the system's devices (for example, how many and what types) and its drivers (names and types).

   Registering the necessary device and driver information with the kernel at run-time is done by the driver during system initialization. It involves storing various critical pieces of driver and device information in the appropriate kernel and filesystem data structures.

2. **How control is passed around within the kernel.** If the driver code can complete the operation immediately, kernel control simply goes from main-kernel to driver and back. This is called a synchronous operation; it is continous. However, if the operation cannot be completed immediately, the driver process must start the

operation, suspend awaiting some event (usually an interrupt), and then continue operation. In asynchronous operations, control is usually passed between various processes before it is completed.

The DG/UX system provides two daemons, the Driver and Generic Daemon, to help complete processing of an asynchronous requests after it is resumed. Asynchronous I/O requests generally require the use of an interrupt service routine and in a symmetric multiprocessing environment the interrupt service routine is only allowed to perform minimal operations. Driver Daemons and Generic Daemons provide an appropriate way to continue processing outside the service routine's restricted environment. Chapter 13 discusses the Driver and Generic Daemons and the routines used to access them.

Eventcounters are the main way control is passed between kernel-level processes. Eventcounters allow you to define and await your own "events". An eventcounter is a counter set to be incremented whenever a particular condition of interrest happens (say an interrupt). You can define an "event" by associating an eventcounter with a variable which contains a critical value. When the eventcounter reaches the critical value, the kernel awakens all processes waiting on the event. Chapter 5 describes eventcounters and the routines used to manipulate them.

3. **How the kernel-level code reports back detailed error and status information.** When the system call returns, it gives user-level code a completion status that roughly describes any errors. The driver frequently has more information about the error than it can return in a generic completion status which is defined on a system-wide basis. Also, it is often the system administrator, not the user-level program, who is interested in detailed status information. Thus, the DG/UX kernel allows drivers to log detailed error messages to a system error logging facility that both system administrators and user-level code can access.

The rest of this chapter is devoted to discussing these three areas.

Setting-Up Background Information

In order to handle a user request, the kernel needs information about the device so that it can invoke the appropriate driver and so the driver can access the device properly. But, most users will not know the detailed information about their device, nor should they have to. How does one handle these competing needs so that the user gets hooked up with the right driver and the right device without having to do too much work? The answer includes: system file entries, master file entries, configuration routines, device special files, and various kinds of device or driver identification parameters such as device codes. We will start our explanation of how the kernel gets from the user to the driver to the device with the user side of things which means looking at device special files.

How the Kernel Gets From the User to the Device: Device Special Files

The user-code view of a device starts with that device's special file. A *device special file* (also called a node) is a special descriptor file that contains information that the kernel needs to get to the driver and that the driver needs to identify the specific device. When a user-level program wants to access a device, it performs an open using a device special file as a parameter. The kernel uses information stored in the special file to find and call the driver's open routine. The kernel then returns a file descriptor that the user-level program will use in all its remaining device requests.

For example, a disk has a special file called *node_name*. To open the device and get a file descriptor, the user program issues the following DG/UX system call from a C program:

```
int fd;
fd = open ("/dev/dsk/node_name", O_RDWR)
```

The kernel passes the request to the disk driver's open routine and returns a file descriptor in **fd**. The user will now use **fd** to access the specific device. For example,

```
read (fd, Buffer, 20);
```

causes the kernel to read from the device identified by the special file *node_name*. **Fd** points to this file, and 20 is the number of bytes to be read into the memory area denoted by the **Buffer** variable.

The device's special file contains information that kernel-level code needs to complete the user's request. The information it contains is:

- **Type of I/O interface (Block or Character)** — When control comes into the fs subsystem, it passes control to its lower levels based on whether the device is to be treated as block or character I/O. This information is then used both in finding the right driver and in telling the driver what type of I/O to perform.
- **The major number** — The major number is used as an index into the kernel's device driver tables and thus allows the kernel to find the appropriate driver. Hence, device special files with the same major number are all serviced by the same driver.
- **The minor number** — The minor number identifies a a specific unit of the family of devices being addressed. The driver uses the minor number to keep track of requests for particular devices.

Users can create special files with **rc** scripts and with the **mknod**(1) command. However, most device special files are created by the various device drivers at configuration time. Of course, to create special files for its devices, the driver must have information about what devices are on the system. Setting up this information is part of the pre-initialization setup task.

Getting Set Up Before and During System Initialization

As part of setting up the system prior to initialization, the system administrator must create: 1) an entry for the driver in a file called the master file; and 2) entries for each specific device in a file called the system file. Master file entries contain driver information that kernel initialization code needs and system file entries contain device information drivers need to build special files. The DG/UX master file already contains entries for Data General-supplied drivers. Appendix B describes how to create master file and system file entries.

At boot time, the system's initialization code assigns a major number for each driver listed in the master file. After initialization, most kernel operations will use the major number, not the device mnemonic, to identify the driver. During initialization, the main kernel initialization code will also call each driver's initialization routine. Initialization routines are generally used to initialize data areas and hardware that must be prepared prior to configuring a specific device.

After passing through the master file, the main kernel code begins the configuration process by passing through all the entries in the system file. It will call the driver's configuration routine for each entry. The configuration routine should perform all operations needed for the specific device to be opened. This usually includes: 1) creating a special file for this specific device; and 2) registering information about this specific device with the kernel.

The driver's configuration routine now has all of the information it needs to create a special file for the device. The main kernel code assigns a major number and passes it to the driver's configuration routine as a parameter. Drivers use minor numbers for their own internal bookkeeping. Thus, the driver can create and use the minor numbers for its devices in any way it wishes. The system file entry indicates whether the device is character or block I/O.

All the information that goes into the special file is available at configuration time. To create a special file however, the driver's configuration routine will have to name the special file. Special files are named using the device's *device specification*, a unique system identifier for the device. You use the device specification to name the special file, and thereafter the kernel will use this name string as a device identification parameter. The next section describes device specifications.

Special files are stored in the **/dev** directory, usually in sub-directories that hold special files for the same types of device. For example, the disk special file called *node_name* is located in **/dev/dsk**.

Identifying a Device: Device Specifications

All devices (controllers, adapters, and devices/units off controllers and adapters) must have a unique software descriptor called a *device specification*. While you may define aliases, generally the actual name of a device's special file is its device specification. (Appendix B of *Installing the DG/UX™ System* discusses device specifications in more detail).

The kernel passes device specification strings to a driver for interpretation. Drivers supply a name-to-device routine that the kernel can call whenever it needs a device specification parsed. (This is one of the routines supplied in the default routines vector for STREAMS drivers.) You could write your driver's name-to-device routine to use a device specification syntax different from that used by drivers supplied with the DG/UX system. However, for intelligibility and to avoid collision, you should implement your device specification like DG/UX drivers which use the following device specification syntax:

```
device_mnemonic [@device_code] ([parameters])
```

where:

`device mnemonic` is the two- to eight-letter mnemonic used to identify the device driver in its master file entry. Appendix A shows the device mnemonics for device and adapter drivers supplied by Data General.

`device code` is a device identifier that uniquely identifies a physical device within its interrupt class. For devices with device codes, you enter the device code as a hexadecimal number preceded by an @ (at) sign (for example, @18). Device codes are defined within each specific device class. However, only devices that directly interrupt the host have device codes. Devices that do not have device codes (such as pseudo-devices or SCSI devices off SCSI adapters) must omit the device code field in their device specification.

`parameters` are values that provide additional information to the driver. The parameters for the device specification depend on the type of device and whether the device is a controller, adapter, or device (unit).

Controller and Adapter Parameters

The device specification for an adapter consists of the adapter's name, its device code, and a single parameter identifying which adapter is being addressed. The device specification for a controller consists of the controller's name, its device code, a parameter identifying which controller is being addressed, and a second parameter specifying which device off the controller is addressed (for example, unit #1 off the controller).

For both controllers and adapters, the first parameter indicates which controller or adapter is being addressed. For drivers supplied with the DG/UX system, you can identify which controller or adapter is being addressed in either of two ways:

- You can specify the controller or adapter by giving its base memory-mapped I/O address. For example, the first **cied** adapter would be **cied(fffef00)**. The **cied** mnemonic stands for Ciprico ESDI disk. Appendix A lists the base addresses for drivers supplied with the DG/UX system.
- If the controller or adapter is located at one of the standard base addresses for a device of its type, you can omit the address and just use the number for the controller/adapter at that address (see Appendix A). For example, you can use **cied(0)** and **cied(1)** to specify the first and second **cied** controllers. If you omit the first parameter, the driver should assume a value of zero. Drivers supplied

with the DG/UX system can deduce the base address from this information.

NOTE: You cannot use this form if you are addressing a controller or adapter whose base address is not a default.

SCSI Device Parameters

For SCSI devices, the first parameter indicates on which adapter the device is located. You identify the adapter with its device specification as just described. For example, the device specification for the SCSI disk off the AViiON station integrated SCSI adapter would be **sd(insc(0),2)**.

The second parameter is the device's SCSI ID. The SCSI ID is a bus identifier jumpered on the device. A device's SCSI ID must be unique on its adapter but not across adapters. (Appendix B of *Installing the DG/UX™ System* lists the default SCSI IDs for standard devices on the DG/UX system.)

The device specification has a third parameter that you can use to specify a unit number if the SCSI device is a controller with multiple units.

NOTE: In device specification, device codes and base addresses are interpreted as hexadecimal numbers. You must *not* precede them with "0x" as is conventional in C language programming.

NOTE: In all cases, an omitted parameter is treated as if it were zero.

The following are valid device specifications:

cied()	**cied** disk controller with all parameters assuming their default values.
cied(0,1)	Drive 1 on **cied** disk controller 0.
cied@77(fffff500,0)	Drive 0 on the **cied** disk controller at the non-standard base address 0xfffff500, with the non-standard device code 0x77.
sd(cisc(1),2)	The SCSI disk at SCSI ID 2, reachable through SCSI adapter 1.
st(insc(0),2)	The SCSI tape at SCSI ID 2, reachable through integrated SCSI adapter.
sd(cisc@77(fffff500),2)	Disk drive 2 on the **cisc** SCSI adapter at the non-standard base address 0xfffff500, with the non-standard device code 0x77.

Registering Device Information

The final stage in preparing a device to be opened is registering device specific information with the kernel. The driver is responsible for tracking the progress of operations on its devices and must do its own bookkeeping. However, the main kernel code does require the driver to register device specific information in the system's device information table (DIT).

To register information, your driver's configuration routine calls the kernel's **io_register_device_info** routine giving a pointer to a data structure as one of the parameters. The layout of the data structure, called a *device information structure*, is largely up to the driver. However, if the device receives interrupts, the first field of this structure must be a pointer to your interrupt service routine. Registering interrupt handlers is the main reason for registering device specific information.

Once the device information structure is registered, the kernel passes a pointer to it as a parameter when it calls the driver's interrupt service routine. In this way, both the driver and the main kernel code have access to basic information about a specific device.

Reporting Errors

Drivers can choose between two major error-reporting destinations: 1) the user-level calling process; and 2) the system error logging facility. Error reporting facilities are described in more detail in Chapter 13. This section provides a brief overview of the facilities.

Drivers do not need to perform any special operation to report statuses back to the user-level process. The kernel passes return values from the driver routine to the user as a completion status. Because users receive return values as statuses, we strongly recommend you encode your driver's unique return values according to standard encoding procedures. Users can decode standardly encoded statuses using the **dg_ext_errno** system call.

If the driver encounters a significant error during its processing (a device failure, for example), it may want to flag this condition to the system administrator. To send an error to the system error-logging facility, the driver must use the services of the system error daemon, **syslogd**, and the pseudodevice, **err**(7). **Err** receives and stores errors from kernel-level processes. **Syslogd** receives and stores errors from all processes connected to the system, remote or local, user- or kernel-level. **Syslogd** periodically retrieves and processes the errors stored in **err**.

How **syslogd** processes errors is determined by its configuration file, **/etc/syslog.conf**. For example, **syslog.conf** may specify that the logged errors are to be printed out to the system console or written to a disk file or other device. See **logger**(1), **syslog**(3), **syslog.conf**(5), and **syslogd**(8) for more information on the system error daemon and how to configure error processing.

The **err** pseudo-device receives and stores errors from drivers on an internal error queue. Your driver can store error messages on this queue using the kernel-supplied routine, **io_err_log_error** (see Chapter 13).

End of Chapter

Chapter 3
Ground Rules of Kernel-level Programming

This chapter discusses a number of facets of kernel-level programming including memory types and allocation, handling interrupts, and using signals. It also provides some general guidelines for writing device drivers.

Driver as Part of the Kernel

The most important fact about a driver is that it is part of the kernel. This means a driver has access to all of system memory and to all devices. Kernel code is protected from write access (note that this protection means a driver cannot use self-modifying code), but no other protection is provided against a driver writing to kernel databases and/or otherwise destroying the kernel internals.

Because of its special status as part of the kernel, a device driver may not use the standard C libraries or DG/UX system calls (described in Chapters 2 and 3 of the *Programmer's Reference for the DG/UX™ System, Volume 1*).

Device driver code may be executed as part of the calling user process, part of the kernel-level system call process or as part of another kernel process such as the Driver or Generic Daemon processes. Driver code executes on the kernel stack of the running process. The kernel stack is of fixed size, so driver code must not nest calls too deeply. A system panic results if a process's kernel stack overflows. Panic codes are listed in a file in **/usr/release**; your DG/UX system Release Notice discusses this file.

Elements of Memory Layout and Allocation

This section describes memory concepts, layout, and operation on the DG/UX system. The most basic breakdown of memory is between logical and physical addresses:

Physical address space — The size of physical address space is determined by the number of bits you have to specify an address. AViiON machines have 32-bit words and thus a 4GByte physical address space.

Physical memory — The size of physical memory is based on the size of machine's physical memory. For example, on an AViiON 300 that has 16 Mbytes of physical memory, the physical space would be from 0 through 16 Mbytes.

Control Space or Memory-mapped I/O Areas — The control space, starting at address 0xffc00000 to the end of memory, contains the memory-mapped hardware registers and buffer areas. This is where most drivers access their devices. Memory-mapped hardware registers and buffer areas are not part of regular physical memory though they are addressed like memory locations.

Physical addresses — Physical addresses refer to unmapped, absolute physical addresses that identify either physical memory locations and/or hardware registers found in control space. For example, as Figure 3-1 shows, on an AViiON 300 there are valid physical addresses for the 16 Mbytes of physical memory and for the control or memory-mapped space.

Logical addresses — Logical addresses are the virtual addresses with which your program operates. Logical addresses must be associated with a physical address when they are used. They span the logical address space which is 4 gigabytes on AViiON machines.

Mapping Logical Addresses to Physical Addresses — The hardware registers for your device are located at a specific physical memory address. Yet, when you read and write to memory, you are always addressing logical address space. To map (associate) a logical address to a specific physical address, use the kernel-supplied map and unmap routines described in Chapter 9. You will not need to map your device's registers if your hardware's I/O area falls in the control area because the control area is mapped to corresponding physical addresses by the kernel initialization code.

Using the Volatile Compiler Directive — When you read a memory-mapped location, you want to know that the value present represents the current state of the hardware. For the sake of speed, optimizing compilers minimize the number of accesses to memory. Thus, a memory access is not necessarily made every time a program addresses a variable. You must use the **volatile** compiler directive for variables assigned to memory-mapped location to ensure that the value is current. The **volatile** directive tells the compiler that the variable should be updated before every access. The **volatile** directive is part of type specification in **typedef** and variable declarations. For example:

```
typedef volatile unsigned log hdw_reg_type
typedef volatile struct
                 { int a;
                   int b;
                 } ab_vol_type
volatile int j;
volatile enum {mon, tue, wed, thu, fri} weekdays;
```

Note that casts can cancel the effect of **volatile**. See your compiler's documentation for more information on **volatile**.

The system maps logical address spaces in and out of physical memory which results in several additional concepts:

Page — A page refers to an area of 4096 bytes of memory. Generally this is considered to be a piece of logical address space.

Page frame — A page frame is the physical area on which a page is loaded or mapped; a page frame has a physical address.

Unwired memory — Unwired memory is memory that can be paged out to disk. A page fault must occur before the memory can be accessed after it has been paged out to disk.

Wired memory — Wired memory is memory that cannot be paged out to disk (see *wired page*). Wired memory is critical for certain kinds of data and text. For example, any kind of data required to service a page fault must be in memory when a page fault occurs. When you link your code into the kernel image, the text and static data goes into wired memory by default.

Wired page — A wired page is a page that is bound to a page frame. Wired pages cannot be paged out to disk until they are unwired.

In addition to addresses, memory has address spaces that enforce the separation of User and Kernel levels with the following result:

User address space — User address space refers to memory accessible to the owning user process. The kernel can also access this memory, but in general, other user processes cannot.

Kernel address space — Kernel address space requires Supervisory Access Privileges to access. It is accessible to the kernel and not, as a rule, to user processes.

Figure 3-1 shows the net result of all of these addresses and address spaces:

```
                   0                                       4GBytes
User Logical       |________________________________________|
Address Space      |                                        |


                   0                                       4GBytes
Kernel Logical     |________________________________________|
Address Space      |                                        |


                   0        16MBytes                       4GBytes
  Physical         |____________|<-- no memory -->|_________|
Address Space      |Physical    |    invalid      |Control  |
                    Main Memory                    Space
```

Figure 3-1 *User, Kernel, and Physical Address Space*

Note that in all cases the address space range is 4 Gigabytes which is the range covered by a 32-bit address. Note also the separation of main system physical memory located at the lower range of the address space and control space which is located at the upper end of the address space.

Basics of the Multiprocessor Address Space

On AViiON machines, all Job Processors (JPs) use the same physical memory. This physical memory includes a kernel address space and multiple user address spaces. Job Processors (JPs) have pointers to these address spaces. Figure 3-2 shows the major elements of the multiprocessor address space.

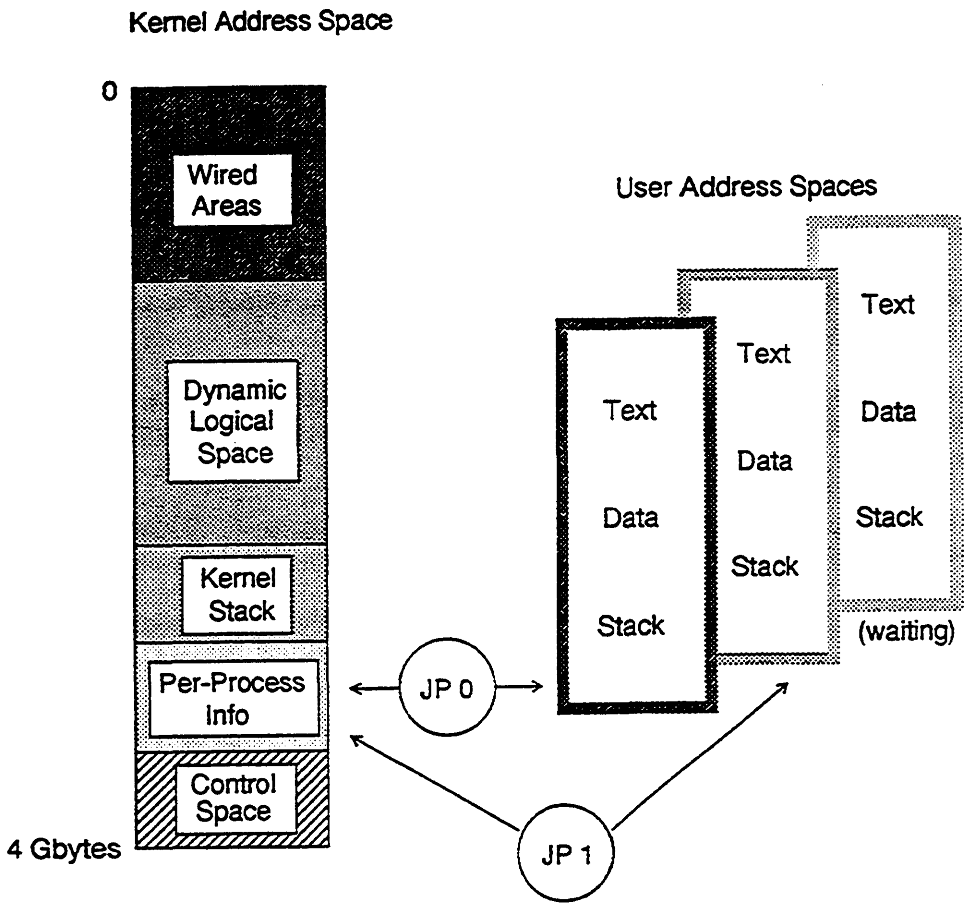


Figure 3-2 *DG/UX 4.30 Address Spaces*

As shown, each JP has two pointers, one pointer to the kernel address space and one to a user address space. On a multiprocessor system, all JPs point to the same kernel address space, but each JP points to a different user address space.

As shown, the kernel address space has five major areas: wired space, dynamic logical space, kernel stack, per-process area, and control space. We've already described the control space. The rest of the areas are:

Wired Space — The wired area shown in the figure contains the main body of kernel text and static data that remains resident in the lower part of physical memory. Your driver's text and static data are included in this area so that they can be accessed from interrupt handlers.

Dynamic Logical Space — The dynamic logical space holds any unwired kernel text or data. Most importantly for drivers, this space also contains a general kernel memory pool.

Memory Pool: The memory pool contains memory from which you can dynamically allocate either wired or unwired data. You use kernel-supplied utilities (for example, **vm_get_wired_memory**) to get and release such memory. Chapter 9 describes the routines used to allocate and deallocate memory.

Kernel Stack — A user process has its own stack in its user space for system calls that do not enter the kernel and a kernel stack for calls within the kernel code. Although all user processes use the same kernel code, every user process has its own kernel stack. The kernel stack has a unique physical page for every user process, but the kernel stack is mapped at the same logical kernel address.

Per-process Information — Per-process data is similar to the "u-area" in traditional UNIX kernels. It holds state information for a specific process. Per-process*or* (or per-JP) data is like per-process data except for a processor.

In order to reduce implementation dependence, DG/UX per-process data is organized differently from the traditional UNIX kernel. In the traditional UNIX kernel all u-areas were declared in one global process table. Consequently, if anything changed in any individual u-area, the entire kernel had to be rebuilt. The DG/UX kernel separates procedure tables by subsystem. Each per-process area (u-field) is declared as an independent per-process variable in the subsystem that owns the field. Per-process variables are put in a special linker section, and are collected together at link time. Thus, the DG/UX equivalent of the System V u-area is built at kernel link time instead of at compile time. When you integrate your driver code you will not have to re-compile the entire kernel. Figure 3-3 shows the traditional versus DG/UX implementation of process (**proc** tables).

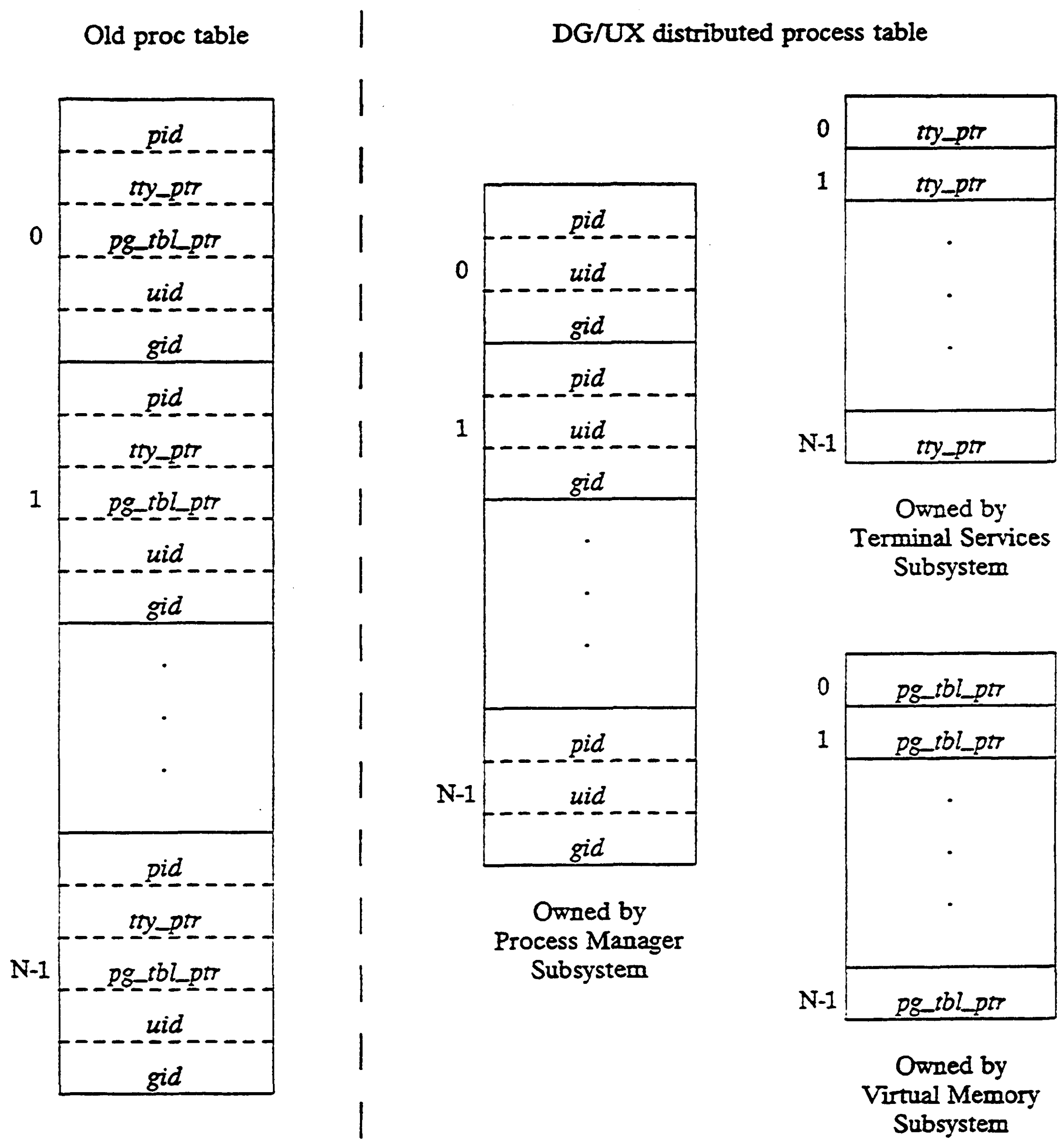

Old proc table
DG/UX distributed process table
pid
tty_ptr
pg_tbl_ptr
uid
gid
0
1
N-1
Owned by Process Manager Subsystem
Owned by Terminal Services Subsystem
Owned by Virtual Memory Subsystem

Figure 3-3 *Traditional and DG/UX Proc Tables*

By convention variables in the per-process area are "my" variables and are named *subsystem*_**my_---**, for example, **sc_my_process_index**. You get the information traditionally found in u-areas by calling kernel-supplied routines that return this information (for example, see Chapter 7 for **pm_get_my_pid**). By using routine calls to get this information, your code becomes independent of the way in which that information is stored which in turn leaves the implementation freer to change.

Because per-process data is only mapped to physical memory when the process is active, it is not available when the process is not running. Some kernel-supplied routines supply per-process information as return values. If a parameter is per-process, you will only be able to reference it when the you are in the calling process. You will not be able to pass such parameters to the Generic or Driver Daemons or to interrupt level because these levels do not operate in the calling process' context.

Handling Interrupts

If your device generates hardware interrupts, your driver must supply an interrupt service routine (interrupt handler) to service those interrupts. When an interrupt occurs, the kernel will read the Interrupt Status register (IST) and pass control to the driver's interrupt service routine as registered in the DIT (see Chapter 2). Once it receives control, the interrupt service routine must clear the interrupt if reading the IST did not clear it. For VME devices, reading the Interrupt Acknowledge register acknowledges the interrupt, and on many devices (Release-on-acknowledge devices) this action also clears the interrupt. However, some devices require additional action to clear the interrupt. Consult the documentation for your device to see when and how your device stops asserting interrupts.

The interrupt service routine must operate in a severely restricted environment. The interrupt service routine will run with all interrupts disabled on the current processor (interrupts on other processors are not affected). It is expected to quickly determine what action to take (usually advancing one or more eventcounters) and then dismiss the interrupt. It must not pend or page fault. To avoid page faults, the service routine should not reference unwired memory. It should also avoid calls to routines that might pend or page fault. Each kernel routine described in Chapters 4 -16 indicates whether or not it might pend or page fault.

Interrupts do not nest in the DG/UX system, so each interrupt handler must quickly finish its job and return to base level. Furthermore, interrupts are handled on the kernel stack of the currently running process; no separate interrupt stack is used. Therefore, the interrupt service routine must limit the amount of stack space used by it and any procedure it calls.

Clearing the interrupt frees the device to issue another interrupt. Because another interrupt may be serviced by another processor, it may be handled before the first interrupt service routine has completed.

Most device driver code executes with interrupts enabled. The driver should not manipulate the state of the interrupt enable register unless absolutely necessary. If the driver must change the interrupt state, it should use the kernel's interrupt enable and disable routines (described in Chapter 8).

Using Signals

Before a device driver waits for an indefinite amount of time for an I/O operation to complete (such as on a read of a user keyboard), it must prepare to receive a signal by calling the appropriate kernel functions. If a signal should occur, the driver must abort the operation and return an appropriate status.

For devices that do not normally require user intervention for an I/O operation to complete (such as a disk), signals do not have to be handled while waiting for the device to respond. The device must, however, be timed out if it fails to respond within a few seconds so that the calling process will not become hung indefinitely if the device should lose power or otherwise fail.

Higher levels of the system are responsible for providing reasonable response to signals. These higher levels may break large user requests into smaller, driver-level requests so that signals are not ignored for too long a time. For example, if a user requests that 100 Mbytes be written to the disk, the driver may see only a succession of 256 Kbyte requests. A device driver need not be concerned about the size of a user's request as long as it is making progress on the request and is not depending upon some indefinite external event for continued progress.

How the DG/UX Kernel Shares Data

Information hiding is one of the guiding principles in the DG/UX kernel. By convention, all data in the kernel is private to the subsystem of which it a part. Instead of exporting a data item, a subsystem exports procedures that define the allowed set of operations on the data item. Every item "owned" by a subsystem effectively eliminates global data from the kernel.

Having clear ownership of all data is extremely important in achieving multiprocessor operation. In a multiprocessor kernel, data structures must be locked before being accessed in order to ensure that the contents are consistent. Ownership restricts access to the few owning functions so that the lock and unlock operations can be clearly and consistently applied. It also allows for the use of finer-grained locks and the briefer use of locks than in global data kernels.

Setting Up Your Driver's Interface

For drivers, the **routines vector** mentioned in Chapter 1 constitutes the primary exported interface. For standard DG/UX drivers, the routines vector should be named **cfv_*xxx*_routines_vector** where *xxx* is the driver's master-file mnemonic. The STREAMS driver's **streamtab** structure functions as the major exported interface and should be named ***xxx*info**. STREAMS driver's may also supply a routines vector for initialization and configuration routines. For STREAMS drivers, you set whether or not you will supply a routines vector (versus using the default system vector) in your master file entry described in Appendix B.

To further support modularity we recommend you use the following groupings as well: a driver header file, a global data file, and a C code file. The header file holds the drivers constants and data structure definitions and is named **dev_xxx_def.h**. The global data file holds the driver's statically allocated global data including the routines vector. The global data file should be named **dev_xxx_global_data.c**. The C code file contains the C code text and local data. A standard driver's C code file should be named **def_xxx_driver.c** and a STREAMS driver's C code file should be named **sfm_xxx_driver.c**.

Building Your Driver Into the Kernel

Code that is imported into the kernel from outside Data General is defined as user text and user data. This user text and data is not built with the same procedures used on internal kernel code which generates specific text sections. Your compiled and assembled code will produce an **.o** file with three sections called **.text**, **.data**, and **.bss**. The **.text** section is your code; **.data** is initialized data; and the **.bss** section contains uninitialized variables.

The procedures for building a new kernel and checking your driver's configuration are described in Appendix C.

End of Chapter

Chapter 4
Lock Management Routines

This chapter describes all DG/UX kernel routines used in implementing locks on critical sections of data. We start with a brief introduction to locks and the locking routines. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_lm.h** for structures beginning with the **lm** acronym) for a complete and current list of all constants and structures.

Overview to Using Locks on the DG/UX System

The kernel lock facilities are used to protect critical sections of code. If more than one process is executing the same code and/or accessing the same data at the same time, the data may become corrupted. The lock facilities guarantee exclusive access to the code or data covered by the lock while the lock-holder is executing in that region.

Most operating systems use locking facilities. However, locking is particularly important in the DG/UX kernel. Unlike traditional UNIX kernels, the DG/UX kernel provides fully pre-emptive scheduling. This means that a process might be suspended while updating a data base and another process that accesses the same data might be given control. In addition, because the DG/UX system runs in a fully symmetric environment, you can't disable interrupts for protection as has often been done in single-processor UNIX kernels. Such disabling only affects one processor — other processes may be running on other processors. To further enhance performance with multiple processors, the DG/UX kernel also provides fine-grained locks that protect individual data bases rather than the traditional approach that locks the entire kernel data base at once.

All locks provide mutual exclusion. However, several types of locks are available each of which provides certain additional features. Different locks also vary in performance and in the memory required to implement them. The three types of locks the DG/UX kernel provides are: sequenced locks, unsequenced locks, and spin locks.

There are three routines for each type of lock: a routine to initialize the lock; a routine to obtain the lock (start locking); and a routine to release the lock (stop locking). Note that routine names consist of the operation and lock type plus the kernel subsystem (and, hence, include file) where the routine is located — for example, **lm_obtain_sequenced_lock**. Each type of lock also uses its own corresponding data structures.

To use a lock, you first allocate space for it and then call the appropriate initialization routine. You then call the obtain routine. Once this call returns, you hold the lock and no other process can have the lock until you call the corresponding release routine. Sometimes someone else may already hold the lock you want when you try to obtain it. This situation, called *contention,* is handled differently depending on the type of lock involved. In fact, what happens during contention is one of the major differences that define the different locks. So, before discussing this situation, we'll describe the different types of locks.

Spin locks are the simplest type of lock. They cause the caller to loop within the call until the lock can be obtained. This looping, called a "busy wait," consumes a lot of CPU resources because the process continues to run on the physical processor until the lock is available.

Spin locks are very dangerous because the potential for deadlock is high. Obviously, to release a lock the owner of the lock must be able execute the release routine for that lock. If someone else busy waits for a spin lock on the same processor that is running the lock-owner process, they will tie up the processor and the lock owner will not be able to call the release routine. This is called *deadlock*.

Because of the potential for deadlock spin locks should generally be avoided. If you must use them, you should use them only for protecting very small critical sections of code (only a few instructions in length). You should also make sure that the process cannot lose the processor on which it is running while holding a spin lock. This means that the process cannot take any action that might require it to be removed from the processor including taking a page fault. Thus, the lock itself must be allocated in wired memory and you must only reference wired memory while holding the lock. Finally, while holding a spin lock, you must also ensure that interrupts are disabled.

The sequence to gain exclusive access to a resource protected by a spin lock is as follows:

```
vp_disable_interrupts();
misc_obtain_spin_lock(&some_resource_spin_lock);
  .
  .
misc_release_spin_lock(&some_resource_spin_lock);
vp_enable_interrupts();
```

You disable interrupts before attempting to gain the spin lock. Then, if the lock is not available, it can only be because another process on a different processor is holding the lock. If this happens, your process, by owning its home processor, will spin until the lock becomes available. Such busy-waiting is a major reason why spin locks should be held only for a short time.

You will use sequenced or unsequenced locks for most locking needs. Neither sequenced nor unsequenced locks use busy waiting; so the holder of the lock can give up the processor. They differ in how waiting is done.

Sequenced locks grant access on a first-come-first-serve basis. They avoid the scheduling overhead by ordering contending processes based on when they first tried to obtain the lock. When the lock is released, only the next process in line is awakened.

Unsequenced locks are faster and take less space and CPU time than sequenced locks. When you call to obtain an unsequenced lock, the kernel removes your process from the processor until the lock is available (that is, when the current owner releases it). They provide no ordering of requesters.

Unsequenced locks, however, may not perform well under high contention, because they can cause a cascade of rescheduling. When the lock is released, ALL the processes waiting on the lock are awakened (made runnable again). At some point after they start running, they will attempt to obtain the lock again. One of them will be first and will succeed in obtaining the lock. The rest will find the lock already locked and will be put back to sleep until the new owner releases the lock and the sequence of events repeats itself. The resulting cascade of awakenings and reschedulings creates a high cost in system time.

Unsequenced locks might also have a "livelock" problem. If new processes are always trying to get the lock, a process might recurrently fail to obtain the lock, and thus wait a long time to make forward progress. The process would not be dead, but would essentially be looping trying but failing to get the lock and continue.

Sequenced locks avoid these problems, but at a cost. The cost is in performance (obtaining and releasing them is slower and takes more CPU time) and space (a sequenced lock takes more space to implement). The cost results from the queue of waiting processes that is associated with a sequenced lock. When a process must wait on the lock, it is entered onto the end of a FIFO queue of waiting processes. When the lock is released, the system wakes up only the process at the head of the list and this process obtains the lock.

Both sequenced and unsequenced locks provide no-wait versions of their obtain routines that return control immediately. The no-wait routines either return with the lock now locked on the caller's behalf, or with an indication that the lock could not be obtained. The no-wait version allows a process to handle other operations and try to obtain the lock again later.

For a more detailed discussion of locks on the DG/UX system, see the reference listed in the Preface of this manual.

The routines described in this section are as follows:

- **lm_initialize_sequenced_lock**
- **lm_initialize_unsequenced_lock**
- **lm_obtain_sequenced_lock**
- **lm_obtain_sequenced_lock_no_wait**

- **lm_obtain_unsequenced_lock**
- **lm_obtain_unsequenced_lock_no_wait**
- **lm_release_sequenced_lock**
- **lm_release_unsequenced_lock**
- **misc_obtain_spin_lock**
- **misc_release_spin_lock**

Routines beginning with **lm** and **misc** require the **i_lm.h** and **i_misc.h** include files, respectively.

Constants and Data Structures

The routines in this section use the following constants and data structures. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_lm.h** for structures beginning with the **lm** acronym). Chapter 1 lists the various include files.

lm_sequenced_lock_type

```
typedef struct
    {
    lm_resource_counter_type    rc;

    }  lm_sequenced_lock_type   ;
```

Description

This type is a sequenced lock. A sequenced lock may be created by simply declaring an instance of this type. The user of the lock is responsible for allocating the space occupied by the lock instance and reclaiming that space when the lock is destroyed.

A sequenced lock is simply a resource counter that has an initial value of one.

lm_unsequenced_lock_type

```
typedef struct
    {
    vp_unsequenced_lock_type     lock;
```

```
        }  lm_unsequenced_lock_type  ;
```

Description

This type is an unsequenced lock. An unsequenced lock may be created by simply declaring an instance of this type. The user of the lock is responsible for allocating the space occupied by the lock instance and reclaiming that space when the lock is destroyed.

misc_spin_lock_type

```
typedef bit32e_type      misc_spin_lock_type ;
```

Description

This type defines a spin lock. The spin lock actually uses only the low bit of the 32. The lock is considered held when the low order bit is 1, and is considered not held otherwise.

lm_initialize_sequenced_lock

Syntax

```
void lm_initialize_sequenced_lock (lock_ptr)

lm_sequenced_lock_ptr_type  lock_ptr;       /*WRITE ONLY*/
```

Summary

This routine initializes a sequenced lock.

Parameters

lock_ptr — A pointer to the lock to be initialized.

Description

This routine initializes a sequenced lock. None of the obtain or release operations should be performed on a lock until it has been initialized by this routine.

Return Values

None.

Exceptions

None.

lm_initialize_unsequenced_lock

Syntax

```
void lm_initialize_unsequenced_lock (lock_ptr)

lm_unsequenced_lock_ptr_type  lock_ptr;    /*WRITE ONLY*/
```

Summary

This routine initializes an unsequenced lock.

Parameters

lock_ptr — A pointer to the lock to be initialized.

Description

This routine initializes an unsequenced lock. None of the obtain or release operations should be performed on a lock until it has been initialized by this routine.

Return Values

None.

Exceptions

None.

lm_obtain_sequenced_lock

Syntax

```
void  lm_obtain_sequenced_lock(lock_ptr)

lm_sequenced_lock_ptr_type  lock_ptr;        /*WRITE ONLY*/
```

Summary

This routine obtains the specified lock.

Parameters

lock_ptr — A pointer to the lock to be obtained.

Return Values

None.

Exceptions

None.

lm_obtain_sequenced_lock_no_wait

Syntax

```
boolean_type  lm_obtain_sequenced_lock_no_wait (lock_ptr)

lm_sequenced_lock_ptr_type  lock_ptr; /*READ/WRITE*/
```

Summary

This routine obtains the specified lock. The calling process is not pended if the lock is not immediately available. A boolean is returned, which indicates whether the lock was obtained.

Parameters

lock_ptr — A pointer to the lock to be obtained.

Description

See Summary.

Return Values

TRUE — The lock was obtained.

FALSE — The lock was not obtained.

Exceptions

None.

lm_obtain_unsequenced_lock

Syntax

```
void lm_obtain_unsequenced_lock(lock_ptr)

lm_unsequenced_lock_ptr_type  lock_ptr;     /*WRITE ONLY*/
```

Summary

This routine obtains the specified lock.

NOTE: The calling process will be pended if the lock is not immediately available.

Parameters

lock_ptr — A pointer to the lock to be obtained.

Return Values

None.

Exceptions

None.

lm_obtain_unsequenced_lock_no_wait

Syntax

```
boolean_type  lm_obtain_unsequenced_lock_no_wait (lock_ptr)

lm_unsequenced_lock_ptr_type  lock_ptr; /*READ/WRITE*/
```

Summary

This routine obtains the specified lock if it is not already held. The calling process will NOT be pended if the lock is not immediately available.

Parameters

lock_ptr — A pointer to the lock to be obtained.

Description

See Summary.

Return Values

TRUE — The lock was obtained.

FALSE — The lock was not obtained.

Exceptions

None.

lm_release_sequenced_lock

Syntax

```
void  lm_release_sequenced_lock(lock_ptr)

lm_sequenced_lock_ptr_type  lock_ptr;        /*WRITE ONLY*/
```

Summary

This routine releases the specified lock. If other processes are waiting for the lock to become available, the next one in sequence will be awakened.

Parameters

lock_ptr — A pointer to the lock that is to be released.

Return Values

None.

Exceptions

None.

lm_release_unsequenced_lock

Syntax

```
void  lm_release_unsequenced_lock(lock_ptr)

lm_unsequenced_lock_ptr_type  lock_ptr;       /*WRITE ONLY*/
```

Summary

This routine releases the specified lock. If other processes are waiting for the lock to become available, all waiting processes will be awakened and one will be given the lock.

Parameters

lock_ptr — A pointer to the lock that is to be released.

Return Values

None.

Exceptions

None.

misc_obtain_spin_lock

Syntax

```
void    misc_obtain_spin_lock     (lock_ptr)

misc_spin_lock_ptr_type lock_ptr; /*READ/WRITE*/
```

Summary

This routine obtains a spin lock. If the lock is not immediately available, the process will loop until it becomes available.

Parameters

lock_ptr — A pointer to the spin lock that is to be obtained.

Description

An attempt is made to obtain the lock. If the lock is already held, the code loops until the lock is obtained. Spin locks are the only locks that can be obtained at interrupt level.

Return Values

None.

Exceptions

None.

misc_release_spin_lock

Syntax

```
void    misc_release_spin_lock    (lock_ptr)

misc_spin_lock_ptr_type lock_ptr; /*READ/WRITE*/
```

Summary

This routine releases a spin lock.

Parameters

lock_ptr — A pointer to the spin lock that is to be released.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

Chapter 5 Eventcounter Routines

This chapter describes all DG/UX kernel routines used in handling eventcounters. We start with a brief introduction to eventcounters and eventcounter routines. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_vp.h** for structures beginning with the **vp** acronym) for a complete and current list of all constants and structures.

Overview to Using Eventcounters

Eventcounters are the primary synchronization mechanism used in the DG/UX kernel. The DG/UX system's treatment of eventcounters and the related concept of a sequencer comes from work by Reed and Kanodia. See the *Communications of the ACM* papers listed in the "Other Documents" section of the Preface for more technical background on eventcounters and sequencers.

The eventcounter synchronization mechanism uses two basic elements: eventcounters and events. An eventcounter is simply a count of the number of times some condition of interest has happened. You create an eventcounter by declaring a variable of eventcounter type and then initializing its count value to zero by calling **vp_initialize_ec**.

Events are separate from eventcounters. Eventcounters allow you to define "events" of interest by connecting an eventcounter with a critical value. When the eventcount is equal to the critical value, the event is said to be "satisfied" and the kernel automatically awakens all processes waiting on the event. You create an event by declaring a variable of event type (**vp_event_type**) and filling in the name of the eventcounter (pointer to its count address) and the critical value. Typically, you will define an eventcounter globally and create successive events from it.

Typically, you will also want to wait for the event to occur once you have created it. You do this by calling **vp_await_ec**. The **vp_await_ec** routine actually allows you to suspend waiting for any of a number of events supplied in an event list. If one or more of the specified events is already satisfied when the await call is made, await returns immediately and the process continues execution. If none of the specified events is satisfied, the process enters the awaiting state where it does not compete for CPU resources. Because a process doing a **vp_await_ec** may suspend indefinitely, it should only hold locks while awaiting an event that can be counted on occurring in a reasonable time (perhaps a second or less).

When one of the events is satisfied, the kernel will awaken the waiting process and pass it the index of the event that has occurred. The index identifies the event in the list that caused the await to be satisfied. However, the event specified by the index is not necessarily the only event that has occurred in the list. You may determine if other events in the list have occurred by calling the routine **vp_has_event_occurred** for each entry in the event list. Note that if you want to wait until ALL of the desired events have occurred, you may need to do several calls to **vp_await_ec.**

Frequently, the event you will want to create is the next occurrence of the condition, that is, the next increment of the eventcounter. You can create such an event by: 1) calling **vp_read_ec** which reads the current count into an event value variable; and 2) calling **vp_increment_ec** which adds one to that count value — making a critical value equal to "the next occurrence." Alternatively, you can call **vp_get_next_ec_value** to perform these two steps in one indivisible step; it reads and returns an incremented count into an event value variable.

Some part of your code will also have to increment the eventcounter each time the condition of interest occurs. You increment (advance) the eventcounter by calling **vp_advance_ec.** After incrementing the specified eventcounter, the advance operation checks to see whether the new value of the incremented eventcounter causes any events to be satisfied. If the process associated with a satisfied event is still in the awaiting state, it is scheduled to run. Because interrupts are one common condition of interest, interrupt service routines are frequently the ones calling **vp_advance_ec.**

Because eventcounters are monotonically increasing values, they map very well into the normal concepts of clocks and times. This allows clocks routines and timer routines to be based on the same eventcounter mechanism. A clock can be considered an eventcounter, and when the clock reaches a certain value an event is triggered. The two routines **vp_convert_clock_value_to_ec_value** and **vp_convert_ec_value_to_clock_value** allow you to convert between clock and eventcounter values.

Because eventcounters are monotonically increasing values, they also provide a natural ordering of events. This allows the eventcounter mechanism to be extended to support sequencing using the concept of "sequencers". Often simply waiting on an event is not enough; what is wanted is a way of ordering, or sequencing, the waiters on an event. This is often the case when the event being awaited is access to a resource of some sort, such as a critical section of code or shared data. Sequencers support such ordering.

Sequencers, like eventcounters, are simply counters with values that increase in a monotonic fashion. Like eventcounters, sequencers are declared and initialized (in sequencer's case, by calling **vp_initialize_sequencer**). Sequencers order events by issuing sequential "tickets." You get a ticket by calling **vp_ticket_sequencer** which atomically increments the current value of the sequencer and returns the new value. Thus, each caller of the ticket operation gets a unique value and the values are ordered by the order in which the calls to ticket were made: the first caller will get 1, the second 2, and so on. You create events using these ticket values and await them using **vp_await_ec.** Each process will see its event in turn — in the same order as the sequencer values. This is exactly the same as in any store where you "take a number for service."

Eventcounters offer several advantages over the more simplistic synchronizations techniques used in most standard UNIX implementations. First, because eventcounters actually count the number of occurrences of an event, you can tell if an event has already happened. Thus, in a sense eventcounters remember previous events. If code tries to wait on an event that has already happened (the event's critical value is less than the current count), the wait returns immediately because the event has been satisfied. There is no danger of the waiting process pending forever as with the standard UNIX sleep and wakeup primitives.

The **vp_are_ec_values_equal** routine for comparing eventcounter values is provided for convenience.

You must be careful of the order in which you perform the tasks involved in creating and awaiting an event lest you accidently create an endless wait situation. Specifically, if you start the I/O operation to be awaited before you create the event, the I/O may be logged before you get the eventcounter and the condition you create will be one count past the operation you started. The best sequence for creating and awaiting events is: 1) create the event to be awaited; 2) start the I/O operation; 3) check the event; 4) if it is not satisfied (and you do want to suspend until it is), start the await process. The typical code sequence is as follows:

```
dev_cird_build_scatter_gather_arrays(request_block_ptr);
vp_get_next_ec_value(&request_block_ptr->sync_io_ec,
                             &request_completion_event.value);
status = dev_cird_start_command_list_request(request_block_ptr);
if (status == OK)
    {
    vp_await_ec(&request_completion_event, (int32_type)1, &result_index);
    }
```

If you use routines from this section, you must allocate the space used by the event and eventcounter instances (see the "Constants and Data Structures" section below). Eventcounters are normally allocated from global memory. Event types are allocated dynamically, as needed.

The following routines are described in this section:

- **vp_add_to_ec_value**
- **vp_advance_ec**
- **vp_await_ec**
- **vp_convert_clock_value_to_ec_value**
- **vp_convert_ec_value_to_clock_value**
- **vp_get_next_ec_value**
- **vp_has_event_occurred**
- **vp_increment_ec_value**

- **vp_initialize_ec**
- **vp_initialize_sequencer**
- **vp_read_ec**
- **vp_ticket_sequencer**
- **vp_are_ec_values_equal**

Routines beginning with **vp** require the **i_vp.h** include file.

Constants and Data Structures

This section discusses some of the data structures used by synchronization routines. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_vp.h** for structures beginning with the **vp** acronym). Chapter 1 lists the various include files.

vp_event_type

```
typedef struct
   {
   vp_ec_ptr_type      name;
   vp_ec_value_type    value;
   }
   vp_event_type ;
```

Description

This structure defines an event, which is an eventcounter name and an eventcounter value. The event is said to occur or to be satisfied when the value of the eventcounter pointed to by the `name` field is greater than or equal to the `value` field.

vp_add_to_ec_value

Syntax

```
void   vp_add_to_ec_value  (ec_value_ptr, addend)

vp_ec_value_ptr_type  ec_value_ptr;   /*READ/WRITE*/
uint32_type           addend;         /*READ ONLY*/
```

Summary

This routine adds the given value to the specified eventcounter value.

Parameters

ec_value_ptr — A pointer to the eventcounter value to be added to.

addend — The value to be added to the eventcounter value.

Description

The specified 32-bit integer is added to the specified eventcounter value.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

vp_advance_ec

Syntax

```
void    vp_advance_ec  (ec_name)

vp_ec_ptr_type     ec_name;              /*READ ONLY*/
```

Summary

This routine performs an advance (by one) on the specified eventcounter. Any processes awaiting on the new value of the eventcounter will be notified.

Parameters

ec_name — A pointer to the eventcounter to be advanced.

Description

The eventcounter is indivisibly incremented, and any processes awaiting on the new value are notified. If a higher priority process becomes eligible to run as a result of the notification, it may be rescheduled. Thus, your process may be pre-empted if you call this routine.

Return Values

None.

Exceptions

None.

vp_awa: _ec

Syntax

```
void :p_await_ec (event_list, list_size, list_index_ptr)

vp_ev :nt_type    event_list[];      /*READ ONLY*/
int3: _type       list_size;         /*READ ONLY*/
int3: ptr_type    list_index_ptr;    /*WRITE ONLY*/
```

Summary

This : outine performs the await operation on one or more events. The calling proce s will be suspended until at least one of the specified events is satisfied.

Paramete s

even: ist — An array of events for which the process wishes to await.

list_s :e — The number of elements in **event_list**.

list_i: dex_ptr — A pointer to the array index (zero based) of an event that is satisf. :d when the call returns.

Descripti n

This outine causes the calling process to be suspended until any one of the supp. :d events has been satisfied. If any of the events is satisfied at the time the call i made, the process is not suspended. When the call returns, the **list_i dex_ptr** is set to the index of an event that is satisfied, but if more than one even .s satisfied, no statement is made about which event will be indicated by **list_i . dex_ptr**.

Return V lues

Non .

Exceptio :s

Nor .

vp_convert_clock_value_to_ec_value

Syntax

```
void vp_convert_clock_value_to_ec_value (clock_value_ptr,
                                         ec_value_ptr)

misc_clock_value_ptr_type  clock_value_ptr; /*READ ONLY*/
vp_ec_value_ptr_type       ec_value_ptr;    /*WRITE ONLY*/
```

Summary

This routine converts a clock value into an eventcounter value.

Parameters

clock_value_ptr — A pointer to a clock value.

ec_value_ptr — A pointer to the location where the corresponding eventcounter value is to be written.

Description

This routine converts a clock value into an eventcounter value. Converting from clock value to eventcounter value requires converting the 64-bit clock value to a 32-bit eventcounter value.

The number of bits to take from the high and low word of the clock value are defined in **i_vp.h** as **VP_CLOCK_TO_EC_HIGH_BITS** and **VP_CLOCK_TO_EC_LOW_BITS**.

Return Values

None.

Exceptions

None.

vp_convert_ec_value_to_clock_value

Syntax

```
void vp_convert_ec_value_to_clock_value (ec_value_ptr,
                                         clock_value_ptr)

vp_ec_value_ptr_type       ec_value_ptr;    /*READ ONLY*/
misc_clock_value_ptr_type  clock_value_ptr; /*WRITE ONLY*/
```

Summary

This routine converts an eventcounter value into a clock value.

Parameters

ec_value_ptr — A pointer to an eventcounter value.

clock_value_ptr — A pointer to the location where the corresponding clock value is to be written.

Description

This routine converts an eventcounter value into a clock value. Conversion from eventcounter value to clock value requires converting a 32-bit eventcounter value to a 64-bit clock value.

The number of bits to assign to the high and low word of the clock value are defined in **i_vp.h** as VP_CLOCK_TO_EC_HIGH_BITS and VP_CLOCK_TO_EC_LOW_BITS.

Return Values

None.

Exceptions

None.

vp_get_next_ec_value

Syntax

```
void vp_get_next_ec_value (ec_name, ec_value_ptr)

vp_ec_ptr_type  ec_name;                /*READ ONLY*/
vp_ec_value_ptr_type  ec_value_ptr;     /*WRITE ONLY*/
```

Summary

This routine indivisibly reads the specified eventcounter and returns its value plus one.

Parameters

ec_name — A pointer to the eventcounter to be read.

ec_value_ptr — A pointer to the location where the eventcounter value (plus one) is to be written.

Description

The eventcounter is read indivisibly with respect to other processors and with respect to the executing processor's interrupt level. The value is then incremented by one, which is equal to the value that will be reached the next time the eventcounter is advanced.

Return Values

None.

Exceptions

None.

vp_has_event_occurred

Syntax

```
boolean_type  vp_has_event_occurred (event_ptr)

vp_event_ptr_type   event_ptr;          /*READ ONLY*/
```

Summary

This routine determines whether the given event has occurred.

Parameters

event_ptr — A pointer to the subject event.

Return Values

TRUE — The event has been satisfied.

FALSE — The event has not yet occurred.

Exceptions

None.

vp_increment_ec_value

Syntax

```
void     vp_increment_ec_value  (ec_value_ptr)

vp_ec_value_ptr_type ec_value_ptr;    /*READ WRITE*/
```

Summary

This routine increments the specified eventcounter value.

Parameters

ec_value_ptr — A pointer to the eventcounter value to be incremented.

Description

This routine simply takes the eventcounter value passed in and increments it.

Return Values

None.

Exceptions

None.

vp_initialize_ec

Syntax

```
void      vp_initialize_ec  (ec_name)

vp_ec_ptr_type      ec_name;              /*READ ONLY*/
```

Summary

This routine initializes an eventcounter.

Parameters

ec_name — A pointer to the eventcounter to be initialized.

Description

The eventcounter value is set to zero.

Return Values

None.

Exceptions

None.

vp_initialize_sequencer

Syntax

```
void            vp_initialize_sequencer  (seq_name)

vp_ec_ptr_type     seq_name;     /*READ ONLY*/
```

Summary

This routine initializes a sequencer.

Parameters

seq_name — A pointer to the sequencer to be initialized.

Description

The sequencer value is set to zero.

Return Values

None.

Exceptions

None.

vp_read_ec

Syntax

```
void             vp_read_ec  (ec_name, ec_value_ptr)

vp_ec_ptr_type         ec_name;          /*READ ONLY*/
vp_ec_value_ptr_type   ec_value_ptr;     /*WRITE ONLY*/
```

Summary

This routine indivisibly reads the specified eventcounter and returns the value in the variable pointed to by **ec_value_ptr**.

Parameters

ec_name — A pointer to the eventcounter to be read.

ec_value_ptr — A pointer to the location in which the eventcounter value is to be written.

Description

The eventcounter is read indivisibly with respect to other processors and with respect to the executing processor's interrupt level.

Return Values

None.

Exceptions

None.

vp_ticket_sequencer

Syntax

```
void     vp_ticket_sequencer  (seq_name, seq_value_ptr)

vp_ec_ptr_type        seq_name;        /*READ ONLY*/
vp_ec_value_ptr_type  seq_value_ptr;   /*WRITE ONLY*/
```

Summary

This routine indivisibly increments the value of the specified sequencer and returns the new value (that is, the value after the increment).

Parameters

seq_name — A pointer to the sequencer to be ticketed.

seq_value_ptr — A pointer to the location in which the new value of the sequencer is to be written.

Description

The sequencer value is incremented and then read as an indivisible operation.

Return Values

None.

Exceptions

None.

vp_are_ec_values_equal

Syntax

```
boolean_type  vp_are_ec_values_equal (value1_ptr, value2_ptr)

vp_ec_value_ptr_type  value1_ptr;    /*READ ONLY*/
vp_ec_value_ptr_type  value2_ptr;    /*READ ONLY*/
```

Summary

This routine compares two eventcounter values for equality.

Parameters

value1_ptr — A pointer to an eventcounter value.

value2_ptr — A pointer to an eventcounter value.

Description

This routine compares two eventcounter values and returns TRUE if they are equal.

Return Values

TRUE — The eventcounter values are equal.

FALSE — The eventcounter values are not equal.

Exceptions

None.

End of Chapter

Chapter 6
Clock Routines

This chapter describes the DG/UX kernel routines used in clock operations. We start with a brief introduction to the clock and clock routines. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_vp.h** for structures beginning with the **vp** acronym) for a complete and current list of all constants and structures.

Overview to Using Clock Routines

The kernel provides three sets of clock routines: 1) routines to create an event that will occur at a specified time; 2) routines to establish and cancel timeouts; and 3) a routine to read the system clock.

The kernel maintains time via the system clock. The system clock is a 64-bit logical counter that increments at a fixed rate in real time. The counter is given value zero at system boot time. System clock values are continuous and monotonically increasing. Continuous means that the value of the system clock is not changed even if the external time-of-day is changed. Therefore, you can use the system clock to time intervals knowing its value will not be reset during the interval.

The system clock maintains the time since the system was booted in **misc_clock_value_type** units. A **misc_clock_value_type** unit is a 64-bit value where the high order 32 bits represent seconds and the low order 32 bits represent a fraction of a second. The number of significant bits in the fractional part of a second is determined by the accuracy of the architecture-dependent hardware clock used to implement the system time. **misc_clock_value_type** and pre-defined values for it are shown in the "Constants and Data Structures" section.

You can read the system clock using the **vp_read_system_clock** routine. This may be useful for applications that are doing timing intervals.

The clock routines also let you schedule events based on system time. You can use these clock events either asynchronously (time-outs) or synchronously (clock events). You use clock eventcounters to await for a time interval synchronously (suspended and thus without continuing processing). You use the routine **vp_create_clock_event** to create a clock event that will occur after some specified system time interval. After you create the event, you await it using the **vp_await_ec** routine described in Chapter 5. Chapter 5 also describes other routines you can use in manipulating eventcounters. The following sample shows how to create a clock event that will occur in 5 seconds:

```
vp_create_clock_event(&delay_event, &misc_five_seconds);
vp_await_ec(&delay_event, (int32_type)1, &result_index);
```

Clock events are typically used in synchronous I/O requests. In this application the driver will issue an I/O request and then suspend waiting for one of three events to occur: the completion of the I/O request; a time-out of the I/O request; or termination of the I/O process. Upon awakening, the driver determines which event occurred and performs the appropriate operations.

The DG/UX system provides time-out services for doing asynchronous processing. With a time-out, the kernel is directed to call a specified routine after a specified interval expires.

The kernel needs a certain amount of data space to handle a time-out. Since it allocates this space dynamically at run-time, you must declare the amount of space you will need. You do this by calling **vp_specify_max_timeouts**.

Once this space has been allocated, it cannot be released and more cannot be allocated. This means several things. First, be rational when setting this value; try not to allocate too many or too few time-outs. Allocating too few time-outs is particularly dangerous. If you ask for more than the specified **max_timeouts**, the system will panic because of insufficient resources. You should make sure that you never have more concurrent time-outs than you specified. The concurrency of time-outs, then, is also critical. The time-out routines **vp_establish_timeout** and **vp_cancel_timeout** are a matched set. One establishes the time-out and the other cancels it; the time-out is still current until you call **vp_cancel_timeout**. You must call **vp_cancel_timeout** once for each call to **vp_establish_timeout**, regardless of whether or not the time-out event has occurred. You can cancel the time-out before it expires, but you must cancel it after it expires.

When you call **vp_establish_timeout**, it returns a time-out ID. You need this ID to cancel the time-out. Cancelling the time-out will prevent your time-out routine from being called if the interval expires, and will free resources for another time-out to run.

Your time-out routine will run at interrupt level with event resources locked. Thus, it should not invoke event routines that might lock event resources (and thus deadlock the system). This includes advancing events and awaiting events. As an alternative, you can have the routine return an eventcounter name to be advanced by the higher process on behalf of the time-out routine when it is safe to do so.

The routines described in this section are as follows:

- **vp_establish_timeout**
- **vp_cancel_timeout**
- **vp_specify_max_timeouts**
- **vp_create_clock_event**
- **vp_read_system_clock**

Routines beginning with **vp** require the **i_vp.h** include file.

Constants and Data Structures

This section describes the format of system clock values and the general clock value constants that may be needed by other subsystems. These constants are allocated in global memory, and the data types are defined in **i_misc.h**. Pointers to the constants are passed to the clock management routines to specify time values. Generally useful values are defined in this section; if a subsystem has a need for a special clock value, it may define the value itself.

Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_misc.h** for structures beginning with the **misc** acronym). Chapter 1 lists the various include files.

misc_clock_value_type

```
typedef struct
    {
    uint32e_type            high;
    uint32e_type            low;
    }
    misc_clock_value_type
```

Description

This type describes a value that the system clock can have. The clock value is treated as a 64-bit signed integer with time values contained in the bottom 63 bits. (The bits are numbered such that bit 63 is the most-significant bit, and bit 0 is the least significant bit with bit 32 representing one second.) Actual resolution of timing may vary but will be accurate to at least 10 milliseconds.

You may use the following defined constants in your driver. They are defined in **i_misc.h**.

misc_five_minutes
misc_one_hundred_seconds
misc_one_minute
misc_ten_seconds
misc_five_seconds
misc_three_seconds
misc_two_seconds
misc_one_second
misc_one_half_second
misc_two_hundred_fifty_milliseconds
misc_two_hundred_milliseconds

misc_ten_milliseconds

vp_establish_timeout

Syntax

```
opaque32_type vp_establish_timeout (time_ptr, routine_ptr,
                                    argument)
misc_clock_value_ptr_type    time_ptr;       /*READ ONLY*/
vp_timeout_routine_ptr_type  routine_ptr;    /*READ ONLY*/
bit32e_type                  argument;       /*READ ONLY*/
```

Summary

This routine establishes a timeout. The timeout will occur **time_ptr** time from the current time, and then the specified routine will be called with the specified argument.

Parameters

time_ptr — A pointer to a clock value indicating the amount of real time that is to elapse before the timeout occurs. Use the clock constants in the "Constants and Data Structures" section for increment values.

routine_ptr — A pointer to a routine that is to be called by the I/O daemon when the timeout occurs.

argument — A 32-bit value that is to be passed to the timeout routine as an argument.

Return Values

timeout_id — The return value is an opaque 32-bit identifier for the timeout. This value may be used only as an argument to **vp_cancel_timeout**.

Exceptions

None.

vp_cancel_timeout

Syntax

```
void    vp_cancel_timeout    (timeout_id)

opaque32_type  timeout_id;  /*READ ONLY*/
```

Summary

This routine cancels a previously established timeout.

Parameters

timeout_id — The **timeout_id** of the timeout to be cancelled. This value was returned by the **vp_establish_timeout** routine.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

vp_specify_max_timeouts

Syntax

```
void         vp_specify_max_timeouts  (count)

uint32_type  count;      /*READ ONLY*/
```

Summary

This routine reserves space for the specified number of timeouts. A device driver should call it to reserve space for the maximum number of timeouts it will ever have in effect simultaneously.

Parameters

count — The number of timeouts for which to reserve space.

Description

Space is reserved for the specified number of timeouts. The space must be reserved before any timeouts are established. This routine will presumably be called several times, once by each driver in the system, as part of its initialization.

The amount of space reserved for timeouts cannot be reduced. Therefore you should try not to ask for more space than you will need during the life of the system.

Return Values

None.

Exceptions

None.

vp_create_clock_event

Syntax

```
void    vp_create_clock_event  (event_ptr, increment_ptr)

vp_event_ptr_type           event_ptr;      /*WRITE ONLY*/
misc_clock_value_ptr_type   increment_ptr;  /*READ ONLY*/
```

Summary

This routine sets up a clock event for a specified (**increment_ptr**) time in the future.

Parameters

event_ptr — A pointer to the event that is to be set up.

increment_ptr — A pointer to a clock value that is to be added to the current system time. Use the clock constants in the "Constants and Data Structures" section for increment values.

Description

event_ptr is set to an event. The value of the eventcounter is set to make the event occur at current time plus the increment. See the "Synchronization Routines" section of this chapter for other routines used in servicing the event. For example, you may want to use **vp_await_ec** to await the occurrence of this event. You do this by specifying the event in **vp_await_ec**'s event list.

Return Values

None.

Exceptions

None.

vp_read_system_clock

Syntax

```
void    vp_read_system_clock  (current_time_ptr)

misc_clock_value_ptr_type  current_time_ptr;   /*READ ONLY*/
```

Summary

The current value of the system clock is returned.

Parameters

current_time_ptr — A pointer to where the current value of the system clock is to be written.

Return Values

None.

Exceptions

None.

End of Chapter

Chapter 7
Process and Signal Management Routines

This chapter describes the DG/UX kernel routines used in sending and receiving signals. We start with a brief introduction to signals and the signal handling routines. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_pm.h** for structures beginning with the **pm** acronym) for a complete and current list of all constants and structures.

Overview to Using Process Signal Management Routines

The DG/UX kernel allows you to send a signal to either a particular process or to a group of processes. You can send signals selectively based on the target process': 1) process index; 2) process ID (PID); or 3) process group (PGRP). The routines you use are: **pm_send_signal_by_process_id**, **pm_send_signal_by_process_index**, and **pm_send_signal_by_process_group**. In the DG/UX kernel, it is more efficient to refer to processes by their process index than by their PID. Hence, signalling by **pm_send_signal_by_process_index** is the preferred method.

Typically you will be sending signals to your own process or other processes in you process group, and you will need the appropriate process identifier. You can determine you own PID and PGRP by calling **pm_get_my_pid** or **pm_get_my_pgrp**, respectively. You can read your process's process index from a per-process variable called **sc_my_process_index**.

You must query the kernel to find out if you have a signal pending. You query for any pending signals using the **pm_is_interrupted** and **pm_is_terminated** routines.

The kernel identifies two classes of signals: normal ones that the calling process can handle; and terminal ones that will cause the calling process to terminate (with or without a core dump). You call **pm_is_interrupted** to find out about both normal and terminate signals. You call **pm_is_terminated** to ask about terminate signals only.

Though you must query for the signal, signals may also be processed in conjunction with events. If a signal is present, the query routine (for example, **pm_is_interrupted**) returns a flag indicating the signal is present and you process the signal. If a signal is not present, the flag is FALSE and the query routine returns an event. You can use this event to suspend and wait for the signal. To do so you add the event to the list of events to be awaited and call **vp_await_ec**. Generally, the event will be satisfied when a signal occurs but this is not always the case. Therefore, when the event is satisfied, you should verify that a signal has occurred by calling the query routine again.

Note that **pm_is_interrupted** may suspend the calling process for an arbitrary amount of time and should only be used when an indefinite wait is possible. For example, if a process that is being debugged receives a signal, it will communicate with its debugger during **pm_is_interrupted** (see **ptrace**(2)). Because of this, **pm_is_interrupted** may pend indefinitely waiting for a debugger to continue the calling process. Because it may pend indefinitely, you should avoid holding locks when calling **pm_is_interrupted**.

pm_is_terminated is designed for use within the kernel when a potentially long, but nevertheless finite length operation is started. For example, a space operation on a tape drive or an I/O request to an NFS server are such potentially long operations. In either case the operation will eventually finish (possibly due to a time-out), but the user may like the option to terminate the operation mid-stream by sending the process a signal. **pm_is_terminated** does not communicate with any debugger process and does not flag non-terminal signals. Because it only informs the caller of terminal signals, you can safely call it while holding a lock.

The routines described in this section are as follows:

- **pm_get_my_pid**
- **pm_get_my_pgrp**
- **pm_is_interrupted**
- **pm_is_terminated**
- **pm_send_signal_by_index**
- **pm_send_signal_by_process_group**
- **pm_send_signal_by_process_id**

Routines beginning with **pm** require the **i_pm.h** include file.

Constants and Data Structures

No special constants or data structures are required by these routines.

pm_get_my_pid

Syntax

```
pm_process_id_type  pm_get_my_pid ()
```

Summary

Returns the process id of the "calling" process.

Parameters

None.

Description

See Summary.

Return Values

The current pid.

pm_get_my_pgrp

Syntax

```
pm_process_id_type  pm_get_my_pgrp ()
```

Summary

Returns the process group of the "calling" process.

Parameters

None.

Description

See Summary.

Return Values

The process group

pm_is_interrupted

Syntax

```
boolean_type pm_is_interrupted (event_ptr)

vp_event_ptr_type  event_ptr;          /*WRITE ONLY*/
```

Summary

This routine handles signals during a system call.

Parameters

event_ptr — The address of a process interrupt event.

Description

This routine handles signal processing. It should be used whenever a system call will pend the calling process until some external event occurs (that is, pend for an arbitrary amount of time). Processing includes the following:

- Interrupting the system call.
- Terminating the process (with or without a core dump).
- Stopping the process for an arbitrary amount of time.

Only the last of these actions is contained entirely within the **pm_is_interrupted** routine. The first two actions are performed in cooperation with the caller.

Typically, you will use the following code fragment:

```
if (pm_is_interrupted(&events[PROCESS_INTERRUPT]))
{
Arrange to return EINTR to the user.  Exit with error EINTR.
}
vp_await_ec(events, N, &index);
Act on the event that was satisfied.
If only the PROCESS_INTERRUPT was satisfied, loop back to
pm_is_interrupted()
```

In the code shown above, the relevant events are those in the `events[]` array in the first line. In addition, the event returned by **pm_is_interrupted** is also important. If the calling process is interrupted, the system call will return an error and will set errno to EINTR. Otherwise, the system call pends until the calling process is interrupted or one of the relevant events has happened.

Return Values

TRUE — A signal is presented to be handled.

FALSE — No signal is present.

[*event*] — **event_ptr** is set to an event that will occur when it is appropriate to check for signals again.

Exceptions

None.

pm_is_terminated

Syntax

```
boolean_type pm_is_terminated (event_ptr)

vp_event_ptr_type  event_ptr;          /*WRITE ONLY*/
```

Summary

This routine checks for termination signals during a system call.

Parameters

event_ptr — The address of a process interrupt event.

Description

This routine determines whether the calling process has any signals that will cause process termination.

Return Values

TRUE — A signal is presented to be handled.

FALSE — No signal is present. **event_ptr** is set to an event that will occur when it is appropriate to re-check for termination signals.

Exceptions

None.

Remarks

This call is designed for use within the kernel when a potentially long but nevertheless finite operation is started. For example, a spacing operation on a tape drive or an I/O request to an NFS server is essentially indefinite. In both of these cases, the operation is guaranteed to eventually finish, perhaps due to a timeout; but the end user may like the option of terminating the operation mid-stream by sending the process a signal.

pm_send_signal_by_index

Syntax

```
void pm_send_signal_by_index (index, signal, signal_source)

sc_process_index_type        index;          /*READ ONLY*/
pm_signal_type               signal;         /*READ ONLY*/
pm_signal_source_enum_type   signal_source;  /*READ ONLY*/
```

Summary

If the subject process exists, this routine sends the process a signal. If the subject process does not exist, this routine has no effect.

Parameters

index — The subject process' index. The index is a unique identifier assigned to each process. It is maintained in per process data and is contained in the variable **sc_my_process_index**.

signal — The signal to send.

signal_source — The reason the signal is being sent.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

pm_send_signal_by_process_group

Syntax

```
status_type pm_send_signal_by_process_group (process_group,
                                             signal_number,
                                             signal_source)

pm_process_id_type           process_group;  /*READ ONLY*/
pm_signal_type               signal_number;  /*READ ONLY*/
pm_signal_source_enum_type   signal_source;  /*READ ONLY*/
```

Summary

This routine sends a signal to a process group.

Parameters

process_group — The process group ID of the target process.

signal_number — The signal being sent.

signal_source — The reason the signal is being sent.

Description

Send the signal **signal_number** to the processes whose process group ID is **process_group**. The signal is sent only to processes that are not system processes and to which the calling process has permission to send a signal.

Return Values

The following values may be returned:

OK — The signal was sent successfully.

PM_ESRCH_NO_SUCH_PROCESS_GROUP — No process corresponding to those specified by **process_group** can be found.

PM_ESRCH_NO_PERMISSION — The calling process does not have permission to signal the processes identified by **process_group**.

pm_send_signal_by_process_id

Syntax

```
status_type pm_send_signal_by_process_id (process_id,
                                          signal_number,
                                          signal_source)

pm_process_id_type            process_id;     /*READ ONLY*/
pm_signal_type                signal_number;  /*READ ONLY*/
pm_signal_source_enum_type    signal_source;  /*READ ONLY*/
```

Summary

This routine sends a signal to a process identified by **process_id**.

Parameters

process_id — The process ID of the target.

signal_number — The signal being sent.

signal_source — The reason the signal is being sent.

Description

Send the signal **signal_number** to the process identified by **process_id**. If **signal_number** is **PM_SIGNAL_SIGKILL**, **pm_send_signal_by_process_id** assumes that **process_id** does not identify a system process.

Return Values

The following values may be returned:

OK — The signal was sent successfully.

PM_ESRCH_NO_SUCH_PROCESS_ID — No process corresponding to that specified by **process_id** can be found.

PM_EPERM_NO_KILL_ACCESS — The sending process does not have permission to signal the receiving process.

pm_send_signal_with_siginfo

Syntax

```
void  pm_send_signal_with_siginfo  (index, siginfor_ptr)

sc_process_index_type   index,            /*READ ONLY*/
pm_siginfo_ptr_type     siginfo_ptr       /*READ ONLY*/
```

Summary

Send a signal to the process identified by **index**.

Parameters

index — The index of the process to whom we are sending the signal.

siginfo_ptr — A pointer to a **pm_siginfo_type** structure that describes the signal.

Description

If the target process exists, this routine will send it the signal described in the structure pointed to by **siginfo_ptr**. If the target process is initializing, terminating or already terminated, this routine does nothing. If the target process is in the Free state, this routine issues a panic.

To call this routine, you must completely describe the signal by filling in all of the fields of the signal information structure.

Return Values

None.

Abort Conditions

Panic may be invoked with the following error codes:

PM_PANIC_SEND_TO_FREE_PROCESS — The target process is in the Free state.

End of Chapter

Chapter 8
Interrupt Management Routines

This chapter describes the DG/UX kernel routines used in handling interrupts. We start with a brief introduction to interrupt handling. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_vp.h** for structures beginning with the **vp** acronym) for a complete and current list of all constants and structures.

Overview to Using Interrupt Management Routines

The kernel provides routines to mask and unmask device interrupts and processor interrupts. The routines to mask and unmask interrupts are architecturally independent, allowing applications that use these routines to be unaware of the processor they are currently executing on.

Disabling Interrupts on a Single Processor

You use **vp_disable_interrupts** to disable all interrupts on the processor on which your process is executing. The call keeps that processor from receiving interrupts. When you disable interrupts on your home processor, your process cannot be preempted or interrupted; in a certain sense your process now "owns" that processor. For this reason, you should use great care when disabling processor interrupts.

In nonsymmetric systems, processes often disable processor interrupts for locking purposes — in order to have exclusive access to a resource, such as a data structure. This does not work under symmetric processing because disabling interrupts on one processor does not prevent another process executing on a different processor from accessing the resource.

Processor interrupts are enabled using the routine **vp_enable_interrupts**. Any interrupt that occurred while interrupts were disabled was saved and will occur immediately when interrupts are enabled.

The disabling of interrupts on a processor is nested, such that for every disable an enable is required before interrupts are actually enabled on the processor. This prevents premature enabling of interrupts. You can use **vp_are_interrupts_disabled** to determine if the processor's interrupts are already disabled.

Masking Interrupts for a Particular Device

In addition to enabling/disabling interrupts for the entire processor, you can use **io_mask_interrupt_variety** and **io_unmask_interrupt_variety** to mask and unmask interrupts for a particular device. Masking device interrupts stops the device from interrupting on all processors.

The device mask and unmask routines take a parameter called **interrupt_variety** of type **uc_interrupt_enum_type** that allows drivers to maintain architectural independence. There are **uc_interrupt_enum_type** constants defined for every interrupt type for all hardware architectures the DG/UX system supports, for example, **Uc_Keyboard_Interrupt, Uc_SCSI_Interrupt, Uc_Duart_Interrupt.** Using these pre-defined values means your code can be used by all Data General machines that use the DG/UX system. For example, code to mask the **duart** interrupt would operate unchanged on all Data General machines that have **duarts** and the DG/UX system.

The masking of a device interrupt is nested, such that for every mask an unmask is required to actually unmask the interrupt.

Device drivers must mask device interrupts as part of the procedure to gain exclusive access to device registers. Because of the DG/UX kernel's symmetric multiprocessing, it is possible for one processor to be issuing commands to a device while another processor is handling a completion interrupt for the device. In both cases, the code must access the device registers. The following standard locking procedure keeps two sets of code from attempting access simultaneously:

```
NOT_INTERRUPT_HANDLER:                | INTERRUPT_HANDLER:
                                      |
io_mask_interrupt_variety(            | if(!obtain_lock_no_wait(
           Uc_Some_Interrupt);        |  &controller_lock)
obtain_lock(&controller_lock);        | return;
                  .                   |   .
   (manipulate registers)             | (manipulate registers)
                  .                   |   .
release_lock(&controller_lock);       |   .
io_unmask_interrupt_variety(          | release_lock(
           Uc_Some_Interrupt);        | &controller_lock);
```

Non-interrupt handler code (let's call it completion code) must mask the device's interrupts before it obtains a lock on the device's registers. This is done to prevent the device from interrupting and invoking interrupt handler code on a processor other than the completion code's home processor. Such masking of the device interrupts prevents a potential deadlock situation that can occur if interrupt handler and completion code are executing on the same processor.

Consider the scenario of completion code executing on Processor A and accessing device X without masking device interrupts. First, the completion code obtains a lock on the device registers. If the device interrupts Processor A while the lock was held, we would enter interrupt handler code. The interrupt handler code would try to obtain the lock but would fail and return. However, the interrupt would still be high on the processor because the interrupt handler could not access the device registers to acknowledge the interrupt. The processor would therefore return immediately to the interrupt handler code to service the interrupt, which would return because it could not obtain the lock, etc. This cycle would continue forever, deadlocking the system. Now, consider the same scenario with the completion code masking the device interrupts before acquiring the device register lock. In this case the device cannot interrupt the processor while the controller lock is held, thus avoiding the deadlock scenario.

In the above scenario, the deadlock could also be avoided by having completion code disable all processor interrupts instead of just the device interrupts. However, under symmetric multiprocessing, disabling processor interrupts is not sufficient.

Consider the scenario of completion code executing on Processor A that simply disables all processor interrupts. This prevents the device from interrupting Processor A, thus avoiding the deadlock described earlier. However, under symmetric multiprocessing the device may still interrupt another processor, say Processor B. In this case, Processor B would execute interrupt handler code, try to obtain the lock held by Processor A, fail and return. As before, the interrupt would remain high, causing Processor B to repeat the procedure. This would continue until the completion code on Processor A released the lock.

Although this situation would not cause the system to deadlock, it is inefficient to have Processor B sit in a loop trying to service the interrupt until the lock is released by Processor A. Now consider the same scenario where completion code masks the device interrupt before obtaining the lock. The masking prevents the interrupt from being generated to ANY processor, thereby avoiding the situation just described.

The two scenarios described show the necessity for a driver to mask device interrupts. The routine **io_unmask_interrupt_variety** is used to unmask the interrupts. If a device generates an interrupt while it is masked by the processor, the interrupt is not lost. When the interrupt is unmasked, it will occur immediately.

The routines described in this section are as follows:

- **io_mask_interrupt_variety**
- **io_unmask_interrupt_variety**
- **vp_are_interrupts_disabled**
- **vp_disable_interrupts**
- **vp_enable_interrupts**

Routines beginning with **vp** and **io** require the **i_vp.h** and **i_io.h** include files, respectively.

Constants and Data Structures

The routines in this section use the following constants and data structures. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_uc.h** for structures beginning with the **uc** acronym). Chapter 1 lists the various include files.

uc_interrupt_enum_type

```
typedef enum
    {
    Uc_System_Alarm_Clock_Interrupt =    0,
    Uc_Keyboard_Interrupt =              1,
    Uc_Parallel_Port_Interrupt =         2,
    Uc_Ethernet_Interrupt =              3,
    Uc_SCSI_Interrupt =                  4,
    Uc_Duart_Interrupt=                  5,
    Uc_Graphics_Device_Interrupt =       6,
    Uc_Level_1_VME_Interrupt =           7,
    Uc_Level_2_VME_Interrupt =           8,
    Uc_Level_3_VME_Interrupt =           9,
    Uc_Level_4_VME_Interrupt =          10,
    Uc_Level_5_VME_Interrupt =          11,
    Uc_Level_6_VME_Interrupt =          12,
    Uc_Level_7_VME_Interrupt =          13,
    Uc_Dma_Terminal_Count_Interrupt =   14,
    Uc_System_Console_Interrupt =       15,
    Uc_Zbuffer_Interrupt =              16,
    Uc_Sync_Interrupt =                 17,
    Uc_Interrupt_Enum_Last =            18

    }    uc_interrupt_enum_type  ;
```

Description

This type is used to describe the type (or variety) of interrupt to be masked or unmasked.

io_mask_interrupt_variety

Syntax

```
void  io_mask_interrupt_variety (interrupt_variety)

uc_interrupt_enum_type interrupt_variety; /*READ ONLY*/
```

Summary

This routine masks a variety of interrupt specified in **interrupt_variety**.

Parameters

interrupt_variety — The type of interrupt to be masked. Any device that uses the interrupt variety to interrupt the system is effectively masked.

Description

This routine masks interrupts for a device with the interrupt type given in **interrupt_variety**. Any devices that use the interrupt variety to interrupt the system are effectively masked. If there are multiple processors, the interrupt is disabled for all processors. It also nests mask and unmask requests.

The routine uses a mask depth associated with the specified device to nest interrupts. This routine increments the mask depth, and if the new value is one, the hardware is updated to reflect a change in the mask.

You may call this routine from base level or from interrupt level. It remembers and correctly restores the state of the interrupt enable flag.

Return Values

None.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

IO_PANIC_ILLEGAL_MASK_INTERRUPT — The mask depth associated with the specified device has become larger than it should. This must be due to incorrect pairing of the mask and unmask functions by the caller.

io_unmask_interrupt_variety

Syntax

```
void io_unmask_interrupt_variety (interrupt_variety)

uc_interrupt_enum_type  interrupt_variety; /*READ ONLY*/
```

Summary

This routine unmasks a variety of interrupt specified in **interrupt_variety**.

Parameters

interrupt_variety — The type of interrupt to be unmasked. Any device that uses this interrupt variety is effectively unmasked.

Description

This routine unmasks interrupts for a device with the interrupt type given in **interrupt_variety**. Any devices that use the interrupt variety to interrupt the system are effectively unmasked. If there are multiple processors, the interrupt is enabled for all processors. The routine nests mask and unmask requests.

The routine uses a mask depth associated with the specified device to nest interrupts. This routine decrements the mask depth, and if the new value is 0, the hardware is updated to reflect a change in the mask.

You may call this routine from base level or from interrupt level. It remembers and correctly restores the state of the interrupt enable flag.

Return Values

None.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

IO_PANIC_ILLEGAL_UNMASK_INTERRUPT — The device's mask depth is equal to zero. This must be due to incorrect pairing of the mask and unmask function calls.

vp_are_interrupts_disabled

Syntax

```
boolean_type    vp_are_interrupts_disabled  ()
```

Summary

This routine returns TRUE if interrupts are disabled in the calling processor.

Parameters

None.

Return Values

TRUE — Interrupts are disabled in the calling processor.

FALSE — Interrupts are enabled in the calling processor.

Exceptions

None.

vp_disable_interrupts

Syntax

```
  void    vp_disable_interrupts  ()
```

Summary

This routine disables interrupts in the calling processor.

Parameters

None.

Description

Interrupts are disabled and the interrupt disable depth count is incremented. An interrupt disable depth count is maintained so that calls to this routine and **vp_enable_interrupts** will nest properly.

Return Values

None.

Exceptions

None.

vp_enable_interrupts

Syntax

```
void    vp_enable_interrupts  ()
```

Summary

This routine counters a previous call to disable interrupts in the calling processor. Interrupts are enabled if the disable depth is returned to zero.

Parameters

None.

Description

Multiple disable interrupt calls are tracked by the disable count depth. This routine counteracts one disable call by decrementing the disable depth count. If this decrement restores the count to its initial value, interrupts are enabled.

Return Values

None.

Exceptions

None.

End of Chapter

Chapter 9
Memory Allocation and Deallocation Routines

This chapter describes the kernel routines that your driver uses for allocating and releasing memory. We start with a brief introduction to allocating and releasing memory. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_vm.h** for structures beginning with the **vm** acronym) for a complete and current list of all constants and structures.

Overview to Using Memory Management Routines

We discuss the basics of kernel memory in Chapter 3. This section examines the particular routines you use for allocating and deallocating wired and unwired memory.

As described in Chapter 3, there are two basic types of kernel memory: wired (memory that will not be paged out) and unwired (memory that may be paged out). Hence, kernel memory management routines come in two corresponding versions, **_wired_** and **_unwired_** referred to in short hand as **_(un)wired_** (for example, **vm_get_(un)wired_memory** refers to both **vm_get_wired_memory** and **vm_get_unwired_memory**). In general, you should allocate unwired memory because wired memory is a limited resource. However, you must allocate wired memory for data structures that are referenced at interrupt level or for code that may be referenced during a page fault (for example, disk driver code).

The kernel provides two different ways to allocate memory: a demand version and a perhaps version. The perhaps version (**vm_perhaps_get_(un)wired_memory**) returns a pointer to a space of the required number of bytes or a VM_INVALID_MEMORY_PTR pointer if no more (un)wired memory is available. The demand version (**vm_get_(un)wired_memory**) should be used only for critical data segments. Like the perhaps version, the demand version returns a pointer to the allocated space if its successful. However, if the requested number of bytes are not available, the demand version causes a system panic.

Memory is deallocated using **vm_release_(un)wired_memory**. Note that you must match the get and release calls. For example, on release, you must supply the same number of bytes to the release call as specified in the get call. You must also release the memory in the same type as which it was originally allocated. Thus, you cannot allocate wired memory, unwire it, then release it as unwired memory — or allocate as unwired memory, wire it, then release it as wired memory.

Once you have allocated memory, you may switch its type by calling **vm_wire_memory** or **vm_unwire_memory**. However, there are certain restrictions on this conversion. The system keeps a "wire-count" which it increments for each wire and decrements for each unwire. You may wire (increment) and unwire (decrement) a memory area as much as you want as long as you do not take its wire count below its initial value: 0 for wired; 1 for unwired.

Memory-mapped I/O requires special allocation routines. Normally, a driver works within a logical address space and does not know or care what physical addresses are being used. However, in memory-mapped I/O, the device's registers are fixed to physical memory addresses. To make sure that the logical addresses the driver uses map to the correct physical addresses, you must direct a particular logical-to-physical mapping for the device's control registers. To do this you first reserve logical address space by allocating memory (that is calling one of the "get" routines). Then you call **vm_map_physical_memory** to map the logical address space to particular physical memory addresses. The device driver can now access the device registers through its logical address space.

The kernel also provides a number of routines to help drivers transfer data between the host and a device. To start a data transfer, the driver must have a buffer set up in wired memory. If the buffer was allocated out of wired kernel memory, then there is no need to wire the memory. Otherwise, if the buffer is in user space or was allocated out of unwired kernel memory, then the driver must wire the buffer memory using **vm_wire_memory**. To start the transfer (read or write), the driver must give the device the buffer's physical address. The driver gets the physical address of the buffer using **vm_get_physical_byte_address**. You cannot assume that the pages occupied by the buffer are physically contiguous. Thus, you must call **vm_get_physical_byte_address** for each page in the wired buffer.

After the I/O operation completes, you should unwire the previously wired memory. After completion the driver must also inform the kernel about the state of the memory. The kernel must know whether the memory was referenced and/or modified. DMA transfers do not set the referenced or modified bits in the page tables. If data was transferred from the buffer, then the device driver should call **vm_mark_ref_and_unwire_memory** to flag the memory as referenced. If data was transferred to the buffer, then the device driver should call **vm_mark_mod_and_ref_and_unwire_memory** to flag the memory as referenced and modified. Failure to appropriately set the reference and/or modified bits can cause new buffer data to be lost.

The routines described in this section are as follows:

- **vm_get_physical_byte_address**
- **vm_get_unwired_memory**
- **vm_get_wired_memory**
- **vm_map_physical_memory**
- **vm_unmap_physical_memory**

- **vm_mark_mod_and_ref_and_unwire_memory**
- **vm_mark_ref_and_unwire_memory**
- **vm_perhaps_get_unwired_memory**
- **vm_perhaps_get_wired_memory**
- **vm_release_unwired_memory**
- **vm_release_wired_memory**
- **vm_unwire_memory**
- **vm_wire_memory**

Routines beginning with **vm** require the **i_vm.h** include file.

Constants and Data Structures

This section discusses the literals used to specify data alignment in calls to **vm_get_wired_memory** and **vm_get_unwired_memory**.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_vm.h** for structures beginning with the **vm** acronym). Chapter 1 lists the various include files.

Page Alignment Literals

The following literals are used to specify alignment when "getting" memory:

```
VM_DOUBLE_WORD_ALIGNED
```

This constant will request double word alignment (64-bit).

```
VM_WORD_ALIGNED
```

This constant will request word alignment (32-bit).

```
VM_BYTE_ALIGNED
```

This constant will request byte alignment.

```
VM_DEFAULT_ALIGNMENT
```

This constant represents the most efficient alignment for use when allocating strings and structures. The default is double word alignment (64-bit) because, in most cases, the system deals with double word alignment most efficiently. Use this constant whenever possible.

After memory has been aligned, you cannot ask for a quantity smaller than the specified alignment. To ask for page alignment for more than one page, use VM_PAGE_ALIGNED shown below. To ask for page alignment for less than one page, use VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS (shown below).

```
VM_PAGE_ALIGNED
```

This constant will request page alignment. Don't use this alignment when VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS is sufficient, because it will result in wasted space.

```
VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS
```

This constant represents an alignment that is guaranteed not to cross a page boundary and will be default-aligned. Allocations that use this alignment are restricted to one page or less.

```
VM_INVALID_MEMORY_PTR
```

This constant will be returned by **vm_perhaps_get_wired_memory** and **vm_perhaps_get_unwired_memory** when the memory allocation fails.

vm_get_physical_byte_address

Syntax

```
void  vm_get_physical_byte_address   (logical_address,
                                      is_user_address,
                                      physical_address_ptr)

byte_address_type    logical_address;       /*READ ONLY*/
boolean_type         is_user_address;       /*READ ONLY*/
byte_address_type *  physical_address_ptr; /*WRITE ONLY*/
```

Summary

This function returns the physical address that corresponds to the given logical byte address.

Parameters

logical_address — The logical address for which a physical address is needed.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

physical_address_ptr — A pointer to the physical address corresponding to the given logical address, filled by this routine. If the logical address was invalid, this address will be **VM_INVALID_PHYSICAL_ADDRESS_PTR**.

Description

See Summary.

Return Values

None.

Exceptions

None.

vm_get_unwired_memory

Syntax

```
pointer_to_any_type vm_get_unwired_memory  (bytes, alignment)

uint32_type         bytes;             /*READ ONLY*/
uint32_type         alignment;         /*READ ONLY*/
```

Summary

This routine allocates unwired memory from available address space.

Parameters

bytes — The number of bytes to be allocated; **bytes** must be a positive value.

alignment — The byte alignment of the allocated space. Constants for the alignment parameter are defined in **i_vm.h**.

Description

This routine allocates unwired memory on the alignment specified by the user. The amount of memory allocated is specified by the **bytes** parameter. If the allocation fails, the system panics.

Return Values

memory_ptr — The routine returns a byte pointer to the allocated space.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

VM_PANIC_GET_UNWIRED_MEMORY — The requested memory could not be allocated.

vm_get_wired_memory

Syntax

```
pointer_to_any_type  vm_get_wired_memory  (bytes, alignment)

uint32_type         bytes;            /*READ ONLY*/
uint32_type         alignment;        /*READ ONLY*/
```

Summary

This routine allocates wired memory from available address space.

Parameters

bytes — The number of bytes to be allocated; **bytes** must be a positive value.

alignment — The byte alignment of the allocated space. Constants for the alignment parameter are defined in **i_vm.h**.

Description

This routine allocates wired memory on the alignment specified by the user. The amount of memory allocated is specified by the **bytes** parameter. If the allocation fails, the system panics.

Return Values

memory_ptr — The routine returns a byte pointer to the allocated space.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

VM_PANIC_GET_WIRED_MEMORY — The requested memory could not be allocated.

vm_map_physical_memory

Syntax

```
status_type      vm_map_physical_memory    ( logical_addr,
                                             physical_addr,
                                             num_bytes,
                                             access_mode,
                                             sharing,
                                             control_flags )

byte_address_type     logical_addr;       /*READ ONLY*/
byte_address_type     physical_addr;      /*READ ONLY*/
uint32_type           num_bytes;          /*READ ONLY*/
bit32_type            access_mode;        /*READ ONLY*/
int32_type            sharing;            /*READ ONLY*/
bit32_type            control_flags;      /*READ ONLY*/
```

Summary

This routine supports the **mmap**(2) system call.

Parameters

logical_addr — The first logical address in a contiguous block of **num_bytes** to be mapped to **physical_addr**. This address must be on a page boundary.

physical_addr — The first physical address in a contiguous block of **num_bytes** to which the **logical_addr** will be mapped. This address also must be on a page boundary.

num_bytes — The total number of bytes to be mapped, which must be an integral multiple of the logical page size.

access_mode — This is the bitwise OR of all appropriate access modes. It may contain any of **PROT_READ**, **PROT_WRITE**, and/or **PROT_EXEC**, from **sys/mman.h**. No checking will be performed as to the appropriateness of the specified modes, and it is assumed that privilege violations will not occur, or be enforced by higher level routines.

sharing — Under the current implementation, this must be **MAP_SHARED**. **MAP_PRIVATE** is not yet supported though it is specified as an option.

control_flags — This is a bit-field used to send special control information. One bit is used for cache control. This bit-field may include information for inhibiting caching which should be passed on to the page table entries of the mapped physical memory. A second bit is used to specify whether the passed logical address is to be in the user or kernel address space. The remaining bits are reserved for later expansion.

The following list describes the bit positions of the currently supported flags:

Bit 0 — This flag specifies whether user or kernel address space is to be used. Use **VM_MMAP_IS_KERNEL_ADDRESS_SPACE_MASK** to specify kernel addresses; otherwise, user addresses are assumed by default. (Bit 0 is the lowest order bit.)

Bit 1 — This flag specifies whether to write through or inhibit caching. Use **VM_MMAP_WRITE_THROUGH_MASK** to specify write-through caching on architectures that support it; otherwise, by default, no caching (cache inhibiting) is assumed.

Bits 2-31 — All other bits are unused and reserved for expansion.

Description

This routine will support the **mmap**(2) system call. It provides the kernel data structure modifications to map an area of a process's address space to real physical memory. Basically, this is done by searching for a contiguous region of a process's data area and setting pointers to the appropriate physical memory frame.

It may be called by either user or kernel processes; that is, the passed **logical_addr** may be either a kernel or user address. A bit in the **control_flags** parameter controls this distinction. The **logical_addr** passed, offset by **num_bytes**, must be part of the current address space before this call is made. Kernel processes should have already allocated unwired memory, while users should **valloc**() the appropriate range before making the mapping call.

The range addresses to be mapped must have referred to an existing region of a process's data area, or else an error will result. Any existing data that was addressed in this range will be discarded. No other explicit cleanup is needed for an address space that has been mapped. If the process that called **mmap**(2) exits, its mapped region will be implicitly unmapped by the exit path. Likewise, if the process calls any version of **exec**(2), the mapped region will also be implicitly unmapped before the new program begins to execute. Finally, in the case of a **fork**(2), the new child will implicitly inherit the parent's mapped address space.

Return Values

OK — All frames were mapped successfully.

VM_EINVAL_MMAP_UNSUPPORTED — This error will be returned if the parameter **sharing** is set to **MAP_PRIVATE**; currently, only **MAP_SHARING** is supported. The function will abort before any modifications are made.

VM_EINVAL_MMAP_BYTES_NOT_MULTIPLE — This error code will be returned if the parameter **num_bytes** is not an integral multiple (greater than zero) of the size of a physical page. The function will abort before any modifications are made.

VM_EINVAL_MMAP_BAD_ADDR_BOUNDARY — This error code will be returned if either of the address parameters, **physical_addr** or **logical_addr**, is not aligned on a page boundary. The function will abort before any modifications are made.

VM_EINVAL_MMAP_SPACE_UNALLOCATED — This error will occur when the **logical_addr** is not already a part of the current process's address space. The function will abort before any modifications are made.

VM_EINVAL_MMAP_ADDRESS_NOT_DATA — This error will be returned when the **logical_addr** is not a part of the calling process's data area. Only regions of a process's address space of the data variety will be accepted for mapping. The function will abort before any modifications are made.

VM_EINVAL_MMAP_BAD_REGION — This error will be returned in several circumstances when the range of addresses to be mapped is inappropriate. The following are specific examples of this: when **logical_addr** offset by the **num_bytes** is greater than the maximum address (4G); or when the region of addresses to be mapped, **logical_addr** to **logical_addr+num_bytes**, does not fit in the size of the process's current data area; or when **logical_addr** cannot be located in any address area. The function will abort before any modifications are made.

VM_EINVAL_MMAP_ALREADY_MAPPED — This error will be returned when any data contained within the passed range of addresses (**logical_addr** through **logical_addr+num_bytes**) is already mapped. A region of a process's data area can be remapped, but only if an explicit **munmap**(2) is done before the remap attempt. The function will abort, and this error will return before any modifications are made.

Abort Conditions

None.

vm_unmap_physical_memory

Syntax

```
status_type  vm_unmap_physical_memory   (logical_addr,
                                          num_bytes,
                                          control_flags )

byte_address_type  logical_addr;         /*READ ONLY*/
uint32_type        num_bytes;            /*READ ONLY*/
bit32_type         control_flags;        /*READ ONLY*/
```

Summary

This routine will support the **munmap**(2) system call.

Parameters

logical_addr — The first logical address in a contiguous block of **num_bytes** to be unmapped. This address must be on a page boundary.

num_bytes — The total number of bytes to be unmapped. This does not necessarily have to be identical to the number that were mapped originally; however, it must be an integral multiple of the logical page size.

control_flags — This is a bit-field used to send special control information. A bit is necessary to specify whether the passed logical address is to be in the user or kernel address space. The remaining bits may be used later, and are reserved for expandability. The following list describes the bit positions of the currently supported flags:

Bit 0 — This flag specifies whether to use user or kernel address space. Use **VM_MUNMAP_IS_KERNEL_ADDRESS_SPACE_MASK** to specify kernel addresses; otherwise, user addresses are assumed by default. Bit 0 is the lowest order bit.

Bits 1-31 — All other bits are unused and reserved for expansion.

Description

This routine will support the **munmap**(2) system call. It will **unmap** an area of a process's address space to which was previously mapped by **mmap**(2). Only previously mapped areas can be unmapped. Upon unmapping, the appropriate address space will be reset to be non-resident but will still be a valid area of the data area.

It may be called by either user or kernel processes and still work properly; that is, the passed **logical_addr** may be either a kernel or user address. A bit in the parameter **control_flags** will be used to make this distinction.

Return Values

OK — All frames were unmapped successfully.

VM_EINVAL_MUNMAP_BYTES_NOT_MULTIPLE — This error code will be returned if the parameter **num_bytes** is not an integral multiple of the physical page size. The function will abort before any modifications are made.

VM_EINVAL_MUNMAP_BAD_ADDR_BOUNDARY — This error code will be returned if the address parameter, **logical_addr**, is not aligned on a page boundary. The function will abort before any modifications are made.

VM_EINVAL_MUNMAP_BAD_REGION — This error code will be returned when several situations occur: if the range of addresses to unmap (**logical_addr** through **logical_addr+num_bytes**) is not entirely valid; if the starting address cannot be found in the process's data area; or if the address range overflows beyond the 4G upper limit.

VM_EINVAL_MUNMAP_DATA_NOT_MAPPED — This error will occur if an attempt is made to unmap any portion of an address space that has not been previously mapped. Only previously mapped regions may be unmapped.

Abort Conditions

None.

vm_mark_mod_and_ref_and_unwire_memory

Syntax

```
void  vm_mark_mod_and_ref_and_unwire_memory (start_address,
                                              is_user_address,
                                              bytes_to_unwire)


pointer_to_any_type  start_address;    /* READ ONLY */
boolean_type         is_user_address;  /* READ ONLY*/
uint32_type          bytes_to_unwire;  /* READ ONLY */
```

Summary

This routine marks the frames indicated as having been referenced and modified, and then unwires the frames.

Parameters

start_address — The byte address indicating the start of the memory to be unwired. The value in **start_address** is rounded down to a page boundary.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

bytes_to_unwire — The number of bytes to be unwired.

Description

This routine marks frames as having been referenced and modified and then unwires them. It starts at **start_address**, goes for **bytes_to_unwire** number of bytes.

Memory needs to be marked as modified if it has been wired and then used as an I/O buffer. I/O uses direct memory access, which does not cause the frame to be marked as modified automatically. Therefore, this routine will set the modified bit explicitly.

Return Values

None.

vm_mark_ref_and_unwire_memory

Syntax

```
void  vm_mark_ref_and_unwire_memory  (start_address,
                                      is_user_address,
                                      bytes_to_unwire)

pointer_to_any_type  start_address;    /*READ ONLY*/
boolean_type         is_user_address;  /*READ ONLY*/
uint32_type          bytes_to_unwire;  /*READ ONLY*/
```

Summary

This routine marks the indicated frames as having been referenced and then unwires them.

Parameters

start_address — The byte address indicating the start of the memory to be unwired.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

bytes_to_unwire — The number of bytes to be unwired.

Description

This routine marks the frames as having been referenced and then unwires them. It starts at **start_address** goes for **bytes_to_unwire** number of bytes.

Memory needs to be marked as referenced if it has been wired and then used as an I/O buffer. I/O uses direct memory access, which does not cause the frame to be marked as referenced automatically. Therefore, this routine will set the referenced bit explicitly.

Return Values

None.

vm_perhaps_get_unwired_memory

Syntax

```
pointer_to_any_type vm_perhaps_get_unwired_memory (bytes,
                                                    alignment)

uint32_type          bytes;           /*READ ONLY*/
uint32_type          alignment;       /*READ ONLY*/
```

Summary

This routine allocates unwired memory.

Parameters

bytes — The number of bytes to be allocated; **bytes** must be a positive value.

alignment — The byte alignment of the allocated space. Constants for the alignment parameter are defined in **i_vm.h**.

Description

Memory is allocated from unwired memory on the alignment specified by the user. The amount of memory allocated is specified by the **bytes** parameter.

Return Values

memory_ptr — The memory was allocated successfully.

VM_INVALID_MEMORY_PTR — The memory could not be allocated.

Exceptions

None.

vm_perhaps_get_wired_memory

Syntax

```
pointer_to_any_type  vm_perhaps_get_wired_memory  (bytes,
                                                   alignment)

uint32_type          bytes;            /*READ ONLY*/
uint32_type          alignment;        /*READ ONLY*/
```

Summary

This routine allocates wired memory.

Parameters

bytes — The number of bytes to be allocated; **bytes** must be a positive value.

alignment — The byte alignment of the allocated space. Constants for the alignment parameter are defined in **i_vm.h**.

Description

Memory is allocated from wired memory on the alignment specified by the user. The amount of memory to be allocated is specified by the **bytes** parameter.

Return Values

memory_ptr — The memory was allocated successfully.

VM_INVALID_MEMORY_PTR — The memory could not be allocated.

Exceptions

None.

vm_release_unwired_memory

Syntax

```
void  vm_release_unwired_memory   (memory_ptr, bytes)

pointer_to_any_type  memory_ptr;        /*READ ONLY*/
uint32_type          bytes;             /*READ ONLY*/
```

Summary

This routine releases unwired memory that was previously obtained via a **vm_get_unwired_memory** or **vm_perhaps_get_unwired_memory** call.

Parameters

memory_ptr — A byte pointer to the start of the memory to be released. **memory_ptr** must be the same pointer that was returned by the **vm_get_unwired_memory** or **vm_perhaps_get_unwired_memory** call when memory was originally requested.

bytes — The number of bytes to be released. **bytes** must be the same number of bytes as given to the **vm_get_unwired_memory** or **vm_perhaps_get_unwired_memory** call when memory was originally requested.

Description

This routine releases the given number of bytes of unwired memory, starting at the given byte address. This memory must have been obtained via a **vm_get_unwired_memory** or **vm_perhaps_get_unwired_memory** call.

Return Values

None.

Exceptions

None.

vm_release_wired_memory

Syntax

```
void  vm_release_wired_memory    (memory_ptr, bytes)

pointer_to_any_type  memory_ptr;        /*READ ONLY*/
uint32_type          bytes;             /*READ ONLY*/
```

Summary

This routine releases wired memory that was previously obtained via a **vm_get_wired_memory** or **vm_perhaps_get_wired_memory** call.

Parameters

memory_ptr — A byte pointer to the start of the memory that is to be released. **memory_ptr** must contain the same pointer that was returned by the **vm_get_wired_memory** or the **vm_perhaps_get_wired_memory** call when memory was originally requested.

bytes — The number of bytes to be released. **bytes** must be the same number of bytes as requested in the **vm_get_wired_memory** or **vm_perhaps_get_wired_memory** call when memory was originally requested.

Description

This routine releases the given number of bytes of wired memory, starting at the given byte address. This memory must have been obtained via a **vm_get_wired_memory** or **vm_perhaps_get_wired_memory** call.

Return Values

None.

Exceptions

None.

vm_unwire_memory

Syntax

```
void  vm_unwire_memory (start_address,
                        is_user_address,
                        bytes_to_unwire)

pointer_to_any_type  start_address;     /*READ ONLY*/
boolean_type         is_user_address;   /*READ ONLY*/
uint32_type          bytes_to_unwire;   /*READ ONLY*/
```

Summary

This routine unwires the memory indicated by **start_address** for the number of bytes indicated by **bytes_to_unwire**.

Parameters

start_address — The byte address indicating the start of the memory to be unwired.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

bytes_to_unwire — The number of bytes to be unwired.

Description

This routine unwires the memory indicated by **start_address** for the number of bytes indicated by **bytes_to_unwire**. Unwiring is done only on blocks of a complete page. Therefore, if **start_address** is not the start of a page, **vm_unwire_memory** starts at the next lowest page boundary. Similarly, if **bytes_to_unwire** does not end on a page boundary, unwiring continues into the next higher page boundary.

Return Values

None.

vm_wire_memory

Syntax

```
status_type vm_wire_memory (start_address,is_user_address,
                            bytes_to_wire)

pointer_to_any_type  start_address;   /*READ ONLY*/
boolean_type         is_user_address; /*READ ONLY*/
uint32_type          bytes_to_wire;   /*READ ONLY*/
```

Summary

This routine wires the memory indicated by **start_address** for the number of bytes indicated by **bytes_to_wire**.

Parameters

start_address — The byte address indicating the start of the memory to be wired.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

bytes_to_wire — The number of bytes to be wired.

Description

This routine wires the memory indicated by **start_address** for the number of bytes indicated by **bytes_to_wire**. Wiring is done only on blocks of a complete page. Therefore, if **start_address** is not the start of a page, **vm_wire_memory** starts at the next lowest page boundary. Similarly, if **bytes_to_wire** does not end on a page boundary, wiring continues into the next higher page boundary.

Return Values

OK — The memory was successfully wired.

[*other error statuses*] — A hard I/O error occurred that prevented a page from being brought in. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 13.

End of Chapter

Chapter 10
User-Data Access Validation Routines

This chapter describes the kernel routines that your driver uses to verify pointers to data buffers. We start with a brief introduction to buffer verification. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_sc.h** for structures beginning with the sc acronym) for a complete and current list of all constants and structures.

Overview to Using User Data Access Validation Routines

Routines in this section are used to validate user-supplied memory addresses and/or to transfer data between user memory and kernel memory. Validation routines verify that the buffer memory has permissions appropriate for the requested operation (for example, read permission is granted for a write operation). All access validation routines operate on the user address space of the calling process.

For most I/O operations, the kernel will validate user-specified buffers before performing read/write buffer operations. However, the kernel cannot validate user-specified buffers for an **ioctl** operation because **ioctl** packets may contain buffer pointers embedded in the packet. Therefore, the driver must validate user buffers itself for **ioctl** operations. The device driver validates the range of user space to and from which data is to be copied before copying the data. Many DG/UX system calls require that the range of user space be validated before attempting to write to it.

To validate the range of user space, a driver calls **sc_check_byte_access** with SC_READ_ACCESS when reading, and with SC_WRITE_ACCESS when writing. If **sc_check_byte_access** succeeds, the driver calls **sc_read_bytes_from_user** to read and **sc_write_bytes_to_user** to perform the write.

Note both **sc_read_bytes_from_user** and **sc_write_bytes_to_user** may fail if a paging I/O error occurs while the copy is in progress. This may happen even if a range of addresses has been validated. If the validation fails, drivers should abort the system call with an EFAULT status.

At the kernel level, you should be careful to read a given byte of user space memory only once during a particular system call. If the user space is part of shared memory, the memory may change between a first read and a second read. Similarly, you should write to a given byte of user space memory only once during a particular system call. Double writing may produce inconsistent results if another process is reading the memory as part of a shared memory area.

You can also use **sc_check_access_and_read_string_from_user** to copy character strings from user space, and **sc_write_string_to_user** to copy character strings to user space. In these routines, the copy is terminated when a null character is encountered or a maximum number of characters has been copied. The string routines must be used when accessing strings to avoid a double read: one read to find the end of the string for validating the access and a second to actually read in the string.

The routines described in this section are as follows:

- **sc_check_access_and_read_string_from_user**
- **sc_check_byte_access**
- **sc_read_bytes_from_user**
- **sc_write_bytes_to_user**
- **sc_write_string_to_user**

Routines beginning with sc require the **i_sc.h** include file.

Constants and Data Structures

The routines in this section use the following constants and data structures. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_sc.h** for structures beginning with the sc acronym). Chapter 1 lists the various include files.

sc_access_mode_type

```
typedef    bit16_type    sc_access_mode_type ;
```

Description

This type contains the access modes that may be specified to the functions that perform user data validation.

You may use the following defines with this type. They are defined in **i_sc.h**.

```
SC_READ_ACCESS
SC_WRITE_ACCESS
SC_EXECUTE_ACCESS
SC_NO_ACCESS
SC_ACCESS_MODE_MASK
```

sc_check_access_and_read_string_from_user

Syntax

```
status_type  sc_check_access_and_read_string_from_user
                         (buffer_ptr_ptr, dest_ptr, count_ptr)

pointer_to_any_ptr_type  buffer_ptr_ptr;  /*READ ONLY*/
pointer_to_any_type      dest_ptr;        /*WRITE ONLY*/
uint32_ptr_type          count_ptr;       /*READ/WRITE*/
```

Summary

This routine checks the user address space starting at **buffer_ptr_ptr** for **count** bytes, or through the terminating null, to verify that read access is available for the entire string. The string is also copied into the destination buffer.

Parameters

buffer_ptr_ptr — A pointer to the byte pointer that marks the start of the string for which access is to be checked.

dest_ptr — A pointer to the kernel buffer into which the string is to be copied.

count_ptr — On input, a pointer to the maximum size, in bytes, of the string, including the terminating null. On output, the size of the string copied into the kernel buffer, including the terminating null.

Return Values

OK — Read access is available for the entire string. The string has been copied to the destination with a terminating null.

SC_EFAULT_STRING_TOO_LONG — Read access is available for the maximum size of the string, but there is no terminating null in that length. The contents of the destination are undefined.

SC_EFAULT_NO_READ_ACCESS — Read access is available for less than the maximum size of the string, and no terminating null was found in the area to which read access was available. The contents of the destination are undefined.

[*other error statuses*] — The bytes could not be read because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 13.

Exceptions

None.

sc_check_byte_access

Syntax

```
status_type sc_check_byte_access (buffer_ptr_ptr,
                                  count,access)

pointer_to_any_ptr_type  buffer_ptr_ptr;  /*READ/WRITE*/
uint32_type              count;           /*READ ONLY*/
sc_access_mode_type      access;          /*READ ONLY*/
```

Summary

This routine checks the user address space starting at **buffer_ptr_ptr** for **count** bytes to verify that **access** access is available for the entire area.

Parameters

buffer_ptr_ptr — A pointer to the byte pointer that marks the start of the area for which access is to be checked.

count — The size, in bytes, of the area to be checked.

access — The access modes to be checked.

Return Values

OK — The requested access is available for the entire area.

SC_EFAULT_NO_ACCESS — One or more bytes of the specified area do not have the required access.

Exceptions

None.

sc_read_bytes_from_user

Syntax

```
status_type  sc_read_bytes_from_user (source_ptr,
                                      dest_ptr, count)

pointer_to_any_type  source_ptr;   /*READ ONLY*/
pointer_to_any_type  dest_ptr;     /*READ ONLY*/
uint32_type          count;        /*READ ONLY*/
```

Summary

This routine moves the specified number of bytes from the user's address space to the kernel address space.

Parameters

source_ptr — A pointer to the location in the user's address space from which the data is to be moved.

dest_ptr — A pointer to the location in the kernel address space to which the data is to be moved.

count — The number of bytes to be moved.

Description

The specified number of bytes are moved from the source to the destination. Access should be checked before reading.

Return Values

OK — The bytes were successfully read from the user address space into kernel address space.

[*other error statuses*] — The bytes could not be read because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 13.

Exceptions

None.

sc_write_bytes_to_user

Syntax

```
status_type  sc_write_bytes_to_user (source_ptr,
                                      dest_ptr, count)

pointer_to_any_type  source_ptr;  /*READ ONLY*/
pointer_to_any_type  dest_ptr;    /*READ ONLY*/
uint32_type          count;       /*READ ONLY*/
```

Summary

This routine moves the specified number of bytes from the kernel address space to the user's address space.

Parameters

source_ptr — A pointer to the location in the kernel address space from which the data is to be moved.

dest_ptr — A pointer to the location in the user address space to which the data is to be moved.

count — The number of bytes to be moved.

Description

The specified number of bytes are moved from the source to the destination. This routine assumes that access has already been checked.

Return Values

OK — The bytes were successfully written to the user's address space.

[*other error statuses*] — The bytes could not be written because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 13.

Exceptions

None.

sc_write_string_to_user

Syntax

```
status_type  sc_write_string_to_user (source_ptr, dest_ptr)

pointer_to_any_type source_ptr;   /*READ ONLY*/
pointer_to_any_type dest_ptr;     /*READ ONLY*/
```

Summary

This routine moves bytes from the kernel address space to the user's address space up to and including the first null byte in the source string.

Parameters

source_ptr — A pointer to the location in the kernel address space from which the data is to be moved.

dest_ptr — A pointer to the location in the user address space to which the data is to be moved.

Description

Bytes are moved from the source to the destination until a null byte is found in the source. The null is transferred to the destination. This routine assumes that access has already been checked.

Return Values

OK — The bytes were successfully written to the user's address space.

[*other error statuses*] — The bytes could not be written because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 13.

Exceptions

None.

End of Chapter

Chapter 11
Buffer Vector Management Routines

This chapter describes the kernel routines that your driver uses in manipulating buffer vectors. We start with a brief introduction to buffer vectors and how to use them. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_lm.h** for structures beginning with the **lm** acronym) for a complete and current list of all constants and structures.

Overview to Using Buffer Vectors

Buffer vectors are data structures that help you manage user-data buffers, especially buffers that are spread over non-contiguous space. The "v" system calls (**readv** and **writev**) can specify non-contiguous buffer space, while the **read** and **write** system calls specify contiguous buffers. And though buffer vectoring is most useful in the non-contiguous case, you can use the buffer vector interface for either set of read or write system calls.

A buffer vector consists of a collection (an array) of individual buffer descriptors with associated state variables. Each buffer descriptor consists of a buffer pointer and a buffer size. Individual buffer descriptors define a location from which data can be read, or into which the data can be written. Buffer vectors for contiguous space (the **read** and **write** system calls) will have a buffer vector array with only one entry (see **io_init_one_entry_buffer_vector**).

The current position within the buffer vector is maintained by the associated state variable. The current position defines where the next byte of data will be read from or written to. The current position is initialized to the first byte of the first buffer descriptor.

Buffer vector routines can be grouped into four categories: initialization, reporting of state information, modifying state information, and data transfer (reading from or writing to the buffer). All buffer vector routines take a pointer to the respective buffer vector as a parameter.

Before you can use a buffer vector with non-contiguous space, you must call **io_init_buffer_vector** to initialize it. In initialization, the buffer vector's current position is set to zero (0) and its logical address and the buffer's total size are stored in the buffer descriptor. You call **io_init_one_entry_buffer_vector** to perform the same function for a buffer vector with contiguous space and only one buffer descriptor.

There are a number of routines you can call to get information about the buffer vector and its current state. You can call **io_get_buffer_vector_position** to get the current buffer vector position and **io_get_buffer_vector_residual** to get the bytes remaining to be transferred to or from the buffer vector. You can also get the total size of all the descriptors in the buffer vector by calling **io_get_buffer_vector_byte_count**. The value returned will be equal to the current position plus the bytes remaining. Finally, you can get the address of the current buffer position and amount of contiguous bytes remaining starting at that position by calling **io_get_buffer_vector_io_info**.

Several other routines allow you to change the state information maintained in the buffer vector. You should use these routines only for exception situations such as when the buffer vector state needs to be changed due to a data transfer failure.

You call **io_reset_buffer_vector_position** to set the buffer vector's current position back to zero (0). You might use this if a data transfer failed and you want to retry the entire transfer again. You call **io_set_buffer_vector_residual** to set the **residual** variable that contains the number of bytes remaining to be transferred. Similarly, the **io_add_to_buffer_vector_position** allows you to change the current position value by a specified amount. Take care not to increase the current position by more than the number of bytes remaining in the buffer vector.

These functions perform the actual data transfer to and from the buffer vector. The current position is updated in accordance with the amount of data transferred.

You use **io_read_from_buffer_vector** and **io_write_to_buffer_vector** to transfer data between a buffer vector and a specified user buffer. However, you cannot use these routines for DMA transfers because they don't update position information. Instead you use **io_get_buffer_vector_io_info** and **io_add_to_buffer_vector_position** to produce the effect of the regular read/write routines. To do this, first get the current position and remaining bytes using **io_get_buffer_vector_io_info**, then perform the DMA, and finally update the current position with **io_add_to_buffer_vector_position**. You repeat this process as necessary given the block size of the transfer. Using **io_get_buffer_vector_io_info** and **io_add_to_buffer_vector_position** allows you to transfer directly from the device to or from the buffer vector without going though an intermediate memory buffer.

The routines described in this section are as follows:

- **io_add_to_buffer_vector_position**
- **io_get_buffer_vector_io_info**
- **io_get_buffer_vector_position**
- **io_get_buffer_vector_residual**
- **io_get_buffer_vector_byte_count**
- **io_init_buffer_vector**

- **io_init_one_entry_buffer_vector**
- **io_read_from_buffer_vector**
- **io_reset_buffer_vector_position**
- **io_set_buffer_vector_residual**
- **io_write_to_buffer_vector**

Routines beginning with **io** require the **i_io.h** include file.

Constants and Data Structures

io_buffer_vector_type

```
      typedef struct
              {
              union
              {
              io_buffer_vector_control_type      many;
              io_buffer_descriptor_type          one;
              } u;
              uint16_type                        descriptor_count;
              uint16_type                        current_descriptor;
              uint32_type                        current_offset;
              uint32_type                        total_remaining;

              }  io_buffer_vector_type ;
```

Description

This structure defines a buffer vector, which is a collection of individual buffer descriptors plus an associated state. A buffer vector may be the source or destination of a single read or write operation; the individual buffer descriptors define the locations from which the data is being read or into which the data is being written.

The current position is where the next byte of data will be read from or written to. The current position is initialized to the first byte of the first buffer descriptor. The current position within the buffer vector is maintained by the associated state.

The fields in this structure are as follows:

many — This structure contains a pointer to the array of buffer descriptors and the total of the sizes of all the elements of the array. This field of the union is used only when **descriptor_count** is non-zero. **io_buffer_vector_control_type** is described in this section.

one — This structure contains the single buffer descriptor when the buffer vector consists of a single descriptor. This field of the union is used only when **descriptor_count** is zero. **io_buffer_descriptor_type** is described later in this section.

descriptor_count — The number of entries in the **many** array of the **io_buffer_vector_control_type**. Not all of these entries are presumed valid; the **total_size** field controls the number of entries that are used. This field is used to determine the actual amount of memory allocated to the array. If this field is zero, then there is no memory allocated to the array and a single descriptor is stored in the union field **one**.

current_descriptor — The index of the descriptor that contains the current position. **io_buffer_vector_control_type** is described later in this section.

current_offset — The offset of the current position in the buffer descriptor indexed by **current_descriptor**.

total_remaining — The total number of bytes remaining to be moved to or from this buffer vector since it was initialized.

io_buffer_descriptor_type

```
typedef struct
     {
     pointer_to_any_type  buffer_ptr;
     uint32_type          size;

     }  io_buffer_descriptor_type  ;
```

Description

This structure describes a buffer from which data is to be read or to which data is to be written.

The fields in this structure are as follows:

buffer_ptr — Pointer to the start of the buffer.

size — The size of the buffer, in bytes.

io_buffer_vector_control_type

```
typedef struct
    {
    io_buffer_descriptor_ptr_type     descriptors_ptr;
    uint32_type                       total_size;

    }  io_buffer_vector_control_type  ;
```

Description

This structure is used in the **many** field of **buffer_vector_type**.

The fields in this structure are as follows:

descriptors_ptr — A pointer to an array of buffer descriptors. The array may contain as many as **UINT16_MAX** entries. (See **c_generics.h** for the definition of **UINT16_MAX**.)

total_size — The sum of the size fields in all the elements of the array buffer descriptors.

io_add_to_buffer_vector_position

Syntax

```
void io_add_to_buffer_vector_position
                                (buffer_vector_ptr, count)

io_buffer_vector_ptr_type  buffer_vector_ptr; /*READ ONLY*/
int32_type                 count;             /*READ ONLY*/
```

Summary

This routine adds the given count to the current position associated with the given buffer vector.

Parameters

buffer_vector_ptr — A pointer to the buffer vector whose current position is to be changed.

count — The number of bytes to be added to the current buffer position.

Description

This routine adds the given count to the current position associated with the given buffer vector. The amount added may be positive or negative. If the new value of the current position would be less than zero or greater than the byte count associated with the buffer vector, the result is undefined. Note that changing the current position changes the residual count by implication, so that the relationship between the current position plus residual count and the overall byte count remains true.

Return Values

None.

Exceptions

None.

io_get_buffer_vector_io_info

Syntax

```
void io_get_buffer_vector_io_info (buffer_vector_ptr,
                                   buffer_ptr_ptr, count_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr;/*READ ONLY*/
pointer_to_any_ptr_type    buffer_ptr_ptr;   /*WRITE ONLY*/
uint32_ptr_type            count_ptr;        /*WRITE ONLY*/
```

Summary

This routine takes the current buffer descriptor and returns the buffer pointer and the number of contiguous bytes left in the buffer from that pointer.

Parameters

buffer_vector_ptr — A pointer to the buffer vector whose I/O information is to be returned.

buffer_ptr_ptr — A pointer to where the buffer pointer at the current position is to be returned.

count_ptr — A pointer to where the number of contiguous bytes starting at the current position is to be returned. This returned value will always be greater than zero.

Description

This routine returns the actual buffer pointer and contiguous byte count associated with that position so that direct access I/O operations can be performed on the buffer.

Drivers can use this routine to produce the same effect as **io_read_bytes_from_buffer_vector** or **io_write_bytes_to_buffer_vector**, but with the transfer going directly between the device and the buffer vector instead of through an intermediate memory buffer. To do this, the driver successively gets the I/O information for the current position, performs direct access I/O, and updates the current position with **io_add_to_buffer_vector_position**.

NOTE: This routine must not be called when the buffer vector residual is zero, as the returned count is defined to always be strictly greater than zero.

Return Values

None.

Exceptions

None.

io_get_buffer_vector_position

Syntax

```
uint32_type io_get_buffer_vector_position
                                         (buffer_vector_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr; /*READ ONLY*/
```

Summary

This routine gets the current position of the specified buffer vector.

Parameters

buffer_vector_ptr — A pointer to the buffer vector from which the current position is to be retrieved.

Return Values

[*position*] — The current position associated with the given buffer vector.

Exceptions

None.

io_get_buffer_vector_residual

Syntax

```
uint32_type io_get_buffer_vector_residual (buffer_vector_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr;  /*READ ONLY*/
```

Summary

This routine gets the number of bytes remaining in the specified buffer vector.

Parameters

buffer_vector_ptr — A pointer to the buffer vector from which the residual byte count is to be retrieved.

Description

This routine gets the number of bytes remaining in the specified buffer vector. This residual count is always equal to the byte count of the buffer vector minus the current position. The **buffer_vector_ptr** is assumed to be valid.

Return Values

count — The residual bytes associated with the given buffer vector.

Exceptions

None.

io_get_buffer_vector_byte_count

Syntax

```
uint32_type io_get_buffer_vector_byte_count
                                            (buffer_vector_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr; /*READ ONLY*/
```

Summary

This routine gets the byte count for the specified buffer vector. This count is the number of bytes of data that this vector can hold.

Parameters

buffer_vector_ptr — A pointer to the buffer vector from which the byte count is to be retrieved.

Description

This routine gets the byte count for the specified buffer vector. This count is the number of bytes of data that this vector can hold. The **buffer_vector_ptr** is assumed to be valid.

Return Values

count — The byte count associated with the given buffer vector.

Exceptions

None.

io_init_buffer_vector

Syntax

```
void  io_init_buffer_vector (buffer_vector_ptr, total_size,
                             buffer_descriptors, count)

io_buffer_vector_ptr_type  buffer_vector_ptr;/*READ/WRITE*/
uint32_type                total_size;       /*READ ONLY*/
io_buffer_descriptor_ptr_type
                           buffer_descriptors;/*READ ONLY*/
uint16_type                count;             /*READ ONLY*/
```

Summary

This routine is used to initialize a buffer vector.

Parameters

buffer_vector_ptr — The buffer vector to be initialized.

total_size — The sum of sizes from the buffer descriptors.

buffer_descriptors — Pointer to the array of buffer descriptors to be associated with the buffer vector.

count — The number of entries in the **buffer_descriptors** array.

Description

This routine is used to initialize a buffer vector. The **buffer_vector_ptr** is assumed to be valid.

Return Values

None.

Exceptions

None.

io_init_one_entry_buffer_vector

Syntax

```
void io_init_one_entry_buffer_vector (buffer_vector_ptr,
                                      buffer_ptr, size)

io_buffer_vector_ptr_type buffer_vector_ptr;/*READ/WRITE*/
pointer_to_any_type       buffer_ptr;       /*READ ONLY*/
uint32_type               size;             /*READ ONLY*/
```

Summary

This routine is used to initialize a buffer vector that will have only one entry in the **buffer_descriptors** array.

Parameters

buffer_vector_ptr — The buffer vector to initialize.

buffer_ptr — A pointer to the buffer that is to be the sole entry in the **buffer_descriptors** array.

size — The size, in bytes, of the sole entry in the **buffer_descriptor** array.

Description

This routine is called if a buffer vector structure is being created with a single buffer descriptor entry. Using this routine to initialize a single entry buffer vector allows optimizations to be performed in buffer vector management.

Return Values

None.

Exceptions

None.

io_read_from_buffer_vector

Syntax

```
status_type io_read_from_buffer_vector (buffer_vector_ptr,
                                        buffer_ptr, count_ptr)

io_buffer_vector_ptr_type buffer_vector_ptr;  /*READ/WRITE*/
pointer_to_any_type       buffer_ptr;         /*READ/WRITE*/
uint32_ptr_type           count_ptr;          /*READ/WRITE*/
```

Summary

This routine is used to read data from the buffer vector into the specified buffer.

Parameters

buffer_vector_ptr — Pointer to the buffer vector from which data is to be read.

buffer_ptr — Pointer to where data from the buffer vector is to be placed.

count_ptr — On entry, the number of bytes to move. On exit, the actual number of bytes moved.

Description

Data is moved into the specified buffer starting at the current position of the specified buffer vector until all the data in the buffer vector has been exhausted or until **count_ptr** bytes have been moved. **count_ptr** is set to the actual number of bytes moved.

Return Values

OK — The bytes were successfully written to the buffer area.

[*other error statuses*] — The bytes could not be read because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 13.

Exceptions

None.

io_reset_buffer_vector_position

Syntax

```
void io_reset_buffer_vector_position (buffer_vector_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr; /*READ ONLY*/
```

Summary

This routine resets the current position of the buffer vector to zero.

Parameters

buffer_vector_ptr — A pointer to the buffer vector whose position is to be reset to zero.

Return Values

None.

Exceptions

None.

io_set_buffer_vector_residual

Syntax

```
void io_set_buffer_vector_residual (buffer_vector_ptr, count)

io_buffer_vector_ptr_type  buffer_vector_ptr;/*READ ONLY*/
uint32_type                count;            /*READ ONLY*/
```

Summary

This routine sets the number of bytes remaining in the specified buffer vector. The current position is unchanged.

Parameters

buffer_vector_ptr — A pointer to the buffer vector whose residual byte count is to be set.

count — The value to which to set the residual.

Description

Because the residual is always equal to the total size minus the current position, and the current position is unchanged by this routine, this routine changes the total size by implication.

Return Values

None.

Exceptions

None.

io_write_to_buffer_vector

Syntax

```
status_type io_write_to_buffer_vector (buffer_ptr,
                                buffer_vector_ptr, count_ptr)

pointer_to_any_type         buffer_ptr;        /*READ ONLY*/
io_buffer_vector_ptr_type   buffer_vector_ptr;/*READ/WRITE*/
uint32_ptr_type             count_ptr;           /*READ/WRITE*/
```

Summary

This routine is used to write data from the specified buffer into the buffer vector.

Parameters

buffer_ptr — Pointer to the buffer from which data is to be read.

buffer_vector_ptr — Pointer to the buffer vector to which data is to be written.

count_ptr — On entry, the number of bytes to be moved. On exit, the actual number of bytes moved.

Description

Data is moved into the buffer vector. The transfer starts at the beginning of the specified buffer and goes until the end of the buffer vector has been reached or until **count_ptr** bytes have been moved. **count_ptr** is set to the actual number of bytes moved.

Return Values

OK — The bytes were successfully written to the buffer area.

[*other error statuses*] — An error terminated the write operation. The list of possible errors is too long to give here. You can decode any status returned here using the status decoding methods described in Chapter 13.

Exceptions

None.

End of Chapter

Chapter 12
Configuration Routines

This chapter describes the kernel routines you can use during set up and configuration. We start with a brief introduction to configuration routines and how the routines are used. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_io.h** for structures beginning with the **io** acronym) for a complete and current list of all constants and structures.

Overview to the Configuration Process and Configuration Routines

The routines in this section are used to perform a number of different configuration tasks required in your driver's configuration routine (we'll call it **xxx_configure**). The system build process creates a list of devices to be configured from the entries in the system file. At boot time, the system initialization code scans this list and invokes the driver's **xxx_configure** routine for each device of the driver's type in that list. The initialization code passes **xxx_configure** a pointer to the device's device specification and its major number. The specification should be in the standard format shown below:

device_mnemonic [@*device_code*] ([*parameters*])

At a minimum, each driver's **xxx_configure** routine should: verify that the device specification it receives is valid; test for device existence, and, if successful, register the device with various parts of the kernel.

The driver must first verify that the device specification identifies a device of its type. It does this by parsing the specification using **io_parse_device_spec**. The **io_parse_device_spec** routine returns the device name, device code, and parameters imbedded in the specification.

If the device specification identifies one of the driver's devices and it is a physical device, the driver should verify that the device is attached and working. It is important to use the kernel-supplied routines for the first access because if a device is not present when an attempt is made to address it directly, a memory fault and system panic may occur. The driver can verify that the device exists by reading the device's I/O registers using **io_do_first_long_board_access** for 32-bit registers or **io_do_first_short_board_access** for 16-bit registers. If verification is successful, the driver can use **io_check_device_spec** to check that the device's address and device code are not already in use. If the address and device code are not in use, call **io_check_device_spec** to reserve them for the current driver.

If the device exists, the driver must perform several steps to appropriately register it with the system. If the device generates interrupts, the driver must register the interrupt service routine with the kernel. This is done by calling **io_register_device_info**. Once **io_register_device_info** is called, any interrupt generated by the device will cause the driver's interrupt service routine to be invoked. Thus, it is important that the driver not register the device until the driver is ready to handle interrupts. The driver can read the device registration information by calling **io_get_device_info**.

The system passes **xxx_configure** a major number that identifies the driver's position in the kernel's driver lookup table. The driver must then allocate a minor number that specifies the particular device's location in the kernel's device tables. The driver can allocate a minor number by calling **io_allocate_device_number**. This routine returns the next available minor number and also puts a parameter supplied by the driver into the device table. Typically, the parameter is a pointer to a data structure associated with the device. The driver can retrieve the stored parameter by calling **io_map_device_number** with the major and minor device number as parameters. Disk drivers may also call **io_add_to_register_list** to have their disk implicitly registered. Implicitly registered disks are known to the file system even if they are not mounted.

The driver creates device nodes in the **/dev** directory dynamically during system initialization. The driver can call **fs_submit_dev_request** to create a device node for each device.

Most of the registration routines described above have a corresponding deregistration routine which should be used during the **xxx_deconfigure** routine or if the device fails any portion of the configuration process. Examples are: **io_deallocate_device_number**, **io_deregister_device_info**, and **io_forget_device_spec**.

The routines described in this section are as follows:

- **fs_submit_dev_request**
- **io_add_to_register_list**
- **io_allocate_device_number**
- **io_deallocate_device_number**
- **io_register_device_info**
- **io_deregister_device_info**
- **io_check_device_spec**
- **io_forget_device_spec**
- **io_do_first_short_board_access**
- **io_do_first_long_board_access**

- **io_get_device_info**
- **io_map_device_number**
- **io_parse_device_spec**
- **io_perform_reset**

Routines beginning with **fs** and **io** require the **i_fs.h** and **i_io.h** include files, respectively.

Constants and Data Structures

The routines in this section use the following constants and data structures. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, since these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_fs.h** for structures beginning with the **fs** acronym). Chapter 1 lists the various include files.

fs_dev_request_type

```
typedef struct
    {
    fs_dev_request_operation_enum_type  operation;
    char_type  dirname[33];
    char_type  filename[33];
    union {
          fs_dev_create_request_type  create;
          }
    op;
    }
    fs_dev_request_type;
```

Description

This structure contains the information required to change a node in **/dev**. The fields in this structure are as follows:

operation — The type of operation requested; for example, delete or create.

dirname — The directory in which the node should reside. This name will be appended to **/dev/**. For example, set **dirname** to "rdsk" to create a node in **/dev/rdsk**. If you want the node in a **/dev** and not a subdirectory, set **dirname**[0] to **FS_NULL_CHAR**.

filename — The filename of the node.

op — The information necessary for the operation requested.

fs_dev_request_operation_enum_type

```
typedef enum
        {
        Fs_Dev_Request_Operation_Create,
        Fs_Dev_Request_Operation_Delete
        }
             fs_dev_request_operation_enum_type  ;
```

Description

This enum type contains the valid operations supported by the **/dev** manager. The fields in this structure are as follows:

Fs_Dev_Request_Operation_Create — Request to create a node in **/dev**. See **fs_dev_create_request_type**.

Fs_Dev_Request_Operation_Delete — Request to delete a node from /dev

fs_dev_create_request_type

```
typedef struct
        {
        io_device_number_type    device;
        df_file_mode_type        mode_bits;
        }
             fs_dev_create_request_type  ;
```

Description

This structure contains the information required to create a node in **/dev**. The fields in this structure are as follows:

device — The device number of the node.

mode_bits — The initial mode bits of the node. This includes the file type information.

io_dev_adapt_info_type

```
typedef struct
     {
     char_ptr_type         name;
     io_device_code_type   device_code;
     char_ptr_type         params[IO_DEV_ADAPT_MAX_PARAMS];
     char        device_spec[IO_DEV_ADAPT_MAX_SPEC_SIZE];

     }  io_dev_adapt_info_type  ;
```

This structure provides a method to pass data back from the **i_io_parse_dev_spec** routine. The fields in this structure are as follows:

name — A pointer to the null terminated string of a device or adapter name.

device_code — The device code.

params — An array of pointers to null terminated strings for each of the parameters.

device_spec — A copy of the device specification where the **name**, and **params** pointers will point.

uc_device_class_enum_type

```
typedef enum
     {
     Uc_Integrated_Device_Class =   0,
     Uc_Vmebus_Device_Class =       1,
     Uc_Invalid_Device_Class =      2,

     }    uc_device_class_enum_type  ;
```

Description

This type describes the classes of devices supported by the DG/UX kernel. A device is uniquely identified by its interrupt class and device code.

As new classes of devices are supported, this type definition will change. Check the **i_uc.h** (in **/usr/src/uts/aviion/ii**) include file for the latest supported classes.

uc_device_code_type

```
typedef  uint32_type                uc_device_code_type  ;
```

Description

This type is used to describe a device code, which, along with its associated device class, is used to identify an I/O device.

Device codes must be unique within a class, but the same value device code can be found in multiple classes. Thus, device codes are fit to the device class to which they apply.

The device codes for integrated devices are pre-defined and will be the same across all architectures. Note that there is no association between the pre-defined integrated device codes and physical hardware. The kernel will map the pre-assigned device code to the device interrupt on a given machine.

VMEbus class devices do not have pre-assigned device codes because the VME interrupt vector mechanism allows devices to be set up to use any valid VME vector. In the **VMEbus** class, the device code is the value of the VME vector. It is up to drivers to register device information specifying the appropriate VME vector to the kernel. When a VME class interrupt occurs, the kernel will return the VME vector of the interrupting device.

Integrated Device Code Literals

This section defines the values for the integrated device type pre-assigned device codes. The values below apply for all machine architectures. During driver initialization, a device's driver links its device code with an interrupt handler by registering the device. Use the following literals as the device codes for integrated class devices:

```
UC_SYSTEM_ERROR_DEVICE_CODE
UC_SYSTEM_TIMER_DEVICE_CODE
UC_KEYBOARD_DEVICE_CODE
UC_ DUART_0_DEVICE_CODE
UC_DUART_1_DEVICE_CODE
UC_PARALLEL_PORT_DEVICE_CODE
UC_ETHERNET_0_DEVICE_CODE
UC_ETHERNET_1_DEVICE_CODE
UC_SCSI_0_DEVICE_CODE
UC_DMA_TERMINAL_COUNT_DEVICE_CODE
UC_GRAPHICS_CARD_DEVICE_CODE
UC_CROSS_INTERRUPT_DEVICE_CODE
UC_PER_JP_TIMER_DEVICE_CODE
UC_DUART_TIMER_DEVICE_CODE
UC_SIGHP_DEVICE_CODE
UC_LOCATION_MONITOR_DEVICE_CODE
UC_POWER_FAIL_DEVICE_CODE
UC_ZBUFFER_DEVICE_CODE
UC_SCSI_1_DEVICE_CODE
UC_SYNC_0_DEVICE_CODE
UC_SYNC_1_DEVICE_CODE
```

fs_submit_dev_request

Syntax

```
void   fs_submit_dev_request (dev_request_ptr)

fs_dev_request_ptr_type  dev_request_ptr;  /*READ ONLY*/
```

Summary

This routine is used to submit a request to create or delete a **/dev** entry. If the root is not mounted, then the request will not be performed until the root is mounted.

Parameters

dev_request_ptr — A pointer to the necessary information to manipulate a **/dev** entry. For the create operation, this information includes the file's major and minor device numbers, mode bits, type (block or character), containing directory (for example, "." or "rdsk") and the filename of the new file (for example, "tty05"). For the delete operation, only the filename and containing directory fields are required.

Description

A request to manipulate a **/dev** entry is accepted. The request will be processed immediately if the root has been mounted. Otherwise, the request is added to a queue for later processing.

Return Values

None.

io_add_to_register_list

Syntax

```
void  io_add_to_register_list  (device_number)

io_device_number_type   device_number;     /*READ ONLY*/
```

Summary

This routine adds the specified device to the list of disks that will be implicitly registered as part of system initialization. This routine is optional and is used only with disks.

Parameters

device_number — Device number of the disk to be registered.

Description

Registration makes a physical disk known to the file system and the logical disk manager. Thus, implicitly registered disks are known to the file system without being specifically mounted.

Return Values

None.

Exceptions

None.

io_allocate_device_number

Syntax

```
status_type io_allocate_device_number (major, handle,
                                       unit, minor_ptr)

io_major_device_number_type     major;     /*READ ONLY*/
bit32e_type                     handle;    /*READ ONLY*/
uint16_type                     unit;      /*READ ONLY*/
io_minor_device_number_ptr_type minor_ptr; /*WRITE ONLY*/
```

Summary

This routine assigns the device a minor device number. The major device number identifies the family of devices to which the device belongs.

Parameters

major — The device's major device number.

handle — The device handle which identifies the device to its driver.

unit — The unit number that identifies the device to its controller.

minor_ptr — A pointer to the location where the allocated minor device number is returned.

Description

See Summary.

Return Values

OK — No errors were discovered, so all returned arguments are valid.

IO_ENXIO_ALL_MINOR_NUMBERS_IN_USE — The minor device number table for this major device number contains no unused slots and has grown to the maximum size.

Exceptions

None.

Abort Conditions

None.

io_deallocate_device_number

Syntax

```
void io_deallocate_device_number (device_number)

io_device_number_type  device_number;   /*READ ONLY*/
```

Summary

This routine terminates the association between the device and its minor device number.

Parameters

device_number — Contains the major and minor device numbers of the device being deconfigured.

Description

See Summary.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error codes:

IO_PANIC_MAJOR_NUMBER_EXCEEDS_MAX — An invalid major device number was used.

IO_PANIC_DEVICE_IS_NOT_CONFIGURED — An attempt was made to deallocate a device which was not configured.

IO_PANIC_DEVICE_IS_NOT_CONFIGURED2 — An active entry in the minor device number table does not exist at the offset specified by the minor device number argument.

io_deregister_device_info

Syntax

```
void   io_deregister_device_info  (dev_code, dev_class)

io_device_code_type               dev_code;  /*READ ONLY*/
uc_device_class_enum_type         dev_class; /*READ ONLY*/
```

Summary

This routine deregisters the device by removing its current interrupt handler and device information structure from the DIT.

Parameters

dev_code — device code for which the current interrupt handler is to be disassociated.

dev_class — device class for which the current interrupt handler is to be disassociated.

Description

This routine reverses the effect of **io_register_device_info**. It deregisters the device by removing its current interrupt handler and device information structure from the DIT. After this call completes, future interrupts on the specified device code will be directed to the system supplied "nodevice" interrupt handler. If you make this call on a device code that does not currently have an interrupt handler, a panic will occur.

Return Values

None.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

IO_PANIC_ILLEGAL_DEREGISTER_DEVICE_INFO — An attempt was made to deregister a device on a device code that did not have information registered.

io_check_device_spec

Syntax

```
status_type  io_check_device_spec (device_address,
                                    device_code)

opaque_ptr_type       device_address;       /*READ ONLY*/
io_device_code_type   device_code;          /*READ ONLY*/
```

Summary

This routine checks that the address and device code specified for the device are not already in use.

Parameters

device_address — The address of the primary registers for the device.

device_code — The device code for the device.

Description

This routine checks that the address and device code specified for the device are not already in use. Such address and device code validation will not prevent overlap of registers or RAM areas. It does help avoid the most common user errors in device specification. Only the first address for a device is checked.

Return Values

OK — The address and device code are not already in use.

IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED — The address or device code are already in use.

Exceptions

None.

io_forget_device_spec

Syntax

```
status_type  io_forget_device_spec     (device_address,
                                        device_code)

opaque_ptr_type       device_address;   /* READ ONLY  */
io_device_code_type   device_code;      /* READ ONLY  */
```

Summary

Release (that is, forget) a device specification that was claimed as the result of a previous call to the **io_check_device_spec** routine.

Parameters

device_address — The address of the primary registers for the device.

device_code — The device code for the device.

Description

When a device is deconfigured, the **device_address** claimed for the device must be freed by calling this routine. If you do not free the device address, calls to **io_check_device_spec** using this device address will fail.

Return Values

OK — The address/device code pair is freed.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — The address/device code to be freed were not found.

Exceptions

None.

io_do_first_short_board_access

Syntax

```
status_type io_do_first_short_board_access (register_ptr,
                                            register_contents_ptr,
                                            write_to_register)

bit16e_ptr_type    register_ptr;          /*READ/WRITE*/
bit16e_ptr_type    register_contents_ptr; /*READ/WRITE*/
boolean_type       write_to_register;     /*READ ONLY*/
```

Summary

This routine tests for the existence of the board at a particular memory-mapped I/O address. Use this routine for boards with short (16-bit) registers.

Parameters

register_ptr — A pointer to the register on the board to be accessed.

register_contents_ptr — A pointer a one-word read/write buffer. For a write operation, the contents of this buffer will be written to the register. For a read operation, the data read from the register will be stored in this buffer.

write_to_register — A boolean indicating whether the operation is read or write. When it is TRUE, the routine writes to the register. When it is FALSE, the routine reads from the register.

Description

Do the first access to a board register such that if a board is not present, the system will not hang or panic. The board should not be accessed again if IO_ENXIO_DEVICE_DOES_NOT_EXIST is returned. This routine assumes that the register is a short register.

Return Values

OK — The register was accessed successfully.

IO_ENXIO_DEVICE_DOES_NOT_EXIST — The board is not accessible.

Exceptions

None.

io_do_first_long_board_access

Syntax

```
status_type io_do_first_long_board_access (register_ptr,
                                           register_contents_ptr,
                                           write_to_register)

bit32e_ptr_type    register_ptr;            /*READ/WRITE*/
bit32e_ptr_type    register_contents_ptr;   /*READ/WRITE*/
boolean_type       write_to_register;       /*READ ONLY*/
```

Summary

This routine tests for the existence of the board at a particular memory-mapped I/O address. Use this routine for boards with long (32-bit) registers.

Parameters

register_ptr — A pointer to the register on the board to be accessed.

register_contents_ptr — A pointer to the contents of the given register. On input, if **write_to_register** is TRUE, this value will be written to the register. On output, when write_to_register is FALSE, this value will be the value read from the register.

write_to_register — A boolean indicating, when TRUE, to write to the register. Otherwise a read will be done.

Description

Do the first access to a long board register such that if a board is not present, the system will not hang or panic. The board should not be accessed again if IO_ENXIO_DEVICE_DOES_NOT_EXIST is returned. This routine assumes that the register is a long register.

Return Values

OK — The register was accessed successfully.

IO_ENXIO_DEVICE_DOES_NOT_EXIST — The board is not accessible.

Exceptions

None.

io_get_device_info

Syntax

```
status_type io_get_device_info (dev_code, dev_class,
                                interrupt_handler,
                                dit_entry_ptr)

io_device_code_type          dev_code;          /*READ ONLY*/
uc_device_class_enum_type    dev_class;         /*READ ONLY*/
io_service_interrupt_routine_ptr_type
                             interrupt_handler;/*READ ONLY*/
word_address_ptr_type        dit_entry_ptr;     /*WRITE ONLY*/
```

Summary

This routine retrieves the device information pointer associated with the device specified by the device code and device class.

Parameters

dev_code — The device code of the device for which the device information pointer is to be retrieved.

dev_class — The device class of the device for which class the device information pointer is to be retrieved.

interrupt_handler — The service interrupt routine pointer stored at the beginning of the device information structure. This argument is used to ensure that the device information pointer returned by this routine really does belong to the requestor.

dit_entry_ptr — A pointer to where the device information pointer is to be returned.

Description

The device information pointer registered with the specified device is retrieved. If the specified device code has no device information registered to it, or if the service interrupt routine pointer in the device information structure does not match the service interrupt routine pointer supplied as an argument to this call, then an error status is returned and the returned device information pointer is undefined.

Return Values

OK — The device information pointer was successfully returned.

IO_ENXIO_DEVICE_CODE_OUT_OF_RANGE — The supplied device code is not supported on this system.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — No device information pointer was found for the device code or the device code does not belong to the requestor.

Exceptions

None.

io_map_device_number

Syntax

```
status_type  io_map_device_number (device_number,
                                   handle_ptr, unit_ptr)

io_device_number_type  device_number;     /*READ ONLY*/
bit32e_ptr_type        handle_ptr;        /*WRITE ONLY*/
uint16_ptr_type        unit_ptr;          /*WRITE ONLY*/
```

Summary

This routine translates the previously allocated major and minor device numbers to device handle and unit number.

Parameters

device_number — Contains the major and minor device numbers of the device.

handle_ptr — Pointer to the location where the device handle is returned.

unit_ptr — Pointer to the location where the unit number is returned.

Description

This routine is typically called by a driver's open routine to map the major and minor device numbers to a specific device.

Return Values

OK — No errors occurred.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — An attempt was made to map a device that is not configured.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_MAJOR_NUMBER_EXCEEDS_MAX — The major device number argument exceeds the maximum specified by **cf_io_device_driver_count.**

IO_PANIC_MAJOR_NUMBER_EXCEEDS_MAX2 — The major device number argument exceeds the maximum specified by **cf_io_major_number_count**.

io_parse_device_spec

Syntax

```
boolean_type  io_parse_device_spec      (spec_ptr,
                                         dev_adapt_info_ptr,
                                         spec_size_ptr)
char_ptr_type                  spec_ptr;          /*READ ONLY*/
io_dev_adapt_info_ptr_type     dev_adapt_info_ptr;/*WRITE ONLY*/
int32_ptr_type                 spec_size_ptr;     /*WRITE ONLY*/
```

Summary

Parse the device or adapter specification string for the positions of all specification components.

Parameters

spec_ptr — Pointer to a null terminated device or adapter specification string.

dev_adapt_info_ptr — Pointer to a structure where the pointers to the parsed string are to be returned.

spec_size_ptr — Pointer to the location where the length of the parsed device/adapter specification is returned. This location remains unchanged on error.

Description

This routine parses a device or adapter specification string (null terminated) into components. The components parsed for are: the device/adapter name, device code, and up to IO_DEV_ADAPT_MAX_PARAMS parameters. The parse leaves the original string intact. If a given component was not present, its pointer will point to a null character. Upon successful parsing, the length, in bytes, of the parsed specification will be returned in **spec_size_ptr**.

At a minimum the device/adapter specification must consist of a sequence of characters followed by an open and a close parenthesis. If a device code is present it must be prefixed with an IO_DEV_ADAPT_DEVICE_CODE_DELIMITER (at-sign, @), consist of two hexadecimal characters, and occupy the space immediately in front of the open parenthesis. Any number of parameters up to IO_DEV_ADAPT_MAX_PARAMS may be present, but they must be separated by commas. For more detailed information about the device and adapter specification, refer to Chapter 1. If the parsing fails, then all information within the **dev_adapt_info** structure must be assumed to be invalid.

Return Values

TRUE — The specification was successfully parsed.

FALSE — The parsing failed and the state of the **spec_ptr** string and the **dev_adapt_info_ptr** structure elements are unknown.

io_perform_reset

Syntax

```
void io_perform_reset  (reset_variety)

uc_reset_enum_type      reset_type;   /*READ ONLY*/
```

Summary

This routine performs the specified type of reset.

Parameters

reset_variety — An enumeration specifying which type of reset is to be done.

Description

This routine performs the specified type of reset. It uses the clock and await mechanisms, so it should not be used in an environment where this is not possible (for example, during shutdown or after reset).

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error codes:

IO_PANIC_BAD_RESET_TYPE — The parameter passed is not recognizable.

io_register_device_info

Syntax

```
status_type  io_register_device_info (dev_code,dev_class,
                                      info_ptr)

io_device_code_type        dev_code;  /*READ ONLY*/
uc_device_class_enum_type  dev_class; /*READ ONLY*/
word_address_type          info_ptr;  /*READ ONLY*/
```

Summary

This routine associates a pointer given in **info_ptr** with the device specified by the device code and device class. This process establishes an interrupt handler for the given device code.

Parameters

dev_code — The device code of the device with which a device information structure is to be associated.

dev_class — The device class of the device with which a device information structure is to be associated.

info_ptr — A pointer to the device information structure to be associated with the specified device code. The device information structure must contain a pointer to an interrupt handler as the first field. This interrupt handler becomes the handler for interrupts from the specified device code.

Description

This routine creates an entry in the appropriate device class device interrupt table (DIT) for the device code. If the slot in the DIT is already occupied or if the device code is larger than the maximum device code supported on this system, then an error is returned and the association between the device code and device information structure is NOT established.

Return Values

OK — The **device_info** was successfully registered.

IO_ENXIO_DEVICE_CODE_OUT_OF_RANGE — The supplied device code is not supported on this system. The **device_info** is not registered.

IO_ENXIO_DEVICE_CODE_ALREADY_ASSIGNED — An attempt was made to configure a device on a device code that is already assigned.

Exceptions

None.

End of Chapter

Chapter 13
Driver Daemon, Generic Daemon And Error Processing Routines

This chapter describes the DG/UX kernel routines used for sending messages to the driver and generic daemons and to the error logging facility. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_io.h** for structures beginning with the **io** acronym) for a complete and current list of all constants and structures.

The routines described in this section are:

- **io_queue_message_to_driver_demon**
- **io_specify_max_demon_messages**
- **io_queue_message_to_generic_demon**
- **io_specify_max_generic_demon_messages**
- SC_ENCODE_STATUS
- **io_err_log_error**

Routines beginning with sc require the **i_sc.h** include file and those beginning with **io** require **i_io.h**.

Overview to Driver Daemon and Generic Daemon Routines

Driver daemons and Generic daemons are classes of daemon processes that drivers use in asynchronous I/O requests. They provide the recommended path for completing asynchronous processing after the interrupt handler.

The two classes of daemon processes have exactly the same interface and method of operation. Each class has a global queue on which requests are placed. You put requests on these queues by calling **io_queue_message_to_driver_demon** or **io_queue_message_to_generic_demon.** Requests consist of a pointer to a routine to execute and an argument to be passed to the routine. Daemon processes continuously remove and process entries from the request queue by calling the routine with the specified argument. Because more than one daemon process may be removing requests from the same queue, multiple requests may be executed in parallel on systems with multiple processors. Each request, however, is only executed once and by a single daemon. All the daemon processes working off the same queue are in the same class. Note that the requestor's routine will run in the daemon's context not in the requestor's context.

The two classes of daemons differ in what kinds of operations the routine in the request may perform. Routines in Driver Daemon requests must not perform any operation that might have to wait for the completion of a disk I/O operation. For example, such routines may not cause a page fault, because servicing the page fault may require waiting for a disk I/O to complete. In addition, such routines must not directly or indirectly send signals or perform terminal-related operations. Because of all these restrictions, the Driver Daemons will generally only be used by disk device drivers.

Routines in Generic Daemon requests are allowed to wait on disk I/O, send signals, and perform terminal-related operations. The lesser restrictions make the Generic Daemons usable by terminal-handling code and other higher level parts of the system.

CAUTION: Disk device drivers must not use the Generic Daemons because a deadlock condition could result.

As with time-outs, the kernel needs a certain amount of space to process daemon messages and this space is allocated dynamically at run-time. You must declare the maximum space needed by calling **io_specify_max_demon_messages** (for driver daemon messages) and/or **io_specify_max_generic_demon_messages** (for generic daemon messages) before you send any messages to the corresponding daemon. Once you have allocated the maximum number of messages, you may not request more and you must not exceed the number specified.

Error Encoding and Logging Routines

This section describes a macro you can use to create system-compatible error numbers for your device's errors and a routine you can use to log errors to the system error facility.

You use the **io_err_log_error** routine to queue your driver's error messages on the pseudo-device **err**(7) until they can be retrieved by the system error daemon and written to system error log. You pass your message to **io_err_log_error** in the form of a **printf** string with a format parameter and accompanying variables.

The error encoding macro helps you integrate system-compatible **errnos** into your status codes. A compatible **errno** can be passed all the way back to the user level. In normal processing, once a status is sent to the user-level, the **errno** is extracted from the status and returned to the user.

To create a status containing an **errno**, use the following format:

SS_EEEE_DDDDDD

Here, SS is a subsystem identifier; your device driver will use "DEV" if it is a standard driver and "SFM" if it is a STREAMS driver. EEEE is the full name of the

errno to be returned to the user; use standard **errnos** found in **errno.h**. DDDDDD is a description of the state that caused the status to be returned. An example of a status code is as follows:

IO_EIO_DEVICE_TIMED_OUT

If you do not want to use the status to return an **errno** to the user, pass SC_NO_ERRNO to this macro. Higher levels of code will deal with the status before it gets back to the user.

Whenever possible, use I/O statuses already defined in **dev_status_codes.h** found in **aviion/dev**. For statuses that you will handle within your driver, use DEV as the subsystem, SC_NO_ERRNO as the **errno** and simply chose a status number that is higher than the last one used in **dev_status_codes.h**. The DEV_ENCODE macro in **dev_status_codes.h** will set up the status for you correctly.

Note that the convention through the rest of the kernel is to use STATUS instead of the EEEE **errno** when no errno is used. For example, IO_STATUS_REQUEST_STILL_IN_PROGRESS will not return a status to the user. An example of how to create a new status for your device is as follows:

```
#define DEV_STATUS_FOO_DEVICE_IN_BAR_STATE DEV_ENCODE(SC_NO_ERRNO,0107)
```

Constants and Data Structures

This section defines the "no error" constant.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_sc.h** for structures beginning with the sc acronym).

SC_NO_ERRNO

```
#define SC_NO_ERRNO               0
```

Use this value to indicate that the status does not contain an **errno** value.

io_queue_message_to_driver_demon

Syntax

```
vp_ec_ptr_type  io_queue_message_to_driver_demon
                    (completion_routine_ptr, data, do_advance)

io_completion_routine_ptr_type completion_routine_ptr;
                                                    /*READ ONLY*/
bit32e_type                     data;         /*READ ONLY*/
boolean_type                    do_advance;/*READ ONLY*/
```

Summary

This routine queues a message to the Driver Daemon.

Parameters

completion_routine_ptr — A pointer to the value to go in the **completion_routine** field of the message. When the Driver Daemon dequeues this message, it will call the routine pointed to by the **completion_routine_ptr** field.

data — The value to go in the **data** field of the message. The Driver Daemon will use this value as a parameter when it calls the routine pointed to by **completion_routine_ptr**.

do_advance — A boolean indicating whether to advance the Driver Daemon eventcounter. See Description below.

Description

This routine queues a message to the I/O Driver Daemon. A free message is allocated from the Driver Daemon free list, filled in with the arguments given, and queued to the I/O Driver Daemon queue.

If **do_advance** is TRUE and the queue is empty, the null eventcounter pointer is returned and the daemon eventcounter will be advanced by one.

If **do_advance** is FALSE, the daemon eventcounter is not advanced under any circumstances. Rather, if the message queued is the only message in the queue, the address of the daemon eventcounter is returned. Otherwise, the null eventcounter pointer is returned.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_DEMON_FREE_LIST_EMPTY — A free message could not be allocated from the Driver Daemon free list when needed. A device driver has used more messages than the number of messages it requested to be allocated for the daemon. See **io_specify_max_demon_messages**.

Remarks

The **do_advance** boolean is needed to handle timeouts. When a driver's timeout routine queues a message to the daemon, the eventcounter must not be advanced because the await table lock is already held by the await table routine that found the timeout entry. Instead, the eventcounter address is passed all the way back to the await table code, which will perform the advance when the await table is unlocked.

io_specify_max_demon_messages

Syntax

```
void    io_specify_max_demon_messages  (count)

uint32_type  count;        /*READ ONLY*/
```

Summary

This routine defines the maximum number of messages that the calling driver can have in the daemon's queue simultaneously.

Parameters

count — The maximum number of messages. The **count** parameter must be a positive integer; it is not possible to reduce the maximum number of messages.

Description

This routine allocates space for the specified number of messages and adds them to the daemon's free queue. It must be called by each device driver before that driver sends a message to the daemon. A given driver may make this call more than once if the maximum number of messages grows. The maximum number of messages may not be reduced.

In general, the maximum number of messages a driver will need depends on the number of devices it must service and on the way the driver handles and clears interrupts from those devices.

Return Values

None.

Exceptions

None.

io_queue_message_to_generic_demon

Syntax

```
  vp_ec_ptr_type  io_queue_message_to_generic_demon
                                    (completion_routine_ptr,
                                     data, do_advance)

  io_completion_routine_ptr_type
                  completion_routine_ptr;/*READ ONLY*/
  bit32e_type     data;                  /*READ ONLY*/
  boolean_type    do_advance;            /*READ ONLY*/
```

Summary

This routine queues a message to the Generic Daemon.

Parameters

completion_routine_ptr — The value to go in the **completion_routine** field of the message.

data — The value to go in the data field of the message.

do_advance — A boolean indicating whether to advance the generic daemon eventcounter. See "Description" below.

Description

This routine queues a message to the Generic Daemon. A free message is allocated from the Generic Daemon free list, is filled in with the arguments given, and queued to the Generic Daemon queue.

If the **do_advance** boolean is TRUE, the return value will be the null eventcounter pointer and the Generic Daemon eventcounter will be advanced if the message queue is the only message in the queue.

If the **do_advance** boolean is FALSE, the Generic Daemon eventcounter is not advanced under any circumstances. Rather, if the message queued is the only message in the queue, the address of the demon eventcounter is returned. Otherwise, the null eventcounter pointer is returned.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error codes:

IO_PANIC_GENERIC_DEMON_FREE_LIST_EMPTY — A free message could not be allocated from the Generic Daemon free list when needed. The device driver has used more messages than the number of messages it requested to be allocated for the daemon.

io_specify_max_generic_demon_messages

Syntax

```
void   io_specify_max_generic_demon_messages  (count)

uint32_type  count; /*READ ONLY*/
```

Summary

This routine informs the Generic Daemon of the maximum number of messages that the calling driver will have in the daemon's queue.

Parameters

count — The maximum number of messages. This value must be a positive integer. Once **count** has been set you can add to but not reduce the maximum number of messages.

Description

This routine allocates space for the specified number of messages and adds them to the Generic Daemon's free queue. It must be called by each device driver before that driver sends a message to the Generic Daemon. A given driver may make this call more than once if the maximum number of messages grows. However, the maximum number of messages may not be reduced.

In general, the maximum number of messages a driver will need depends on the number of devices it must service and on the way the driver handles and clears interrupts from those devices.

Return Values

None.

Exceptions

None.

SC_ENCODE_STATUS

Syntax

```
#define    SC_ENCODE_STATUS (subsystem_id, errno, sequence)

(status_type)((subsystem_id << 18) + (errno << 9) + sequence))
```

Summary

This macro constructs a status value from the subsystem ID for a subsystem, the **errno** to be inserted into the status, and a sequence number to distinguish multiple statuses with the same subsystem ID.

Parameters

subsystem_id — The subsystem ID for the subsystem.

errno — The **errno** that is to be inserted into the status. The value of errno must be less than or equal to 511.

sequence — A sequence number to distinguish multiple statuses with the same subsystem ID. The sequence number must have a value between 1 and 511.

Description

The status is constructed so the sequence number occupies bits 0-8, the errno occupies bits 9-17, and the subsystem ID occupies bits 18-26. Bits 27-31 are unused and set to 0. The **errno** parameter specifies the **errno** value that will be returned to the user. To get the subsystem ID and sequence numbers, the user should call the **dg_ext_errno** system call.

Return Values

status — The newly encoded status.

io_err_log_error

Syntax

```
boolean_type  io_err_log_error  (priority, format,
                                 value_00,value_01,value_02,
                                 value_03,value_04,value_05,
                                 value_06,value_07,value_08,
                                 value_09,value_10,value_11,
                                 value_12,value_13,value_14,
                                 value_15,value_16,value_17)

uint32e_type  priority;         /* READ ONLY  */
char_ptr_type format;           /* READ ONLY  */
bit32e_type   value_00;         /* READ ONLY  */
bit32e_type   value_01;         /* READ ONLY  */
bit32e_type   value_02;         /* READ ONLY  */
bit32e_type   value_03;         /* READ ONLY  */
bit32e_type   value_04;         /* READ ONLY  */
bit32e_type   value_05;         /* READ ONLY  */
bit32e_type   value_06;         /* READ ONLY  */
bit32e_type   value_07;         /* READ ONLY  */
bit32e_type   value_08;         /* READ ONLY  */
bit32e_type   value_09;         /* READ ONLY  */
bit32e_type   value_10;         /* READ ONLY  */
bit32e_type   value_11;         /* READ ONLY  */
bit32e_type   value_12;         /* READ ONLY  */
bit32e_type   value_13;         /* READ ONLY  */
bit32e_type   value_14;         /* READ ONLY  */
bit32e_type   value_15;         /* READ ONLY  */
bit32e_type   value_16;         /* READ ONLY  */
bit32e_type   value_17;         /* READ ONLY  */
```

Summary

If an error queue element is available on the free queue, the indicated message is formatted and copied into it, and the element is placed on the ready queue.

Parameters

priority — The priority of this error message. See **syslog.h** for priority definitions.

format — A **printf** format string that specifies the format to be used for the message.

value_00-17 — The parameters to be substituted into the **printf** string format.

Description

If the error daemon **syslogd** has not opened the **err** pseudodevice, the message is formatted and printed on the console. If an empty record is available, the priority number and the message are formatted into it. Long messages are truncated. The formatted record is placed on the ready queue, and the event counter is advanced. If there are no available records, the message is ignored.

Return Values

TRUE — If an error queue element was available.

FALSE — If no error queue element was available.

End of Chapter

Chapter 14
Select Manager Routines

This chapter is contains routines used by a select routine for a standard DG/UX driver and may not be appropriate for a STREAMS driver. Throughout the chapter we will refer to the driver's select routine as the **dev_xxx_select** routine.

This chapter describes the DG/UX kernel routines used for accessing device select lists. We start with a brief introduction to select operations and the select manager routines. Following the introduction is a "Constants and Data Structures" section, which lists some of the major constants and data structures used by routines in this section. Check the appropriate include files (for example, check **i_io.h** for structures beginning with the **io** acronym) for a complete and current list of all constants and structures.

Overview to Using the Select Manager Routines

The select manager facility consists of a set of utility routines that a driver may use to help implement its **dev_xxx_select** routine. Select operations allow a user to wait for multiple I/O requests from a device without directly suspending. The kernel's select routines help your driver manage select operations by maintaining lists of outstanding select operations for each device. Note that some devices (for example, disks) always return TRUE when selected because the operations complete quickly and cannot be interrupted. Drivers for such devices will not need the routines in this section because their select operations do not need to keep lists of waiting users.

The select manager keeps a list of the processes waiting for I/O events on a particular device. To use the select manager routines, during initialization you must allocate a data structure of type **io_select_list_type** for each physical device that may be selected. This structure is the head of the select-request list and holds information the select manager needs to handle the list. You must initialize each select list by calling **io_select_init** before the list is used.

When a user makes a select request, the kernel passes control to the driver's **dev_xxx_select** routine. If the select condition is satisfied, **dev_xxx_select** will return TRUE immediately. If the select condition is not satisfied, **dev_xxx_select** can place an entry on the device's select list by calling **io_select_register**. The entry records the intent of the select: read, write, or exception. It also contains a pointer to the eventcounter to be raised when the select condition is satisfied.

When an I/O event occurs on the device (for example, the driver receives data, learns the device is ready for writing, or discovers an exceptional condition on a device), the driver calls **io_select_satisfy** with the corresponding intent (read/write/exception) flagged. **Io_select_satisfy** traverses the device's select list to advance the eventcounters and hence wake up the processes interested in that event.

Note that **io_select_satisfy** leaves the entry on the select list whether it is satisfied or not. To remove the entry **dev_xxx_select** calls **io_select_cancel**. Until it is cancelled the entry's eventcounter will continue to advanced when the indicated select condition is satisfied. Every **io_select_register** must eventually be followed by a call to **io_select_cancel** so old entries are not left on the list.

CAUTION:

It is essential that you note the following items:

The select manager operations do NOT lock the select-request lists. The device driver must lock the list structure to ensure that accesses to the select list are single-threaded or ensure that at most one of the select manager functions is in progress on a particular select list at one time. If, for example, **io_select_register** is trying to add a new entry to the select list at the same time **io_select_satisfy** is trying to traverse the list, indeterminate results may occur, possibly even a kernel panic.

Io_select_satisfy will frequently be called from an interrupt service routine. If you call it from a service routine, be sure to mask out the device's interrupts before you call other routines with its select list. If you don't, the interrupt level may encounter a partially processed list.

The following routines are described in this section:

- **io_select_cancel**
- **io_select_init**
- **io_select_register**
- **io_select_satisfy**

Routines beginning with **io** require the **i_io.h** include file.

Constants and Data Structures

io_select_intent_type

```
typedef bit16_type  io_select_intent_type  ;

IO_SELECT_INTENT_READ
IO_SELECT_INTENT_WRITE
IO_SELECT_INTENT_EXCEPTION
IO_SELECT_INTENT_NONE
```

Description

This type describes the select options that may be specified to a device driver's **dev_xxx_select** routine. The READ, WRITE, and EXCEPTION options start a select for the corresponding operation. You can use any combination of these three options in a single **dev_xxx_select** call. IO_SELECT_INTENT_NONE is used as a return value from **io_select_cancel** when no intent has been satisfied.

io_select_cancel

Syntax

```
io_select_intent_type io_select_cancel (select_list_ptr,
                                        ec_ptr)

io_select_list_ptr_type  select_list_ptr;    /*WRITE ONLY*/
vp_ec_ptr_type           ec_ptr;             /*READ ONLY*/
```

Summary

This routine removes the process identified by **ec_ptr** from the select list.

Parameters

select_list_ptr — A pointer to a select list.

ec_ptr — A pointer to a process's select eventcounter.

Return Values

The type of select intent satisfied (or none).

io_select_init

Syntax

```
void     io_select_init (select_list_ptr)

io_select_list_ptr_type  select_list_ptr;   /*WRITE ONLY*/
```

Summary

This routine initializes the given select list.

Parameters

select_list_ptr — A pointer to a select list.

Return Values

None.

io_select_register

Syntax

```
void   io_select_register (select_list_ptr, intent, ec_ptr)

io_select_list_ptr_type  select_list_ptr;    /*READ/WRITE/*
io_select_intent_type    intent;             /*READ ONLY*/
vp_ec_ptr_type           ec_ptr;             /*READ ONLY*/
```

Summary

This routine registers a select with the given intent and eventcounter on the given select list.

Parameters

select_list_ptr — A pointer to a select list.

intent — The intent of the select.

ec_ptr — A pointer to the select eventcounter of the selecting process.

Description

See the "Constants and Data Structures" section for a list of defines for **intent.**

Return Values

None.

io_select_satisfy

Syntax

```
void   io_select_satisfy (select_list_ptr, intent)

io_select_list_ptr_type  select_list_ptr;  /*WRITE ONLY*/
io_select_intent_type    intent;           /*READ ONLY*/
```

Summary

This routine searches the given select list for processes interested in the given I/O event. The select eventcounters for those processes are advanced.

Parameters

select_list_ptr — A pointer to a select list.

intent — The type of select to satisfy.

Return Values

None.

End of Chapter

Chapter 15
Nodevice Routine Stubs

This chapter lists pre-written stub routines that drivers may use for required driver interface routines. Instead of writing your own routine, you can use one of the routines listed anytime your driver does not process the I/O operation indicated in the routine's name. For example, if your device cannot be used as a dump device, you can use the **io_nodevice_open_dump** routine instead of supplying your own open dump stub. To use the nodevice stub, you simply supply the nodevice routine's name in the driver's routines vector structure.

The nodevice routines support both block and character operations so they can serve as stubs for both types of requests. The routines in this section generally return at least an error and, in some cases, a panic. Before you use one of these routines, make sure its error return is acceptable and appropriate for your device.

The following routines are described in this section:

- **io_nodevice_open**
- **io_nodevice_close**
- **io_nodevice_read_write**
- **io_nodevice_select**
- **io_nodevice_ioctl**
- **io_nodevice_start_io**
- **io_nodevice_configure**
- **io_nodevice_deconfigure**
- **io_nodevice_name_to_device**
- **io_nodevice_device_to_name**
- **io_nodevice_open_dump**
- **io_nodevice_write_dump**
- **io_nodevice_read_dump**
- **io_nodevice_close_dump**

- **io_nodevice_powerfail**
- **io_nodevice_mmap**
- **io_nodevice_munmap**
- **io_nodevice_maddmap**
- **io_nodevice_service_interrupt**

Constants and Data Structures

There are no special constants or data structures required for these routines.

io_nodevice_open

Syntax

```
status_type io_nodevice_open (device_number, channel_flags,
                              device_handle_ptr)

io_device_number_type      device_number;     /*READ ONLY*/
io_channel_flags_type      channel_flags;     /*READ ONLY*/
io_device_handle_ptr_type  device_handle_ptr; /*WRITE ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to open a non-existent device.

Parameters

device_number — The major and minor device number from the special file that is being opened.

channel_flags — A set of flags specifying whether the I/O to the device will be reads, writes, or both.

device_handle_ptr — A pointer to the location where the device handle is to be returned. Because **io_nodevice_open** always fails, no device handle is ever returned.

Description

This routine returns a status indicating that the device does not exist.

Return Values

IO_ENXIO_DEVICE_DOES_NOT_EXIST — This value is always returned.

Exceptions

None.

io_nodevice_close

Syntax

```
status_type io_nodevice_close (device_handle, channel_flags)

io_device_handle_type  device_handle;  /*READ ONLY*/
io_channel_flags_type  channel_flags;  /*READ ONLY*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

device_handle — The device handle for the device that is being closed.

channel_flags — The flags with which the device was opened.

Description

Panic is invoked.

Return Values

None.

Exceptions

None.

Abort Conditions

This routine always panics with the following panic code:

IO_PANIC_NODEVICE_CLOSE — An attempt was made to close a major device number for which no driver exists.

io_nodevice_read_write

Syntax

```
status_type     io_nodevice_read_write (request_info_ptr)

io_request_info_ptr_type  request_info_ptr;  /*READ ONLY*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

request_info_ptr — A pointer to a packet containing the information necessary to specify a read or write request.

Description

Panic is invoked.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic is always called with the following panic code:

IO_PANIC_NODEVICE_READ_WRITE — An attempt was made to do a read or write operation on a major device number for which no driver exists.

io_nodevice_select

Syntax

```
void  io_nodevice_select (device_handle, select,
                          ec_ptr, intent_ptr)

io_device_handle_type      device_handle;  /*READ ONLY*/
boolean_type               select;         /*READ ONLY*/
vp_ec_ptr_type             ec_ptr;         /*READ ONLY*/
io_select_intent_ptr_type  intent_ptr;     /*READ WRITE*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

device_handle — The device handle of the device that is the target of select. This handle must be a device handle that was returned by the open routine of this driver.

select — If TRUE, this is the start of a select operation; conditions that are not immediately TRUE should be recorded so that the eventcounter can be advanced when they become TRUE. If FALSE, this is the end of a select operation; any previously remembered conditions should be forgotten.

ec_ptr — Specifies the eventcounter to be advanced by the driver when the select is satisfied if it is not immediately satisfied.

intent_ptr — On input, **intent_ptr** specifies whether a select is to be instituted for a combination of read, write, or exceptional conditions.

Description

Panic is invoked.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic is always invoked with the following panic code:

IO_PANIC_NODEVICE_SELECT — An attempt was made to do a select operation on a device for which no driver exists.

io_nodevice_ioctl

Syntax

```
status_type io_nodevice_ioctl  (device_handle, command,
                                 parameter, return_value_ptr)

io_device_handle_type     device_handle;  /*READ ONLY*/
bit32e_type               command;        /*READ ONLY*/
bit32e_type               parameter;      /*READ/WRITE*/
int32e_ptr_type           parameter;      /*WRITE ONLY*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

device_handle — The device handle of the device that is the target of the I/O control operation.

command — A command to the device. The interpretation of the command is specific to the driver.

parameter — An argument to the command. The interpretation of the parameter is specific to the driver and the command. The parameter may be used to transfer information in either direction between the caller and the device. In particular, it may be a pointer to a buffer supplied by the caller.

return_value_ptr — A pointer to the value to be returned to the user.

Description

This routine causes a system panic.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic is always invoked with the following panic code:

IO_PANIC_NODEVICE_IOCTL — An attempt was made to do an ioctl operation on a device for which no driver exists.

io_nodevice_start_io

Syntax

```
status_type  io_nodevice_start_io (op_record_ptr)

io_operation_record_ptr_type  op_record_ptr;  /*READ ONLY*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

op_record_ptr — A pointer to the operation record for the asynchronous request. The operation record contains fields indicating the minor device that is the target of the operation, the operation to be performed, the offset on the device from which the operation is to commence, the size of the transfer, the address of the main memory buffer, and the address of the routine that is to be called when the operation completes.

Description

This routine causes a system panic.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic is always invoked with the following panic code:

IO_PANIC_NODEVICE_START_IO — An attempt was made to do an start_io operation on a device for which no driver exists.

io_nodevice_configure

Syntax

```
status_type io_nodevice_configure (device_name_ptr,
                                   major_number)

char_ptr_type                 device_name_ptr;/*READ ONLY*/
io_major_device_number_type   major_number;   /*READ ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to configure a non-existent device.

Parameters

device_name_ptr — A pointer to the character string name of the device to be configured.

major_number — The major device number on which the device is to be configured.

Description

This routine always returns an error.

Return Values

IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED — This status is always returned.

Exceptions

None.

Abort Conditions

None.

io_nodevice_deconfigure

Syntax

```
status_type     io_nodevice_deconfigure (device_name_ptr)

char_ptr_type   device_name_ptr;  /*READ ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to deconfigure a non-existent device.

Parameters

device_name_ptr — A pointer to the null-terminated string specifying the device to be deconfigured.

Description

This routine always returns an error.

Return Values

IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED — This status is always returned.

Exceptions

None.

io_nodevice_name_to_device

Syntax

```
status_type io_nodevice_name_to_device (device_name_ptr,
                                        number_ptr)

char_ptr_type               device_name_ptr;/*READ ONLY*/
io_device_number_ptr_type   number_ptr;     /*WRITE ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to do name-to-device conversion on a non-existent device.

Parameters

device_name_ptr — A pointer to the null-terminated device name that is to be translated.

number_ptr — A pointer to where the corresponding device number is to be written.

Description

This routine always returns an error.

Return Values

IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED — This status is always returned.

Exceptions

None.

io_nodevice_device_to_name

Syntax

```
status_type io_nodevice_device_to_name  (device_number,
                                          name_ptr, size)

io_device_number_type    device_number;  /*READ ONLY*/
char_ptr_type            name_ptr;       /*WRITE ONLY*/
uint32_type              size;           /*READ ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to do device-to-name conversion on a non-existent device.

Parameters

device_number — The device number to be translated into a device name character string.

name_ptr — A pointer to where the null-terminated character string name is to be written.

size — The maximum number of bytes, including the terminating null, that is to be written to **name_ptr**.

Description

This routine always returns an error.

Return Values

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — This status is always returned.

Exceptions

None.

io_nodevice_open_dump

Syntax

```
status_type     io_nodevice_open_dump (device_name)

char_ptr_type   device_name;  /*READ ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to dump to a non-existent device or to a device that does not support dumps.

Parameters

device_name — The character string name of the device to which the dump is being written.

Description

This routine always returns an error.

Return Values

IO_STATUS_DUMP_NOT_SUPPORTED — This status indicates that the device does not support dumps. This status is always returned.

Exceptions

None.

Abort Conditions

None. This routine must not panic because it is invoked as part of the panic sequence.

io_nodevice_write_dump

Syntax

```
status_type io_nodevice_write_dump (buffer_ptr, buffer_size)

pointer_to_any_type  buffer_ptr;   /*READ ONLY*/
uint32_type          buffer_size;  /*READ ONLY*/
```

Summary

This routine is a stub for handling devices that do not exist. It is a system fatal error to call this routine.

Parameters

buffer_ptr — A pointer to the buffer of data to be written to the system dump.

buffer_size — The size, in bytes, of the buffer.

Description

This routine should never be called because **io_nodevice_open_dump** always fails.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_NODEVICE_WRITE_DUMP — An attempt was made to write dump information to a non-existent device.

io_nodevice_read_dump

Syntax

```
status_type io_nodevice_read_dump  (buffer_ptr, buffer_size)

pointer_to_any_type   buffer_ptr;     /* WRITE ONLY  */
uint32_type           buffer_size;    /* READ  ONLY  */
```

Summary

This function is a stub for handling devices that do not exist. It is a system fatal error to call this function.

Parameters

buffer_ptr — A pointer to the buffer to which data is to be read.

buffer_size — The size, in bytes, of the buffer.

Description

This function should never be called because **io_nodevice_open_dump** always fails.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error codes:

IO_PANIC_NODEVICE_READ_DUMP — An attempt was made to read dump information from a non-existent device.

io_nodevice_close_dump

Syntax

```
status_type io_nodevice_close_dump ()
```

Summary

This routine is a stub for handling devices that do not exist. It is a system fatal error to call this routine.

Parameters

None.

Description

This routine should never be called because **io_nodevice_open_dump** always fails.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_NODEVICE_CLOSE_DUMP — An attempt was made to close a non-existent dump device.

io_nodevice_powerfail

Syntax

```
status_type     io_nodevice_powerfail  ()
```

Summary

This routine is a stub routine that simply returns OK, because there is nothing to do in order to perform powerfail restart on nodevice.

Parameters

None.

Description

The status OK is returned.

Return Values

OK — This value is always returned.

Exceptions

None.

Abort Conditions

None.

io_nodevice_mmap

Syntax

```
status_type    io_nodevice_mmap    ()
```

Summary

This routine is a stub for handling the **mmap** system call. The **errno** EINVAL is returned.

Parameter

None.

Description

This routine always returns an error.

Return Values

IO_EINVAL_MMAP_NOT_SUPPORTED — The **mmap** operation is not supported for this device.

Exceptions

None.

io_nodevice_munmap

Syntax

```
status_type    io_nodevice_munmap    ()
```

Summary

This routine is a stub for handling the **munmap** system call. The **errno** EINVAL is returned.

Parameters

None.

Description

This routine always returns an error.

Return Values

IO_EINVAL_MUNMAP_NOT_SUPPORTED — The **munmap** operation is not supported for this device.

Exceptions

None.

io_nodevice_maddmap

Syntax

```
status_type    io_nodevice_maddmap    ()
```

Summary

This function is a stub for incrementing reference counts to memory mapped sections. The **errno** EINVAL is returned.

Parameters

None.

Description

EINVAL is returned.

Return Values

IO_EINVAL_MMAP_NOT_SUPPORTED — The **maddmap** operation is not supported for this device.

Exceptions

None.

io_nodevice_service_interrupt

Syntax

```
void  io_nodevice_service_interrupt  (device_code, device_class)

io_device_code_type       device_code;  /*READ ONLY*/
uc_device_class_enum_type device_class;  /*READ ONLY*/
```

Summary

This routine handles unexpected interrupts from devices that are not configured into the kernel.

Parameters

device_code — The device code of the interrupting device.

device_class — The device class of the interrupting device.

Description

This routine runs at interrupt level. It handles interrupts from devices that are not configured and, therefore, which should not be generating interrupts. This routine does not obey the standard interface for service interrupt routines. Because it must service interrupts from all devices, it uses a device code as the argument instead of a device information structure pointer.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_NODEVICE_INTERRUPT_OVERRUN — Too many unexpected interrupts were received in too short a time. This panic probably indicates the existence of a hardware problem that is generating spurious interrupts.

End of Chapter

Chapter 16
Miscellaneous Routines

This chapter describes the DG/UX kernel routines used for a variety of driver operations including: checking self-IDs; converting hexadecimal strings to their integer values; formatting a print line; creating a system panic; and determining whether the calling process has super-user privileges.

The routines described in this section are:

- **fs_check_self_id**
- **io_hex_str_to_int**
- **misc_format_line**
- **pm_is_super_user**
- **sc_panic**

Routines beginning with fs, misc, pm, sc, and io require the **i_fs.h**, **i_misc.h**, **i_pm.h**, **i_sc.h**, and **i_io.h** include files, respectively.

Constants and Data Structures

There are no special constants or data structures required for these routines.

fs_check_self_id

Syntax

```
boolean_type      fs_check_self_id  (blocks_ptr, self_id_ptr,
                                     count_ptr)

pointer_to_any_type   blocks_ptr;      /*READ ONLY*/
df_self_id_ptr_type   self_id_ptr;     /*READ ONLY*/
uint32_ptr_type       count_ptr;       /*READ/WRITE*/
```

Summary

This routine checks the self-ID for the given set of blocks.

Parameters

blocks_ptr — Pointer to the beginning of the first block to be checked.

self_id_ptr — Pointer to the self-ID that the first block is expected to have.

count_ptr — On input, the number of bytes to be checked. On output, the number of bytes that checked out OK. On both input and output, **count_ptr** must be a multiple of the block size, though this is not checked.

Description

A self-ID is an identifying number used to identify different non-data disk blocks used in disk administration (for example, header blocks). For each block, this routine checks its self-ID against the prototype self-ID. If any block fails, FALSE is returned along with the number of bytes that passed the check. If all blocks pass, then TRUE is returned along with the number of bytes that were checked.

Return Value

TRUE — All blocks were successfully checked.

FALSE — At least one block failed a self-ID check.

io_he:_str_to_int

Synta:

```
boolean_type      io_hex_str_to_int   (str_ptr, int_value_ptr)
cha:_ptr_type     str_ptr;            /*READ  ONLY*/
uint32_ptr_type   int_value_ptr;      /*WRITE ONLY*/
```

Summa:y

Ret:rn the integer value of the null terminated hexadecimal string at **str_ptr**.

Parame ers

str_ ptr — A pointer to the beginning of the string to convert.

int_ alue_ptr — Pointer to location where the integer value is to be returned.

De :ription

Sc . a string **str_ptr** consisting only of the characters '0' - '9', 'a' - 'f', and 'A' - 'F' and terminated with a null character, returning its unsigned 32-bit value at int value_ptr. If any other characters are encountered or the value exceeds what ca: be expressed in a 32-bit unsigned value then **int_value_ptr** is unchanged and an :rror is returned.

Retur: Values

F: _SE — Successful conversion occurred.

T: UE — The string conversion failed.

misc_format_line

Syntax

```
uint32_type misc_format_line
(result_buf, rb_size, format, value_00, value_01, value_02
                              value_03, value_04, value_05
                              value_06, value_07, value_08
                              value_09, value_10, value_11
                              value_12, value_13, value_14
                              value_15, value_16, value_17
                              value_18, value_19)

char_ptr_type  result_buf;    /*WRITE ONLY*/
uint32_type    rb_size;       /*READ ONLY*/
char_ptr_type  format;        /*READ ONLY*/
bit32e_type    value_00;      /*READ ONLY*/
bit32e_type    value_01;      /*READ ONLY*/
bit32e_type    value_02;      /*READ ONLY*/
bit32e_type    value_03;      /*READ ONLY*/
bit32e_type    value_04;      /*READ ONLY*/
bit32e_type    value_05;      /*READ ONLY*/
bit32e_type    value_06;      /*READ ONLY*/
bit32e_type    value_07;      /*READ ONLY*/
bit32e_type    value_08;      /*READ ONLY*/
bit32e_type    value_09;      /*READ ONLY*/
bit32e_type    value_10;      /*READ ONLY*/
bit32e_type    value_11;      /*READ ONLY*/
bit32e_type    value_12;      /*READ ONLY*/
bit32e_type    value_13;      /*READ ONLY*/
bit32e_type    value_14;      /*READ ONLY*/
bit32e_type    value_15;      /*READ ONLY*/
bit32e_type    value_16;      /*READ ONLY*/
bit32e_type    value_17;      /*READ ONLY*/
bit32e_type    value_18;      /*READ ONLY*/
bit32e_type    value_19;      /*READ ONLY*/
```

Summary

This routine provides limited **sprintf**(3) functionality; it formats output and performs value substitutions. It formats a line, creating a string by substituting values according to field descriptors. The field descriptors in the input format are a subset of the field descriptors that the standard library routine **printf** provides. Currently provided are **%c, %s, %d, %o, %x, %u** and the ability to specify field length and zero padding.

Parameters

result_buf — Resulting formatted output placed here.

rb_size — Size of the buffer.

format — The format string. This format string is the same as those used in **printf**.

value_00...value_19 — Place holder for 0 to 19 format substitution values. Use **value_00** for the first value, **value_01** for the second value, etc.

Return Values

None.

Exceptions

None.

pm_is_super_user

Syntax

```
boolean_type pm_is_super_user ()
```

Summary

This routine determines whether the calling process has superuser permission. If so, it notifies the kernel that the process has used superuser permission so it can be recorded for accounting information.

Parameters

None.

Return Values

TRUE — Caller is superuser.

FALSE — Caller is not superuser.

Exceptions

None.

Abort Conditions

None.

sc_panic

Syntax

```
void  sc_panic     (panic_code)

sc_panic_code_type  panic_code;     /*READ ONLY*/
```

Summary

This routine is the panic routine that you call when serious errors or inconsistencies are detected. A panic message is written to the system console, and the emergency shutdown sequence is entered.

Parameters

panic_code — A value identifying the cause of the panic. This value will be written to the system console along with the panic message. Non-standard devices should use a panic code between 0 and 511 decimal to avoid collision with existing system panic codes.

Description

The panic lock is obtained to ensure that only one processor enters the panic and emergency shutdown code. If any other processors are running, they are stopped. The routine **sc_write_line** is called to write the panic message to the system console and the emergency shutdown routine is entered.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

End of Chapter

Appendix A
Defining Device Specification Parameters

Appendix B of *Installing the DG/UX™ System* describes device specifications and lists the default memory-mapped I/O addresses, interrupt levels, and interrupt vectors for Data General-supplied devices. It also lists default values for SCSI IDs. This appendix describes the conventions for selecting default values for these variables for a new device.

Additional details on specific peripherals and their drivers are given in the man pages for the device listed under the device's mnemonic. For example, the Ciprico ESDI disk (mnemonic **cied**) is described in **cied**(7).

Peripheral defaults are specific to the interrupt class (see Chapter 1). So this appendix breaks down default tables by interrupt class: VMEbus and Integrated.

VMEbus Class I/O Defaults

On AViiON systems, the VMEbus control area (memory-mapped address area) is grouped into subareas based on the VME data width to be used by the device, for example, *a16* or *a32*. Figure A-1 shows the VMEbus control area. Addresses of the *a16* and *a32* data width areas are fixed by the kernel.

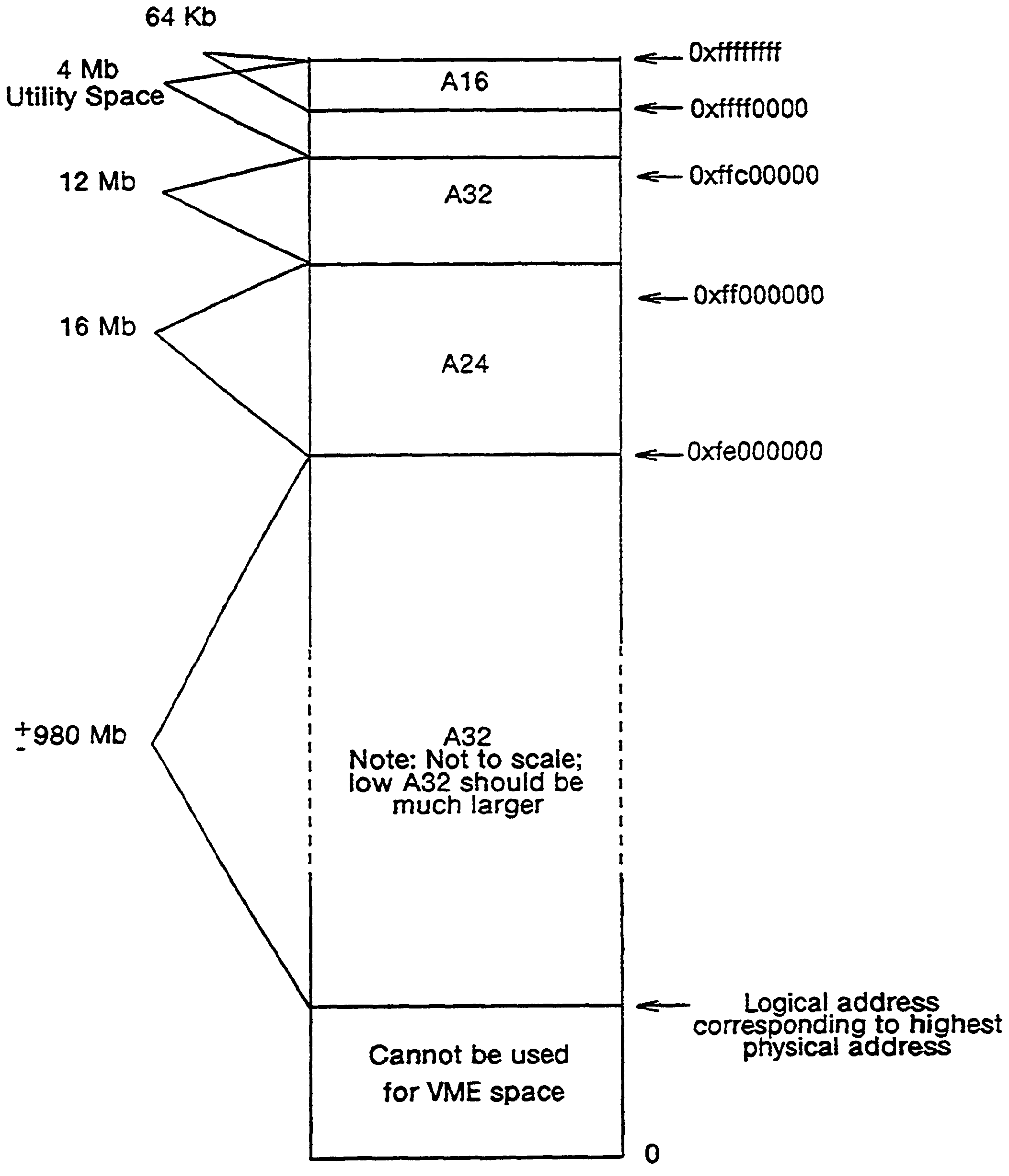

64 Kb
4 Mb
Utility Space
12 Mb
16 Mb
±980 Mb
A16
A32
A24
A32
Note: Not to scale; low A32 should be much larger
Cannot be used for VME space
0xffffffff
0xffff0000
0xffc00000
0xff000000
0xfe000000
Logical address corresponding to highest physical address
0

Figure A-1 *VMEbus Memory-Mapped I/O Addresses and Data Width Areas*

Table A-1 shows the device mnemonics for various standard VMEbus devices with their default memory-mapped I/O address, interrupt level, and interrupt vector. The "A16 Address" and "A32 Address" columns in Table A-1 indicate whether the base address is in *a16* or *a32* space. Some devices, such as the Hawk LAN, require two I/O address areas, for example, one in A16 memory and one in A32 memory. The "Length in Bytes" column gives the length of the device's memory-mapped area. The base address plus the number of bytes defines the memory area reserved for the device.

NOTE: **cimd** (SMD disk) and **cied** (ESDI disk) devices share the same default interrupt vectors and base addresses. The **cird** driver handles both SMD or ESDI disks and thus it too shares these defaults. Thus, the default values listed for **cird** apply to the **cimd** and **cied** drivers as well. If you have both SMD and ESDI devices, use the **cird** mnemonic and treat the two types of disks as instances of a **cird** device.

Table A-1 VMEbus I/O Addresses and Interrupt Level/Vector Defaults

Device	A16 Address	A32 Address	Interrupt Level/Vector	Length
Ciprico Controller (cied, cimd, cird)				
cird(0)	0xFFFFEF00		2/0x18	512 bytes
cird(1)	0xFFFFF100		2/0x19	
cird(2)	0xFFFFFB00		2/0x1A	
cird(3)	0xFFFFFD00		2/0x1B	
Ciprico SCSI Adapter (cisc)				
cisc(0)	0xFFFFF300		2/0x28	512 bytes
cisc(1)	0xFFFFF500		2/0x29	
cisc(2)	0xFFFFF700		2/0x2A	
cisc(3)	0xFFFFF900		2/0x2B	
cisc(4)	0xFFFFED00		2/0x2C	
cisc(5)	0xFFFFD700		2/0x2D	
cisc(6)	0xFFFFD900		2/0x2E	
cisc(7)	0xFFFFDB00		2/0x2F	
cisc(8)	0xFFFFDD00		2/0x20	
cisc(9)	0xFFFFDF00		2/0x21	
cisc(A)	0xFFFFE100		2/0x22	
cisc(B)	0xFFFFE300		2/0x23	
cisc(C)	0xFFFFE500		2/0x24	
cisc(D)	0xFFFFE700		2/0x25	
cisc(E)	0xFFFFE900		2/0x26	
cisc(F)	0xFFFFEB00		2/0x27	

Device	A16 Address	A32 Address	Interrupt Level/Vector	Length
High-Availability Disk-Array Adapter (hada)				
hada(0)	0xFFFF1000		2/0x70	1024
hada(1)	0xFFFF1400		2/0x71	
hada(2)	0xFFFF1800		2/0x72	
hada(3)	0xFFFF1C00		2/0x73	
Interphase VME Ethernet Controller (hken)				
hken(0)	0xFFFF4000	0x55900000	3/0x15	4Kbytes
hken(1)	0xFFFF5000	0x55980000	3/0x16	in A16
hken(2)	0xFFFF4200	0xE1100000	3/0x10	area
hken(3)	0xFFFF4400	0xE1180000	3/0x11	
hken(4)	0xFFFF4600	0xE1200000	3/0x12	512 bytes
hken(5)	0xFFFF4800	0xE1280000	3/0x13	in A32
hken(6)	0xFFFF4A00	0xE1300000	3/0x14	area
hken(7)	0xFFFF4C00	0xE1380000	3/0x17	
VME VSC Synchronous Controller (ssid)				
ssid(0)		0x55B00000	3/0x50	4Kbytes
ssid(1)		0x55B10000	3/0x51	
ssid(2)		0x55B20000	3/0x52	
ssid(3)		0x55B30000	3/0x53	
ssid(4)		0x55B40000	3/0x54	
ssid(5)		0x55B50000	3/0x55	
ssid(6)		0x55B60000	3/0x56	
ssid(7)		0x55B70000	3/0x57	
ssid(8)		0xE2080000	3/0x58	
ssid(9)		0xE2090000	3/0x59	
ssid(A)		0xE20A0000	3/0x5A	
ssid(B)		0xE20B0000	3/0x5B	
ssid(C)		0xE20C0000	3/0x5C	
ssid(D)		0xE20D0000	3/0x5D	
ssid(E)		0xE20E0000	3/0x5E	
ssid(F)		0xE20F0000	3/0x5F	

Device	A16 Address	A32 Address	Interrupt Level/Vector	Length
Systech Asynchronous Terminal Controller (syac)				
syac(0)		0x60000000	4/0x60	128Kbytes
syac(1)		0x60020000	4/0x61	
syac(2)		0x60040000	4/0x62	
syac(3)		0x60060000	4/0x63	
syac(4)		0x60080000	4/0x64	
syac(5)		0xE30A0000	4/0x65	
syac(6)		0xE30C0000	4/0x66	
syac(7)		0xE30E0000	4/0x67	
syac(8)		0xE3100000	4/0x68	
syac(9)		0xE3120000	4/0x69	
syac(A)		0xE3140000	4/0x6A	
syac(B)		0xE3160000	4/0x6B	
syac(C)		0xE3180000	4/0x6C	
syac(D)		0xE31A0000	4/0x6D	
syac(E)		0xE31C0000	4/0x6E	
syac(F)		0xE31E0000	4/0x6F	
VME Token Ring Controller (vitr)				
vitr(0)		0x61000000	3/0x40	512 bytes
vitr(1)		0x61002000	3/0x41	
vitr(2)		0xE4004000	3/0x42	
vitr(3)		0xE4006000	3/0x43	
vitr(4)		0xE4008000	3/0x44	
vitr(5)		0xE400A000	3/0x45	
vitr(6)		0xE400C000	3/0x46	
vitr(7)		0xE400E000	3/0x47	

Conventions for Selecting VMEbus I/O Addresses

The following conventions and restrictions apply to selecting your VMEbus memory-mapped I/O address, interrupt level, and interrupt vector:

- To select your base memory-mapped I/O address, you simply find an unreserved area of memory in the correct data width area. We recommend that you use the highest data width area possible to maximize device speed. Thus, use Extended Addressing (a32) mode if possible. Use Short Addressing (a16) mode only if your device does not support any higher data width.

NOTE: The a32 logical address space is used by the DG/UX kernel. Therefore, if you are using a memory mapped address in a32, you must map the a32 physical address you want to another logical address. You do this using **vm_get_unwired_memory** to get a logical address and then using **vm_map_physical_memory** to map this address to the desired physical address.

- In setting your VME address modifiers, always use the Supervisory mode.
- On DG/UX systems, the standard interrupt levels for different devices are as follows:

```
2  for disks
3  for networks
4  for terminal controllers
2  for SCSI adapters
```

We recommend that you follow these defaults if you have one of the devices listed above. If you have a non-standard device, you may choose whichever interrupt level you want. Bear in mind that when you mask your device, you will be masking all others using the same interrupt level.

- The VME vector number uniquely identifies a controller or adapter for the whole VME bus. As with I/O addresses, vector numbers for Data General-supplied devices are pre-assigned and usually come correctly jumpered from the factory or are set by driver software. Unlike memory-mapped I/O addresses, the vector numbers of some devices can be set by the device driver at configuration time.

  On the AViiON system, you can select VME vector numbers from 0 through 255. To get a vector number for your device, simply refer to Table A-1 and select an unused vector number less than 255.

Integrated Class I/O Defaults

Base addresses and device codes for Integrated Class devices are specific to the AViiON machine on which they are defined. Check the CPU manual for your particular machine for that machine's integrated device addresses. The *Guide to AViiON® and DG/UX™ System Documentation* lists the hardware manuals for all AViiON machines (see also the on-line version, **/usr/release/doc_guide**). In your code, use the Integrated class **defines** to specify your devices' device code. *Installing the DG/UX™ System* also lists the device codes for Data General-supplied drivers.

Table A-2 shows the device mnemonics and default memory-mapped I/O address for an AViiON 300 series machine. The length of the address space is given in parentheses following the base address.

Table A-2 AViiON Station I/O Address Defaults

Mnemonic	Base Address (bytes)	Description
-	N/A	Power Fail
-	N/A	Parity Error
-	N/A	Z8536 C10 Interrupt
kbd	0xfff82800 (1K)	Keyboard
duart(0)	0xfff82000 (255)	DUART
duart(1)	0xfff82C00 (255)	DUART
lp	0xfff82400 (1K)	Parallel Port
inen	0xfff8c000 (4K)	Ethernet Controller
insc	0xfff8a000 (4K)	SCSI Controller
ncsc(0)	0xfffb0000 (64)	NCR53C700 SCSI Interface
ncsc(1)	0xfffb0080 (64)	NCR53C700 SCSI Interface
dgen(0)	0xfffb0100 (64)	2nd Gen. Ethernet Cntrller
dgen(1)	0xfffb0140 (64)	2nd Gen. Ethernet Cntrller
-	N/A	DMA Terminal Count Reached
-	N/A	DMA Write Protect Error
-	N/A	DMA Valid Bit
grf	0xfff89000 (4K)	Graphics
-	N/A	Software Interrupt

The following conventions and restrictions apply to selecting your memory-mapped I/O address.

- The parentheses following each base address show the number of bytes that the device uses. The base address plus the number of bytes equals the entire memory area reserved for the device.

Disk and Tape Command Set Compatibility

In order for a tape device to be compatible with the DG/UX SCSI tape driver (**st**), it must conform to the ANSI SCSI-1 Command Set specification (X3.131- 1986). In addition, it must also support the following commands listed as optional in the ANSI specification

- **Inquiry Command**
- **Space Command**
- **Test Unit Ready Command**
- **Mode Sense/Select Commands**

In order for a disk device to be compatible with the DG/UX SCSI disk driver (**sd**), it must conform to the ANSI SCSI-1 Command Set specification (X3.131- 1986). In addition, it must also support the following commands listed as optional in the ANSI specification

- **Inquiry Command**
- **Mode Sense/Select Commands**
- **Prevent/Allow Medium Removal Commands**
- **Read Capacity Command**
- **Test Unit Ready Command**

End of Appendix

Appendix B Preparing Master File and System File Entries

Master files are administrative files that contain the base information about a class of devices. All master files for all products are kept in the **master.d** directory along with the main DG/UX master file, **/usr/etc/master.d/dgux**. These files and their entries are discussed in the **master**(4) man page. While we discuss master file entries below, you should check this man page for the latest details on entry format as they may be subject to change.

If you are writing a standard device driver, a STREAMS driver or module, or a communications protocol driver, you will need to make a master file entry to enter information defining the kind of device (or operation) your driver supports. You will need to create a master file with your driver's master file entry and store it in the **master.d** directory.

NOTE: All files listed in the **master.d** directory are included in the configuration process. Therefore, do not keep old or backup copies of your master file in **master.d** as this will cause duplicate entry names. The master file has three kinds of sections to which you may want to add entries. They are:

- *Device Section*: holds descriptions of device drivers whether STREAMS or standard DG/UX drivers. You must make an entry for you driver in this section. There is also a *Protocol section* that holds descriptions of protocols supported by the socket (2) system call and *STREAMS Module section* that holds descriptions of the STREAMS modules. You do not make entries in these sections except perhaps if you have a STREAMS module as well as a STREAMS driver.

- *Keywords section*: defines and sets all configurable parameters. Keyword entries are optional.

- *Alias section*: allows you to define aliases for master file device entries. Alias entries are optional.

We discuss these entries in the next section.

Describing Your Device: The Device Section Entry

You will probably want to list one of the master files in **master.d** to see the layout of the different sections. We provide a sample entry below for discussion purposes. Lines that start with # are comments.

```
#--------------------------------------------------------
# Disks:
#  Name              Restriction    Concurrency
#  Prefix            Flags          Set
#  ----------        --------       ------------
   sd                n              default
   cird              n              default
#
   xdev              n              default
#
#--------------------------------------------------------
```

Device description entries have three fields: device name, restriction flags, and Concurrency. The device name is the driver's mnemonic as described in Chapter 2. It is the common connection between the system file entry, the master file entry and the driver's code — so be sure it's correct. The **xdev** entry above is a non-standard device we have added to the master file. We'll use this entry as an example to describe the fields in the device section. Entry information is case sensitive. The three **xdev** values are:

```
xdev      n      default
```

xdev
: **entry name** — This field identifies a family of devices, specifically, all devices that use the same device driver. The entry name or name prefix is a two- to eight-letter device mnemonic. It is also used as part of the corresponding device driver's name, in the device specification (the device mnemonic field) and in corresponding system file entries. The device mnemonic uses any characters that are valid for C language filenames.

n
: **restrictions flag** — This flag signals configuration restrictions for this device. The flags are specified as a string of characters with the following definitions (these options are case sensitive). The master file man page lists the current options available but a sample set is listed in Table B-1.

default
: **STREAMS Concurrency Set** — This field defines a set of STREAMS processes that may *not* run concurrently (that is they are mutually exclusive). Conversely, drivers that belong to separate sets may run concurrently. The name given in the field specifies the concurrency set to which the driver will belong. The concurrency set name has no meaning for non-STREAMS drivers which by convention are assigned the set named **default**. A set may contain drivers, modules or both. Two special sets exist: **module** and **streams**. Drivers in the **module** set have sets defined on a per-board basis. For example, syac code can run concurrently on **syac(0)** and **syac(1)** but different lines (different minor numbers) on the same board will run exclusively. Drivers in the **streams** set have sets defined on a per-stream basis. For example, **syac** code for all lines and boards will run exclusive of each other. All other name values specify a named set.

Table B-1 Restriction Flags

Option	Meaning
o	Specifies that the driver will allow only one device of this type to be configured. For example, the system console is defined as being the only device of its type.
r	Indicates that the device is required and will be placed in the system whether or not the system file specifies it. If the device is not specified, default values will be given for device specification values.
s	This option indicates that the device is a DG/UX-style STREAMS device.
S	This option indicates that the device is a System V-style STREAMS device.
N	This STREAMS device uses the new (System V.4) style open/close interface.
z	This device may be configured either explicitly or implicitly as part of nested declaration of another device. For example, "st(insc()),4)" declares "insc()" implicitly.
n	No restrictions apply. Choose this option if you do not use any of the others listed above.

Parameters: The Keyword Section Entry

If you want to create a parameter for your driver code that can be set at system configuration time, you can add a keyword entry to both the master file and system file. For example, the pseudoterminal driver has a variable giving the number of pseudoterminals to be configured. Most device drivers will not use the keyword section.

The master file entry for a parameter should be placed in the keyword section. This entry has four fields:

- **The variable name.** The variable name is used in the corresponding system file entry.

- **The default value for this variable.** This value is used if you do not add a corresponding system file entry to declare the variable's actual value.
- **The variable's data type.** If you don't specify this field, the kernel uses long integer for the data type.
- **The implied value.** This value is used if you add a system file entry but do not give that entry a value. This field is optional and exists primarily to give configuration flexibility for certain special devices such as the Network Filesystem (ONC™/NFS®).

Some sample keyword section entries are shown below:

```
#
# Variable                      Default                   Implied
# Name                          Value       Type          Value
# --------                      -------     --------      ------
#
cf_sc_nodename[]                "no_node"   char
cf_sci_daylight_savings_kind     1          uint16_type
maxup                            50         uint16_type
```

You reference your variable as an external variable by inserting a line similar to the following in your device driver:

```
extern uint16_type maxup;
```

During kernel configuration, all parameters listed in the master files will automatically be assigned their default values, unless explicitly overridden in a system file entry. To override a parameter's default value, you must add an entry to the system file's tunable parameters section with the parameter name and desired value (see the **system**(4) man page). For example, to change the maximum number of user processes (**maxup** master file entry), add the the following system file entry:

```
maxup       150
```

At configuration time, the **config** program combines the master and system entries to produce the file **conf.c**. As a result of the system file entry shown above, **conf.c** will contain a constant **maxup** with an updated value of 150. After configuration, you can check **conf.c** to see if your variable has been properly set (see Appendix C).

Master File Aliases: The Alias Section

The Alias section of the master file allows you to create aliases for your master file entry name. For example, the Ciprico ESDI (**cied**) and Ciprico SMD (**cimd**) disk controllers both use the same device driver, **cird**. Defining an alias for them allows you to access the driver while still using a mnemonic that clearly identifies the controller in the system file entry. The aliases might be as follows:

```
Alias  Alias
Name   Value
-----  ----------
cied   cird
cimd   cird
```

In the system file, an SMD controller can be specified as follows:

```
cimd()
```

Adding a System File Entry

Where the master file describes the class of devices, the system file describes specific information about a particular device. The system administrator must add an entry to the system file for each device to be configured.

The system file contains two sections: the device selection section and the tunable parameters section. We have already described how to add an entry to the tunable parameters section to set a parameter defined in the master file (see "Parameters: The Keyword Section"). As described, entries to this section are optional.

The system file entry contains information that the driver will need to access the device, particularly hardware I/O addresses. The system file entry consists of a *device specification* (described below) that encodes the necessary device information in its syntax. See the **system**(4) man page for more information.

You must add entries to the device selection section for each physical device of your driver's device type. A typical set of device entries for our **xdev** device might be as follows:

```
xdev@72()
xdev@73(ffff6000,4)
```

Here, xdev is the entry name for the master file device description entry. The number 72 is the device code for the first controller, and 73 is the device code of the second controller of this particular class of device. The empty parentheses () in the first entry indicate that the default parameters, including the default base address, apply for this device. The second instance of the xdev device shows a non-standard base address and a second parameter of four (4). The parameter's meaning will be specific to the driver's implementation.

End of Appendix

Appendix C
Rebuilding the System and Checking Configuration

This appendix describes the procedure for rebuilding the kernel with your driver integrated into it. It also describes several ways in which you can check to see if the **system** and **master** file setup was correctly handled. We will use **xxx** as your device mnemonic.

Compile-time Checklist

You compile the file containing your driver routines (in **dev_xxx_driver.c**) and global data (in **dev_xxx_global_data.c**) with your **dev_xxx_def.h** and the appropriate system include files to produce object files that will be linked into the system image. The appropriate system include files must include:

- Include files for the kernel itself

  All drivers must include three files that contain constants and data structures used by the kernel itself. These files are **c_generics.h**, **os_generics.h**, and **architecture.h**. These files are found in **/usr/src/uts/aviion/ext**.

- Include files for kernel-supplied routines

  If you use a kernel-supplied routine, you will need to include an include file specific to that routine's class. The routine's class is indicated by the first few letters of its name. The include file for a class of routines starts with these same few letters. For example, if you use a virtual memory ("vm") routine, like **vm_wire_memory**, you must include the **i_vm.h** include file. The possible include files are listed in Table C-1 below.

Table C-1 Routine Classes and Their Include Files

Routine Class	Acronym	Include File
File system	fs	**i_fs.h**
I/O	io	**i_io.h**
Lock management	lm	**i_lm.h**

Miscellaneous	misc	**i_misc.h**
Process management	pm	**i_pm.h**
System control	sc	**i_sc.h**
Virtual memory	vm	**i_vm.h**
Virtual process	vp	**i_vp.h**
Micro-code	uc	**i_uc.h**

These files are stored in **/usr/src/uts/aviion/ii.**

There are also three defines (STANDALONE, KERNEL and _PRODUCT_DGUX) used during compilation as shown in the next section. If you want to avoid specifying these defines in the compile line, you can add them to one of your source files.

If you use ANSI C function prototypes in your driver's source code, you will also need the define, _STDC_. As before,you can define this either in your source files or at compile time. If _STDC_ is defined, the kernel-supplied include files in **/usr/src/uts/aviion/ii** will also appear to use ANSI style function prototype declarations.

Rebuilding and Rebooting the System

You use the standard system-generation procedure, **sysadm**, to build a new system image. However, before you use **sysadm**, you must complete the following steps:

1. Make your changes to the system file and master file as described in Appendix B. We recommend you create your own master file for your master file entries and put it in **usr/etc/master.d**. You may give this file any name you want as long as it does not match any existing file names in the **master.d** directory.

2. Compile your driver file **dev_xxx_driver.c** to create the object file **dev_xxx_driver.o.**

   If you compile using the GNU compiler that comes with the DG/UX system, we recommend you use the following compile command line:

```
gcc -c -DSTANDALONE -DKERNEL -D_PRODUCT_DGUX
                              -fno-omit-frame-pointer
                              -mno-underscores
                              -I/usr/src/uts/aviion dev_xxx_driver.c
```

   If you compile using the Green Hills compiler, we recommend you use the following compile command line:

```
ghcc -c -DSTANDALONE -DKERNEL -D_PRODUCT_DGUX
                             -ga -X58 -X153  -X405
                             -I/usr/src/uts/aviion dev_xxx_driver.c
```

3. Place your driver object file and any archive files you may need into the directory **/usr/src/uts/aviion/lb.**

4. Create a file called **Libs.***driver_name* that lists all the object files and archive files you want included in the build. Place this file in the directory **/usr/src/uts/aviion/cf.** You can get the format of this file by examining other **Libs.** files.

Once you have completed these steps you are ready to build a new system. *Installing the DG/UX™ System* describes how to use **sysadm** to build a new kernel. The output of the build is a new system image that you will move to the root directory (**/**).

After the new system image is ready, you can shut down the current system and reboot.

Checking the Configuration Process

To verify that your device is properly configured, check both **conf.c** and the special files for your devices. We describe both of these sources below.

The Conf.c File

The **conf.c** file contains the system tables generated by the **config** program. You can use these structures to verify your configuration and to determine the location of your driver's routines. It is found in **/usr/src/uts/aviion/Build.** A partial listing of **conf.c** structures and variables is given below with descriptions on how to use the information to verify proper configuration.

The Configurable Variable Section

The configurable variable section lists the variables as defined in the keyword section of the master files and modified in the tunable parameters section of the system file. You can check this section for the proper setting of any parameters you set. A partial listing of this section is given below:

```
/*
/* Configurable Variable Section  */
/* ---------------------------    */
/*                                */
char          cf_sc_machine[]                      =     "AViiON";
char          cf_sc_sysname[]                      =       "dgux";
char          cf_sc_release[]                      =       "4.30";
char          cf_sc_version[]                      =         "00";
uint16_type   cf_sci_daylight_savings_time_kind    =             1;
uint8_type    cf_sfm_max_modules_per_stream        =             9;
uint32_type   cf_sfm_max_data_message_length       =          4096;
uint32_type   cf_sfm_max_control_message_length    =          1024;
```

```
uint16_type    cf_ps_max_semaphore_sets            =         10;
uint16_type    cf_ps_max_semaphores_per_set        =         25;
   .
   .
   .
```

Device Driver Tables

The kernel uses an internal table of device driver structures to configure new devices into the system. The Device Driver Table in **conf.c** listed below shows this table of driver entries. You should use this table to verify that your driver has been added to the list. Each such entry contains (as its first structure element) a pointer to the routines vector for that driver. Chapter 2 explains how you supply a routines vector for your driver.

```
/* Device Drivers Table  */
/* --------------------- */

extern void * cfv_sd_routines_vector;
extern void * cfv_st_routines_vector;
extern void * cfv_cisc_routines_vector;
extern void * cfv_duart_routines_vector;
extern void * cfv_syscon_routines_vector;
extern void * cfv_devtty_routines_vector;
extern void * cfv_mem_routines_vector;
extern void * cfv_ldm_routines_vector;
extern void * cfv_err_routines_vector;
extern void * cfv_xdev_routines_vector;

struct driver_table_type
    {
    void *  routines_vector_ptr;
    void *  streamtab_ptr;
    char *  streams_set_name;
    int     is_streams_device   :1;
    int     uses_svr4_interface :1;
    int     padding             :30;
    };

struct driver_table_type cf_io_driver_table [] =
[
    {
    (void *) &cfv_sd_routines_vector,
    (void *) 0,
    "default",
    0,
    0,
    0
    },
{
(void *) &cfv_st_routines_vector,
(void *) 0,
"default",
0,
0,
0
},

{
(void *) &cfv_cisc_routines_vector,
(void *) 0,
"default",
0,
0,
0
},

{
(void *) &cfv_duart_routines_vector,
(void *) 0,
"default",
0,
0,
0
},

{
(void *) &cfv_syscon_routines_vector,
(void *) 0,
"default",
0,
0,
0
},

{
(void *) &cfv_devtty_routines_vector,
(void *) 0,
"default",
0,
0,
0
},

{
(void *) &cfv_mem_routines_vector,
(void *) 0,
"default",
0,
0,
0
},
        {
        (void *) &cfv_ldm_routines_vector,
        (void *) 0,
        "default",
        0,
        0,
        0
        },

        {
        (void *) &cfv_err_routines_vector,
        (void *) 0,
        "default",
        0,
        0,
        0
        },

        {
        (void *) &cfv_xdev_routines_vector,
        (void *) 0,
        "default",
        0,
        0,
        0
        },

    };

    uint16e_type cf_io_driver_table_count = 10;
```

The Configuration List

The configuration list shows all the devices configured on the system. Check for all your system file entries.

```
    /* ------------------- */
    /* Configuration List */
    /* ------------------- */

    char * cf_init_configuration_list [] =
        {
        "syscon()",
        "cied()",
        "devtty()",
        "mem()",
        "ldm()",
        "st(cisc(0,0),*)",
        "syac(0)",
"err()",
"con()",
"xdev@72()",
"xdev@73(0xffff6000,4)",
"pts()",
"ptc()",
"prf()",
"hken()",
"meter()",
"loop()",
""
};
```

Your Special Files

At reboot time, the system will call the driver's configuration routine which should generate a special file for all devices of your driver's type in the **/dev** directory. One way to check configuration is to list the files for your devices to verify their setup. There should be a special file for each unit serviced by your driver. The devices shown here will reflect those you specified in the system file and should be named according the correct device specification. Listing the special files with the ls -l command will display the major and minor device numbers of each unit, as well as the access permissions. You can also verify that appropriate special files exist for block versus character access for a device.

End of Appendix

Appendix D Using STREAMS in the DG/UX Multiprocessor Environment

DG/UX STREAMS is fully compatible with industry standard STREAMS. As Chapter 1 notes, the *Programmer's Guide: STREAMS (UNIX System V Release 4)* describes how to write code for such industry standard STREAMS. You need only create a master file entry with default settings for each STREAMS module and driver to integrate industry standard code transparently into the DG/UX system.

However, DG/UX STREAMS also allows you to access some additional features of the DG/UX kernel that are not available under the basic industry standard STREAMS. Chapter 1 mentions that you can add some additional routines via a routines vector. The *Writing a Device Driver for the DG/UX System* manual describes how to create this routines vector and these additional routines. This appendix looks at a second, more important option under DG/UX STREAMS, how to take advantage of the symmetric multiprocessor environment without changing your code. The feature that allows you to do this is called *concurrency sets*. Concurrency sets are controlled solely through your master file entries.

This appendix describes what concurrency sets are and how to use them and then discusses how different concurrency sets affect performance.

What Are Concurrency Sets?

Industry standard STREAMS does not yet support symmetric multiprocessing and portability is very important in a STREAMS program. As we've seen, one of the keys to programming in a fully symmetric environment is guaranteeing that only one process has access to critical sections of code or shared data at the same time. Normally, to run concurrently you would have to add locks to your code to guarantee exclusivity when the process is accessing critical code or data. But adding locks makes the code less portable. Concurrency sets allow you to define which modules/drivers can run concurrently and which cannot by setting a flag in their master file entry. Thus, concurrency sets support multiprocessing STREAMS while maintaining complete portability.

A concurrency set is most easily understood as a set of modules that are associated with a common lock. Notice that different modules sharing a common lock means that modules in the same set cannot run at the same time because only one member of the set can hold the common lock at a time. On the other hand, modules that do not share the same lock can run at the same tiem. Thus, paradoxically members of the *same* concurrency set *cannot* run concurrently while members of *different* concurrency sets *can*.

The master file entry lets the kernel's STREAMS management code know which modules/drivers share a common lock. Whenever a message is passed between modules, this management code checks to see if the receiving module's lock is available. If it is, the new module can run. If it is not, the new module is put on a deferral list awaiting its lock's release.

This is the basic idea of concurrency sets. In order to describe their use we need to look more carefully at STREAMS modules and drivers in action. To do this let us first introduce some basic terminology.

Terminology

The three basic components of a stream are: a stream head, modules, and a driver.

- **Stream Head** — The stream head is the code segment at the top of the stream. It is the interface between user space and kernel space for a stream and provides synchronization between the stream and kernel/user space.

- **Module** — A module is a stream component that manipulates data. Users can push one or more modules onto the stream between the stream head and the driver using a special **ioctl** system call. Every module has a read queue and a write queue. The read queue holds data that is going up the stream and the write queue holds data that is going down the stream. A module's job is to respond to messages coming in on its own read/write queues and send messages to the read/write queues of modules above and below it on the stream.

- **Driver** — The driver is a special case of a module; it is the module at bottom of the stream. Because it is at the bottom, it serves as the external interface for the stream. Drivers often address hardware devices, but they may also address pseudodevices.

- **Messages** — Messages are the items that are passed along the stream. They contain data and other state information about the type of message etc. Messages can be passed downstream (from the stream head towards the driver) or upstream (from the driver towards the stream head). Modules perform various operations on message information as it is passed up and down the stream.

Figure D-1 shows a basic stream. The stream head is at the top, the driver is at the bottom, and in between you have some number of optional modules.

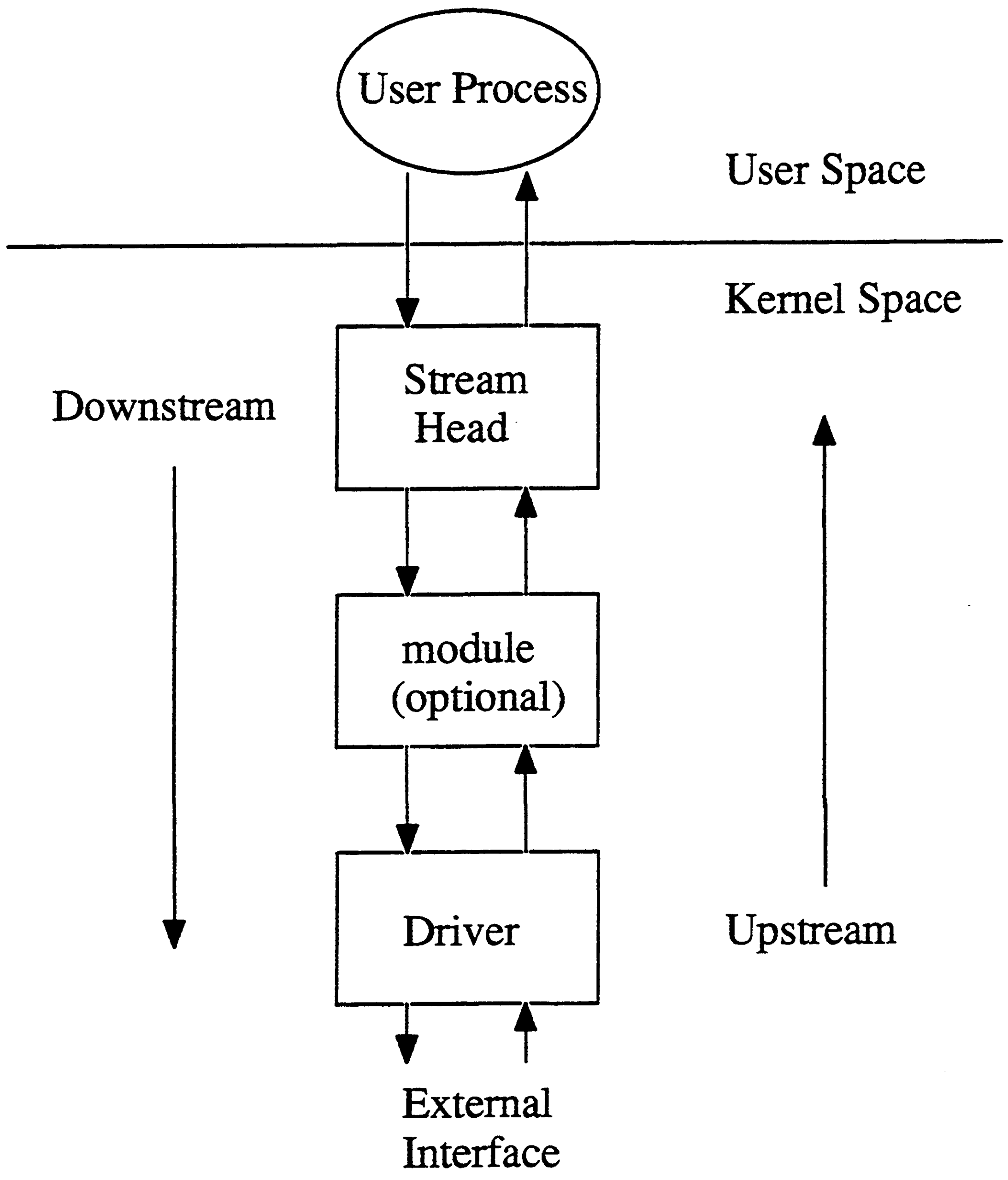


Figure D-1 *Basic STREAM Layout*

Figure D-2 shows an example of a more complicated set of streams, the type that frequently arise in real world applications. This example shows multiple streams feeding into a TCP/IP communications driver.

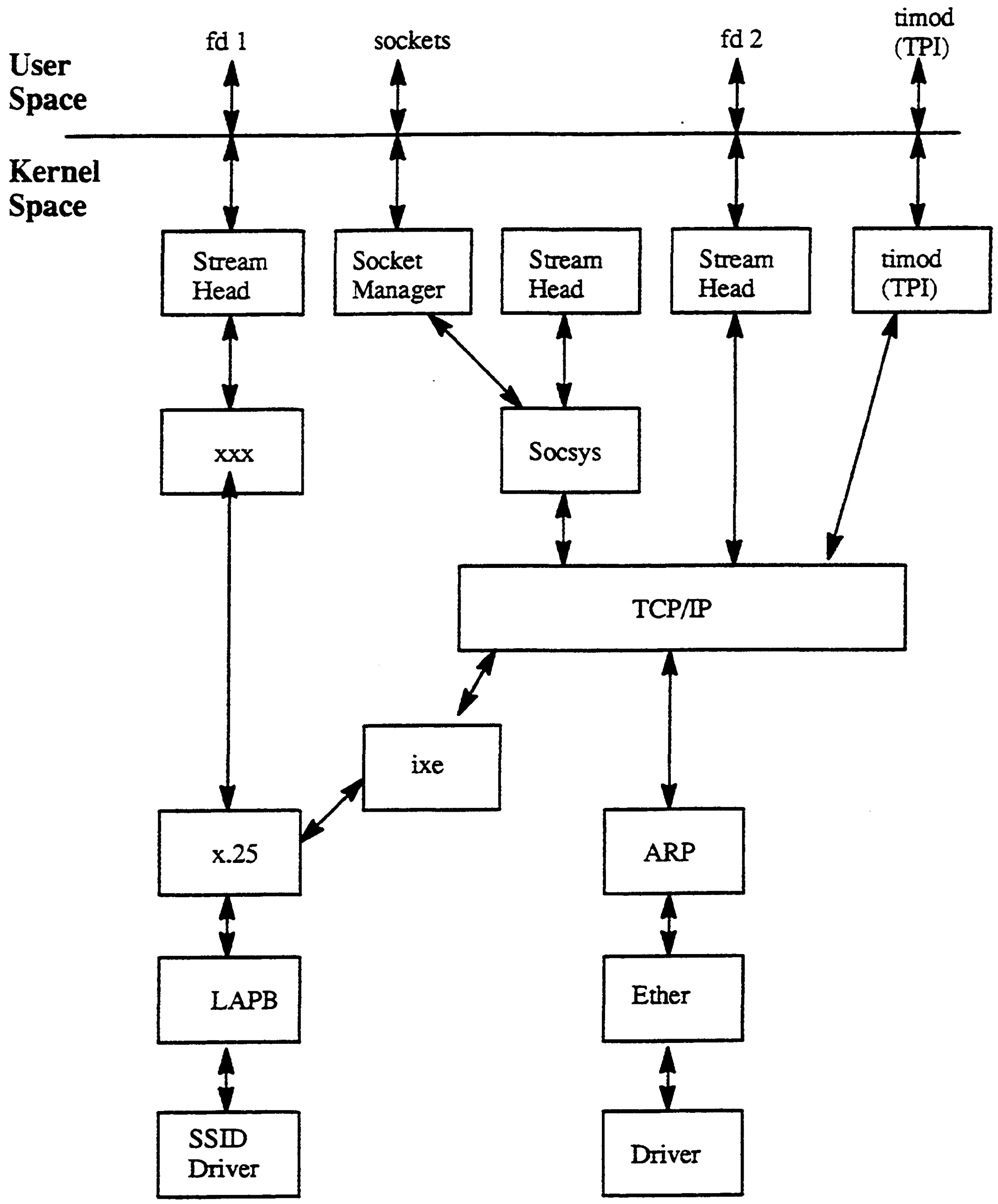


Figure D-2 *Multiplexor STREAM*

Because several streams end at TCP/IP, it is a driver. But the TCP/IP module also branches out into several other streams. A module that maps one or more upper streams to one or more lower streams is called a multiplexor.

How Different Types of Concurrency Sets Work

A concurrency set associates a lock with a set of modules. It provides mutual exclusion for all the modules belonging to the set. A single lock protects all the modules in the set from colliding with each other.

There are four types of concurrency sets, that is, four ways that you can associate a lock with a module. The four types are: per-stream, per-module, set and default concurrency. Note that the label "none" is reserved for future use and should not be used as a concurrency set name.

You define a module/driver's concurrency in its master file entry by entering "stream" for per-stream, "module" for per-module, a set name for set concurrency and "default" for default concurrency. Thus, in the example shown below, **hken** is per-stream, **sd** is per-module, the **xdev** driver belongs to a set concurrency set named "george" and the **cird** driver used "default" concurrency.

```
#-----------------------------------------------------
# Disks:
#  Name            Restriction    Concurrency
#  Prefix          Flags          Set
#  ----------      --------       ------------
   hken            n              stream
   inen            n              module
   ttcodpat        n              george
   ldterm          n              default
#
#-----------------------------------------------------
```

If a module or driver has per-stream concurrency, it uses the same lock as its stream head. Figure D-3 shows per-stream concurrency.

Per–stream Concurrency

* Within each stream, the stream–head, module, and driver all share a common lock.

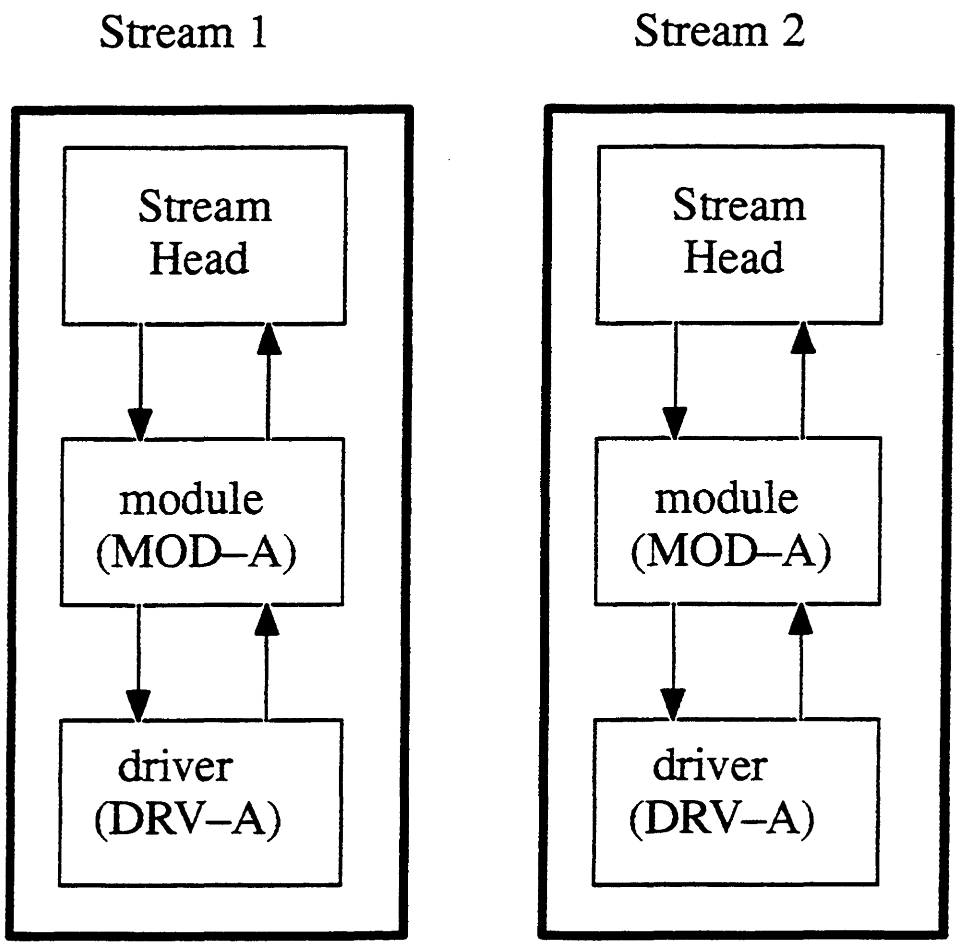
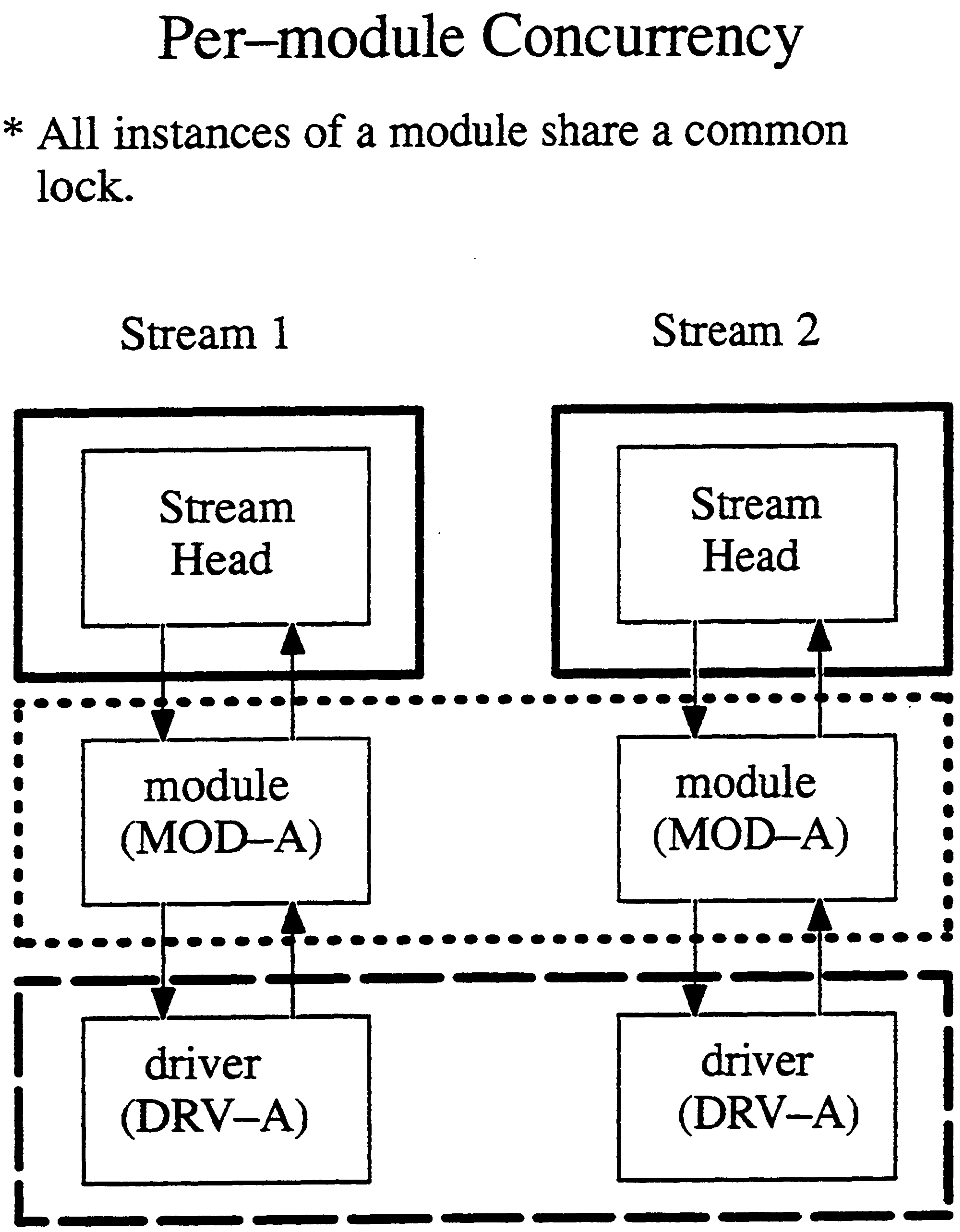


Figure D-3 *Per-stream Concurrency*

A stream-head is by definition in per-stream concurrency. This cannot be changed. In this example, the module, MOD-A, and the driver, DRV-A, have also been defined as per-stream concurrency in their master file entries. Notice that both streams have instances of MOD-A and DRV-A. Per-stream concurrency means that the instances of MOD-A and DRV-A share a lock with their respective stream heads. The big box around each stream shows that the stream's components are in the same concurrency set and in a different concurrency set from the neighboring stream. This means that STREAM-1 and STREAM-2 can run concurrently because they use different locks. Conversely, the stream-head, MOD-A, and DRV-A in each stream run separately because they all share the same lock.

In per-module concurrency, all instances of a module or driver have the same lock. Figure D-4 shows per-module concurrency.


Per–module Concurrency

* All instances of a module share a common lock.

Figure D-4 *Per-module Concurrency*

The module, MOD-A and the driver, DRV-A have been defined as having per-module concurrency. Both STREAM-1 and STREAM-2 contain instances of the module, MOD-A. The box around these two instances of MOD-A show that they share the same lock, which means each instance will run exclusive of the other. The same logic holds for the two instances of DRV-A as well. The stream heads are, as always, per-stream concurrency and thus use a different lock from both MOD-A and DRV-A. Consequently, the stream-head, MOD-A, and DRV-A of STREAM-1 can all run concurrently. In this example, exclusivity holds only between instances of MOD-A and DRV-A.

Note that module concurrency for drivers is handled on a major number basis. Thus, if you have the same driver with different major numbers for it, then only instances with the same major number will have the same lock.

The final type of concurrency is called set concurrency. In set concurrency, modules belong to the set named in the master file. The set name can be any legitimate UNIX name string. Figure D-5 shows set concurrency in which modules MOD-A and MOD-B are both part of the the "george" concurrency set.

Set Concurrency

* All instances of one or more modules share a common lock.

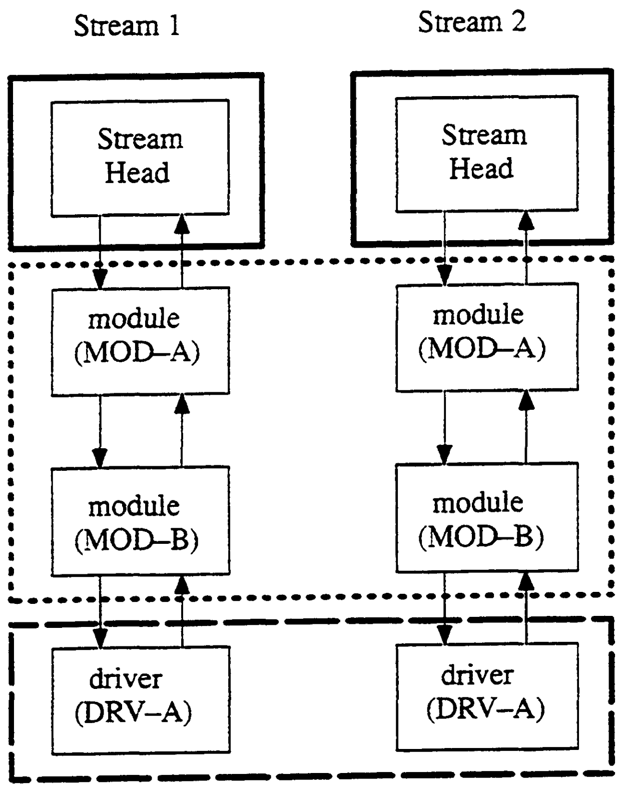


Figure D-5 *Set Concurrency*

Default concurrency is actually a special case of set concurrency. All modules with default concurrency belong to the same set. Thus, members of the default set run exclusively of each other and concurrently with other sets.

Note that if you define a driver to be in set concurrency, all instances of that driver will have the same lock regardless of the major number. Only in per-module concurrency does the major number make a difference.

Recommendations on How to Use Concurrency Sets.

We recommend you define all modules/drivers as per-stream concurrency except in the three special case conditions listed below. Using per-stream concurrency provides more concurrency than the default concurrency set and it avoids complicated overhead costs that can arise in other forms of concurrency. Note that stream heads are always defined as per-stream concurrency.

The three exceptions to this recommendation are as follows:

- All multiplexors must be defined as per-module, set or default concurrency. They cannot be defined as per-stream concurrency because they don't belong exclusively to any one stream-head (they map multiple upper streams to multiple lower streams).

- Because you must provide mutual exclusion for shared data, instances of modules that share data, or multiple modules that share data must belong to the same concurrency set.

- If you must be guaranteed that a **putnext** procedure runs immediately when called, then adjacent modules must be in the same concurrency set. The need to guarantee that the **putnext** runs immediately sometimes arises in state driven situations; for example, a module needs to know that a particular action has occurred when it thinks it has occurred, and it relies on the return value of the **putnext**. Thus, if you set up the **putnext** procedure to return an integer and it must come back accurately, then two adjacent modules must be in the same set.

Notes on Creating STREAMS Code on the DG/UX System

STREAMS kernel programming is generally the same as other kernel programming. If you are creating STREAM code from scratch and portability is not an issue, you can use fine-grained locks, eventcounters and most of the other facilities described in the body of this manual. However, there are a few kernel facilitites, such as buffer vectors and select management routines, that are not appropriate for STREAMS code. STREAMS restrictions, if any, are noted in the chapters that cover particular topics. Be sure to create master file entries for all STREAMS modules and drivers. Stream-heads do not require master file entries because their concurrency set is always per-stream concurrency. Use the system file entry to request either a default routines vector or your own. See *Writing a Standard Device Driver for the DG/UX System* for more information on how to create your own routines vector.

In addition, you should observe the following guidelines:

- Use the standard STREAMS utilities to manipulate queues and pass messages to other modules. Using your own routines to traverse the stream and manipulate queues is particularly dangerous in a multiprocessing environment. In particular, always use **putnext**() to pass messages between modules. Do not create your own **queue_t** structures and manipulate them with standard STREAMS utilities.
- Use only the **putq** and **qenable** STREAMS utilities in your interrupt handler at interrupt level. No other STREAMS utilites may be used.
- Explicitly protect shared data in interrupt handlers.
- Don't make assumptions about major number assignments for device drivers.

Notes on Porting STREAMS Code to the DG/UX System

The list of programming guidelines given above holds for ported code as well as newly created code. Check your program for violations of these guidelines before attempting to port. In particular, note the restrictions on STREAMS utilities at interrupt level.

In addition, there are four AT&T STREAMS utilities that do not hold on the DG/UX system. These are: **splstr**(), **splx**(), **sleep**(), and **wakeup**(). **Splstr**() and **splx**() are used to enable and disable interrupts. In AT&T's single-processor environment, such routines can be used to provide mutual exclusion. However, on a symmetric multiprocessor system, disabling interrupts does not provide exclusivity (as Chapter 1 describes). **Sleep**() and **wakeup**() do not work in the DG/UX environment for similar multiprocessing reasons.

Remove instances of **splstr**() and **splx**(). Review the code to see if the exclusion that these functions provided must be replaced through concurrency sets or by adding your own locks using the kernel-supplied locking routines. Replace instances of **wakeup**() with **su_str_wakeup**(). Replace instances of **sleep**() with **su_str_get_next_event**() and **su_str_sleep**() as shown in the coding examples below. Be sure to use **su_str_get_next_event**() to get the event before using **su_str_sleep**() to await it. The first example shows the AT&T version of the code, followed by the DG/UX equivalent.

```
                              AT&T CODE

n = splstr();
while ( condition )

      {
      if (sleep(resource, PCATCH))
            {
            /*a signal was detected*/
            }
       }

[optionally perform mutually exclusive operations
after event has occurred.]

splx ( n );
-----------------------------------------------------------------------

                              DG/UX CODE

next_event = su_str_next_event ( resource );
while ( condition )

      {
      if (su_str_sleep(resource, next_event))
            {
            /*a signal was detected*/
            }
       }
 [mutual exclusion is now guaranteed.]
```

The routines used in these examples are described on the next several pages.

su_str_sleep

Syntax

```
int      su_str_sleep(

caddr_t               resource,       /* READ ONLY*/
vp_ec_value_type      next_ec_value   /* READ ONLY*/
                      )
```

Summary

This routine awaits an event specified by the eventcount value **next_ec_value** or the eventcounter specified by **resource** or a process interrupt event.

Parameters

resource — A pointer to the resource being awaited.

next_ec_value — The value of the eventcount to await.

Description

This routine awaits a **su_str_wakeup(resource)** operation that advances the eventcounter associated with **resource** to **next_ec_value**. This routine should be used only by a module or driver open/close routine.

Return Values

0 — sleep was awakened by wakeup

1 — sleep was awakened by an interrupt signal (process is either being terminated or ut must handle a signal).

su_str_wakeup

Syntax

```
void        su_str_wakeup(

caddr_t     resource        /* READ ONLY*/
                            )
```

Summary

This routine causes the eventcount associated with **resource** to be advanced.

Parameters

resource — A pointer to a resource being freed.

Description

See Summary.

Return

None.

Exceptions

None.

su_str_next_event

Syntax

```
vp_ec_value_type      su_str_next_event(

caddr_t    resource   /* READ ONLY*/
```

Summary

This routine returns the next value of an eventcount associated with **resource**.

Parameters

resource — A pointer to a resource being sought.

Description

See Summary.

Return

The value that the eventcount associated with **resource** will attain when it is next advanced is returned.

Exceptions

None.

Index

Note: Boldfaced page numbers (e.g., **1-5**) indicate definitions of terms or other key information.

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

   Send your order form with payment to: Data General Corporation
   ATTN: Educational Services/TIPS G155
   4400 Computer Drive
   Westboro, MA 01581-9973

   b) TELEPHONE – Call TIPS at (508) 870-1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870-1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME	COMPANY NAME
ATTN:	ATTN:
ADDRESS	ADDRESS (NO PO BOXES)
CITY	CITY
STATE ZIP	STATE ZIP

Priority Code ____________ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) | Title | Date | Phone (Area Code) | Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 - $149.99	0%
$150 - $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) ____________

ORDER TOTAL	
Less Discount See B	-
SUB TOTAL	
Your local* sales tax	+
Shipping and handling - See A	+
TOTAL - See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is________. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number | Expiration Date

Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Programming in the DG/UX™ Kernel Environment

093–701083–00

Cut here and insert in binder spine pocket

Data General

Data General Corporation, Westboro, Massachusetts 01580