digital

# VAX/VMS

## Real-Time User's Guide

Order No. AA-H784A-TE

VAX11

March 1980

This manual discusses VAX/VMS features of interest to real-time users. It also provides programming examples illustrating certain important or complex features.

# VAX/VMS
## Real-Time User's Guide

Order No. AA-H784A-TE

**SUPERSESSION/UPDATE INFORMATION:** This is a new document for this release.

**OPERATING SYSTEM AND VERSION:** VAX/VMS V02

**SOFTWARE VERSION:** VAX/VMS V02

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, March 1980

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL
DEC
PDP
DECUS
UNIBUS
COMPUTER LABS
COMTEX
DDT
DECCOMM
ASSIST-11
VAX
DECnet
DATATRIEVE
DECsystem-10
DECtape
DIBOL
EDUSYSTEM
FLIP CHIP
FOCAL
INDAC
LAB-8
DECSYSTEM-20
RTS-8
VMS
IAS
TRAX
MASSBUS
OMNIBUS
OS/8
PHA
RSTS
RSX
TYPESET-8
TYPESET-11
TMS-11
ITPS-10
SBI
PDT

## CONTENTS

## CONTENTS

## FIGURES

## CONTENTS

## FIGURES (Cont.)

## TABLES

# PREFACE

## MANUAL OBJECTIVES

The VAX/VMS Real-Time User's Guide describes VAX/VMS features of interest to real-time application programmers. It describes in general terms functions common to a variety of real-time applications and explains the specific VAX/VMS feacures available to perform these functions. This manual also contains numerous examples, including coding segments and complete programs, to illustrate certain important or complex features.

## INTENDED AUDIENCE

This manual is intended for programmers writing real-time applications. You are assumed to have substantial programming experience and some knowledge of basic VAX/VMS concepts (see "Associated Documents" in this preface).

The programming examples are in VAX-11 MACRO and VAX-11 FORTRAN. Each example, however, is designed to be as meaningful as possible for programmers using any other VAX-11 language.

## STRUCTURE OF THIS DOCUMENT

This manual covers a variety of topics, usually proceeding from less complex to more complex material. Wherever appropriate, this manual relates a topic to other topics discussed elsewhere in the manual.

Chapter 1 introduces the manual. It summarizes the real-time features covered in the manual, describes other features of possible interest and refers to appropriate documentation, and explains some significant concepts.

Chapter 2 discusses ways to control the program execution environment, including creating subprocesses and detached processes and affecting the allocation of physical memory.

Chapter 3 covers mechanisms for communicating between cooperating processes, synchronizing their activities, and sharing data and code.

Chapter 4 discusses real-time I/O, including mapping I/O space and connecting to a device interrupt vector.

Chapter 5 discusses the use of software facilities located in multiport (shared) memory -- specifically common event flag clusters, mailboxes, and global sections.

Chapter 6 explains privileged shareable images, a vehicle that allows you, in effect, to write your own system services.

Chapter 7 provides several complete programming examples with accompanying explanations.

The appendixes present supplementary information. Appendix A shows how to use a common event flag or a queue as a mutual exclusion (mutex) semaphore to lock a resource. Appendix B discusses programming and design considerations for users of the Laboratory Peripheral Accelerator (LPA11-K). Appendix C provides a programming example in VAX-11 BLISS-32. Appendix D is a checklist of optimization techniques for real-time users.

## ASSOCIATED DOCUMENTS

The following manuals explain the VAX/VMS concepts that are prerequisite knowledge for readers of this manual:

- The VAX/VMS Summary Description and Glossary explains the major components of the VAX/VMS system and defines significant terms.

- The VAX-11/780 Technical Summary (order number EA-15963-20) describes the major components and features of the VAX/VMS system.

The following manuals provide more detailed treatment of major concepts and features described in this manual:

- The VAX/VMS System Manager's Guide discusses the system generation (SYSGEN) utility, the user authorization file (UAF), system tuning, and the DISPLAY utility.

- The VAX/VMS System Services Reference Manual provides tutorial chapters on many topics covered in this manual. It also explains the format and requirements for each system service.

- The VAX/VMS I/O User's Guide discusses I/O programming in detail, including chapters on several real-time devices.

- The VAX/VMS Guide to Writing a Device Driver explains how to write your own device driver and includes detailed information on VAX/VMS I/O.

The user's guide for each programming language provides information on using VAX/VMS features and capabilities with that language.

The following handbooks provide information on VAX-11 architecture and hardware:

- The VAX-11 Architecture Handbook (order number EB-17580-18) introduces VAX-11 system architecture, explains addressing modes, and presents the native-mode instruction set.

- The VAX-11/780 Hardware Handbook (order number EB-17835-18) explains VAX-11 hardware elements, including the high-speed synchronous backplane interconnect (SBI), the central processor unit, intelligent console subsystem, MASSBUS and UNIBUS subsystems, main memory, and memory management. This handbook also includes an appendix explaining restrictions on program references to I/O space.

## CONVENTIONS USED IN THIS DOCUMENT

The system service formats and coding example conventions are consistent with those used in the VAX/VMS System Services Reference Manual:

| **Convention** | **Meaning** |
|---|---|
| UPPERCASE | Uppercase letters in a system service format show material that must be entered as shown. |
| lowercase | Lowercase letters in a system service format show variable data. |
| [ ] | Brackets in a system service format indicate an optional argument. |
| ... | Horizontal ellipsis in a coding example indicates that additional arguments necessary for the system service call but not pertinent to the example are not shown. |
| .<br>.<br>. | Vertical ellipsis in a coding example indicates that lines of code not pertinent to the example are not shown. |

# CHAPTER 1

# INTRODUCTION

"Real-time" is a term whose meaning varies with specific applications. However, in most scientific, industrial, and commercial real-time applications, one or both of the following are critical needs:

- High throughput

- Fast response

Applications for which high throughput is essential require the continuous processing of large amounts of data. An example of a throughput-intensive application is signal processing, which is used in speech research, electrocardiogram and electroencephalogram research, vibration analysis, and music synthesis. As another example, a stream of data points is required for many of the qualitative and quantitative methods used in gas and liquid chromatography, mass spectrometry, automatic titration, and colorometry.

In all of these throughput-intensive applications, the primary requirement is to obtain some number of data points equally spaced in time. Some further computation is done, perhaps later, on the data collected.

In other real-time applications, fast response to individual events is the most critical requirement. A typical example that requires fast response is a closed-loop control system. In such a case, some event must be identified as soon as possible; a decision is then made and an output variable is updated. For example, before a jet engine is tested, sensing instruments connected to a processor running a control program might be placed on and near the engine. After the engine is started, the control program must be able to detect, analyze, and correct any abnormality within a few milliseconds -- for instance, by shutting off the engine before an explosion occurs. Applications for which response time is a critical factor include process monitoring and control, synchronous communications, and stimulus-response testing in biological and psychological research.

If response time is critical, the designer must ensure that the application has all the resources it needs immediately whenever it needs them. These resources include:

- CPU time, the availability of which is affected by process priority and, perhaps, interrupt latency

- Memory, which can be controlled by several system services (see Chapter 2)

- I/O bandwidth, which is determined by the hardware configuration

These two real-time requirements, high throughput and responsiveness, are sometimes interrelated. For example, if your application must collect large amounts of data quickly and if the data acquisition is to be triggered by an external event, you need both fast response and high throughput.

Specific real-time applications might involve the following types of programming activities:

- Controlling the program's execution environment, which might require communicating between programs and creating subprocesses or detached processes
- Using the Queue I/O Request system service directly, to achieve faster response and greater throughput
- Coordinating programs running on multiple processors, including the sharing of multiport memory units

Real-time users often employ sophisticated means to make the system respond best to their special processing needs. The VAX/VMS system provides tools to meet these needs.

## 1.1 REAL-TIME NEEDS AND VAX/VMS FEATURES

From its inception, the VAX/VMS system has been designed to meet the real-time processing needs of a wide user base. The VAX-11 architecture provides the necessary hardware foundation with its high I/O bandwidth, interrupt responsiveness, 32-bit processing capabilities, and real-time peripheral interfaces. These architectural features are described in the hardware documentation for your system (see the Preface). This manual will focus on software features. Its approach is to identify functions common to a variety of real-time applications, discuss these functions conceptually, and show how specific VAX/VMS features can be used to perform these functions.

You are assumed to be familiar with basic VAX/VMS concepts, which are defined in the VAX/VMS Summary Description and Glossary. Do not, however, confuse the VAX/VMS term "process" (the program image and the software context in which it executes) with "process" in its generic sense (a sequence of events), as in "industrial process-control applications." Most instances of the word "process" in this manual refer to the image and its context; any other use will be clearly identified.

Table 1-1 summarizes common real-time needs and the features or capabilities available with VAX/VMS to meet these needs. Each feature listed is documented in the VAX/VMS System Services Reference Manual unless another manual is specified. The goal of the present manual is to organize and highlight aspects of special interest to real-time users.

Table 1-1
Real-Time Needs and VAX/VMS Features

| Real-Time Need | VAX/VMS Feature |
|---|---|
| Perform an operation with or after another operation | Use the Create Process ($CREPRC) service to create a subprocess or detached process<br>Use the RUN command to create a subprocess or detached process (see the VAX/VMS Command Language User's Guide) |
| Change the availability of a process for scheduling | Use the Set Priority ($SETPRI) service |
| Keep critical code or data highly accessible | Use the Adjust Working Set ($ADJWSL) system service to adjust the amount of physical memory a process is entitled to use<br>Use the Lock Pages in Memory ($LCKPAG) system service to keep pages in physical memory<br>Use the Lock Pages in Working Set ($LKWSET) system service to keep pages in physical memory as long as the process is in memory<br>Use the Set Process Swap Mode ($SETSWM) system service to keep all or part of a process in physical memory<br>Use the Create and Map Section ($CRMPSC) system service to map a file into process address space |
| Perform I/O quickly or for special purposes | Use the Queue I/O Request ($QIO) system service<br>Map I/O space (using the $CRMPSC service) and/or connect to a device interrupt vector (using the $QIO service)<br>Write your own device driver (see the VAX/VMS Guide to Writing a Device Driver) |
| Synchronize a process with an external event or program | Set and wait for event flags<br>Code and declare asynchronous system trap (AST) service routines<br>Connect to a device interrupt vector<br>Cause processes to hibernate or suspend, and to awaken when needed |

(continued on next page)

Table 1-1 (Cont.)
Real-Time Needs and VAX/VMS Features

| Real-Time Need | VAX/VMS Feature |
|---|---|
| Share code or data between processes | Use the Create and Map Section ($CRMPSC) system service to create and map a global section<br>Use shareable images (see the VAX-11 Linker Reference Manual) |
| Send messages to other processes | Use mailboxes ($CREMBX system service creates mailbox; RMS or I/O system services read and write messages) |
| Use multiport memory (memory shared by multiple processors) | Use common event flag clusters, global sections, and mailboxes located in a shared memory unit |
| Use special-purpose system services | Write privileged shareable images (see Chapter 6) |

## 1.2 OTHER VAX/VMS TOOLS

There are other VAX/VMS tools which may be of interest to some real-time users, but which are outside the scope of this manual. Brief descriptions of these tools follow, with references to other manuals for detailed information.

### 1.2.1 Condition Handling

A condition handler is a procedure that is given control when an exception occurs. An exception is an event that is detected by the hardware or software and that interrupts the execution of an image. Examples of exceptions include arithmetic overflow or underflow and reserved opcode or operand faults.

If you want to handle any or all exceptions yourself, you must code and declare a condition handler. Information on condition handling is available in the VAX/VMS System Services Reference Manual, the VAX-11 Run-Time Library Reference Manual, and the language user's guides.

### 1.2.2 Device Allocation

You can allocate and deallocate devices from within your program with the Allocate Device ($ALLOC) and Deallocate Device ($DALLOC) system services. Allocating a device reserves it for exclusive use by the requesting process. The VAX/VMS System Services Reference Manual explains the $ALLOC and $DALLOC system services.

### 1.2.3 SYSGEN Parameter Selection

There are a number of parameters to the SYSGEN utility whose values affect the paging, swapping, and scheduling operations of the system. All of these parameters have default values that DIGITAL has selected as suitable for a wide range of users; however, real-time users may wish to modify certain parameters or experiment with different combinations of parameters. The VAX/VMS System Manager's Guide discusses major SYSGEN parameters and provides some guidelines for selecting their values. That manual also discusses a number of parameters in relation to system tuning.

### 1.2.4 User Authorization File Entries

The user authorization file (SYSUAF.DAT) includes entries within each record to determine the base priority (PRIORITY), initial working set limit (WSDEFAULT), maximum working set limit (WSQUOTA), and privileges for that user's processes. The VAX/VMS System Manager's Guide explains the user authorization file entries.

### 1.2.5 Networks

A VAX/VMS system can be connected in a communications network to other DIGITAL processors with the same or different operating systems. The family of software products supporting these networks is called DECnet. You can use DECnet to share files and communicate between programs on different processors; however, for faster performance you can use one of the real-time devices mentioned in Section 1.3. For information on the use of DECnet, see the DECnet-VAX User's Guide and the DECnet-VAX System Manager's Guide.

## 1.3 REAL-TIME DEVICES

The following devices are especially suited for real-time applications:

- Laboratory Peripheral Accelerator (LPA11-K)
- Parallel Communications Link (PCL)
- 32-bit Parallel SBI Interface (DR780)
- Synchronous Communications Line Interface (DMC11)
- Multiport Memory (MA780)

This section discusses several of these devices only briefly. For detailed information on using the MA780, see Chapter 5. For information on the other devices, see the VAX/VMS I/O User's Guide and the appropriate hardware documentation.

The LPA11-K controls analog-to-digital (A/D) and digital-to-analog (D/A) converters, digital I/O registers, and real-time clocks. Appendix B discusses programming and design considerations for LPA11-K users.

The DR780 can be used to link user devices to a processor or processors to each other. The DR780 provides a very high-speed 32-bit wide interface to the VAX-11 Synchronous Backplane Interconnect (SBI).

The DMC11 and the MA780 are used primarily to link processors. The MA780 offers memory-access speed and greater capabilities, but the DMC11 is suited for data transmission between processors separated by a great distance. The DIGITAL Data Communications Message Protocol (DDCMP) programmed into the DMC11's microprocessor ensures data integrity.

## 1.4 USER PRIVILEGES FOR REAL-TIME APPLICATIONS

To protect the integrity of the system, VAX/VMS restricts certain functions or operations to processes with the appropriate user privileges. Each process starts with a set of privileges established in one of the following ways:

- For each user who logs in, privileges are designated by the system manager in the user's entry in the user authorization file.

- For each created process, privileges are specified or defaulted in the PRVADR argument to the Create Process ($CREPRC) system service or the /PRIVILEGES qualifier to the RUN command.

You can change a process's privileges in two ways: at the command level with the SET PROCESS/PRIVILEGES command and at the program level with the Set Privileges ($SETPRV) system service.

Most timesharing users need and are given only a limited set of privileges. Real-time users, however, are normally given considerably more privileges, because they need them to perform certain functions. Any privileges required for functions discussed in this manual are documented here or in the VAX/VMS System Services Reference Manual.

Some of the privileges of special interest to real-time users are as follows:

| **Privilege** | **Meaning** |
|---|---|
| ALTPRI | Set process base priority higher than user's own base priority |
| BYPASS | Bypass all UIC-based protection checks |
| CMEXEC | Change mode to executive |
| CMKRNL | Change mode to kernel |
| EXQUOTA | Exceed certain quotas |
| GROUP | Control processes in user's own group |
| GRPNAM | Place entries in group logical name table |
| LOG_IO | Perform logical I/O operations |
| OPER | Perform operator functions |
| PFNMAP | Map to section by physical page frame number |
| PHY_IO | Perform physical I/O |
| PRMCEB | Create permanent common event flag clusters |
| PRMGBL | Create permanent global sections |
| PRMMBX | Create permanent mailboxes |
| PSWAPM | Change process swap mode |
| SETPRV | Grant process privileges other than own current privileges |
| SHMEM | Perform certain functions in memory shared by multiple processors |
| SYSNAM | Place entries in system logical name table and create system-wide global sections |
| SYSPRV | Access resources as if you have a system user identification code (UIC) |
| WORLD | Control any process in the system |

The VAX/VMS System Manager's Guide explains these and the other privileges in greater detail.

### 1.4.1 Privilege Masks

User privileges are stored in a quadword (64-bit) mask, in which specific bits correspond to specific privileges. The operating system actually maintains four separate privilege masks for each process:

- AUTHPRIV - Privileges that the process is authorized to enable, as designated by the system manager or the process creator. The AUTHPRIV mask never changes during the life of the process.

- PROCPRIV - Privileges that are designated as permanently enabled for the process. The PROCPRIV mask can be modified by the Set Privileges ($SETPRV) system service or the SET PROCESS/ PRIVILEGES command.

- IMAGPRIV - Privileges that the current image is installed with.

- CURPRIV - Privileges that are currently enabled. The CURPRIV mask can be modified by the Set Privileges ($SETPRV) system service or the SET PROCESS/PRIVILEGES command.

When a process is created, its AUTHPRIV, PROCPRIV, and CURPRIV masks have the same contents. Whenever a system service must check the process's privileges, it checks the CURPRIV mask. When a process runs a known image, the privileges that the image was installed with are enabled in the CURPRIV mask. Whenever an image exits, the PROCPRIV mask is copied to the CURPRIV mask.

## 1.5 PROCESS QUOTAS

To prevent a process from monopolizing or overusing certain resources, VAX/VMS enforces a number of quotas (limits) on each process. These quotas can be adjusted for each process. The system manager can set quotas for each user in the user authorization file (UAF), and the creator of a detached process or subprocess can specify quotas with the QUOTA argument to the Create Process ($CREPRC) system service (see Section 2.1.3) or with qualifiers to the RUN command (see Section 2.1.4). Default values are used for any quotas not specified.

Each quota is deductible, pooled, or nondeductible:

- A deductible quota value is subtracted from its creator's current value when a subprocess is created and returned to the creator when the subprocess is deleted.

- A pooled quota is shared by a detached process and all its descendent subprocesses. Charges against a pooled quota value are subtracted from the current available total as the resource is used and are added back to the total when the resource is not being used.

- A nondeductible quota is established and maintained separately for each detached process and subprocess.

The VAX/VMS System Services Reference Manual contains more detailed information on process quotas.

Table 1-2 lists each process quota, its function, the defaults used for the user authorization file (UAF) and for process creation, and the minimum value. The table also indicates whether the quota is deductible, pooled, or nondeductible.

Table 1-2
Summary of Process Quotas

| Quota | Function[1] | UAF Default Value | Process Creation Default | Min. Value |
|---|---|---|---|---|
| AST queue limit (ASTLM) | Limits the sum of ASTs and scheduled wake-up requests that can be pending for a process at one time (N) | 10 | 6 | 2 |
| Buffered I/O count limit (BIOLM) | Limits the number of I/O operations that the process can have buffered in system memory (N) | 6 | 6 | 2 |
| Buffered I/O byte count limit (BYTLM) | Limits the number of bytes that the process can use for system buffered I/O operations (P) | 4096 | 8192 | 1024 |
| CPU time limit (CPUTIME) | CPU time limit in milliseconds (0 means no limit) (D) | 0 | 0 | 0 |
| Direct I/O count limit (DIOLM) | Limits the number of I/O operations that the process can have buffered in process address space (N) | 6 | 6 | 2 |
| Open file limit (FILLM) | Limits the number of files that the process can have open at one time (P) | 20 | 10 | 2 |
| Paging file quota (PGFLQUOTA) | Limits the number of pages that the process can use in the system paging file (P) | 10000 | 2048 | 256 |
| Subprocess creation limit (PRCLM) | Limits the number of subprocesses that the process can create (P) | 8 | 8 | 0 |
| Timer queue entry limit (TQELM) | Limits the sum of timer queue entries and temporary common event flag clusters that the process can have at one time (P) | 10 | 8 | 0 |
| Default working set size (WSDEFAULT) | Sets the initial working set size for the process (N) | 150 | 100 | 50 |
| Working set size limit (WSQUOTA) | Limits the size to which the process's working set size can be expanded (N) | 200 | 120 | 50 |

[1] After each "Function" description is a letter in parentheses indicating whether the quota is deductible (D), pooled (P), or nondeductible (N).

### 1.5.1 Resource Wait Mode

By default, a process enters resource wait mode whenever it needs but cannot obtain system dynamic memory or a resource controlled by any of the following quotas:

- Direct I/O limit (DIOLM)
- Buffered I/O limit (BIOLM)
- Buffered I/O byte count limit (BYTLM)

(If any other resource controlled by a quota is unavailable, the process receives the SS$_EXQUOTA error status code.) Resource wait mode places the process in a wait state until the resource becomes available.

In a real-time environment, however, it may not be practical or desirable for a program to wait. In these cases, you can choose to disable resource wait mode for the process, so that when a required resource is unavailable, control returns immediately to the calling program with an error status code. You can disable resource wait mode with the Set Resource Wait Mode ($SETRWM) system service.

How a program responds to the unavailability of a resource depends very much on the application and the particular system service that is being called. In some instances, the program may be able to continue execution and retry the service call later. In other instances, it may be necessary only to note that the program is being required to wait.

## 1.6 PROCESS PRIORITY

At any given time, each process has a priority that affects how it runs relative to other processes in the system. Process priorities can range from 0 through 31, with 0 through 15 designated as timesharing priorities and 16 through 31 designated as real-time priorities.

The "base priority" of a process refers to its minimum priority. You can adjust a process's base priority with the Set Priority system service or the SET PROCESS/PRIORITY command. The priority that affects process operations is its current priority (or simply, priority), which the system dynamically adjusts for timesharing processes.

The system handles timesharing and real-time priorities in different ways. For processes with timesharing base priorities (0 through 15), the system dynamically adjusts the priority according to the process's state and other factors. The actual priority of a timesharing process at any given time might be as much as 7 higher than its base priority. However, the system will never raise a priority in the timesharing range to a real-time level. Furthermore, the system does not alter the priority of a process with a real-time base priority (16 through 31).

When you log in, your initial base priority is determined by a value in your record in the user authorization file. When you create a subprocess or detached process, its initial base priority is determined by the specified or default value for the BASPRI argument to the Create Process ($CREPRC) system service or for the /PRIORITY qualifier on the RUN command. To find out the base priority of your process, you can use the SHOW PROCESS command.

### 1.6.1 Significance of Process Priority

The priority of a process can affect

- How quickly it is scheduled (that is, becomes the current process) after it becomes executable
- Whether it will be interrupted by the scheduling of another process
- Whether it will be swapped out of the balance set if the system needs the physical memory for another process
- How quickly its queued I/O requests are serviced by a device driver

The VAX/VMS scheduler always selects the highest-priority process from among those that are eligible to execute, that is, processes that are "computable" (process state) and in the balance set. (Conditions that can cause a process not to be executable include waiting for an event flag to be set or a resource to become available, or being in a state of hibernation or suspension.) If a lower-priority process is executing and a higher-priority process becomes executable, the lower-priority process is interrupted and the higher-priority process receives control of the processor.

If the working set requirements of all processes in the balance set exceed the system's available physical memory, the VAX/VMS swapper process is activated to "outswap" one or more processes: that is, to save certain information and the working set of each process to be swapped out and to free its memory pages for use by other processes. A real-time process requiring fast response, however, should not be swapped out. In selecting a process for outswapping, VAX/VMS considers the process's state and quantum value in addition to its priority. Therefore, if you must guarantee that a real-time process will not be swapped out, disable swapping for the process with the Set Process Swap Mode ($SETSWM) system service (see Section 2.2.4).

The VAX/VMS system also uses process priority as the basis for ordering I/O requests queued to a driver. That is, the system initiates a queued I/O request issued by a higher-priority process before it initiates one for the same device issued by a lower-priority process.

Because the VAX/VMS operating system's own processes normally have priorities of 16 or lower, real-time users must ensure that one of these system processes is not blocked from execution if its operation is needed by a real-time process. For example, if several real-time processes are in the system, a priority-22 process performing disk file I/O can be blocked by a compute-bound priority-17 process that is preventing the disk ACP (which might be priority 11) from executing. If an operating system process needs to perform functions for a real-time process, you might have to raise the priority of the system's process.

### 1.6.2 Adjusting the Base Priority

Raising process priority can decrease the time required for a program to run to completion. Programs running in real-time processes have more predictable execution times, because the process usually waits only for the completion of requests that it initiates; it does not spend time wating for lower-priority processes to execute.

The higher the process's priority is set, the less likely it is the process will have to wait. However, you must use discretion in raising priorities, because as you increase the number of real-time processes executing concurrently, you potentially decrease the effectiveness of each priority designation.

User privileges are required to set the priority of any process other than your own or to raise the priority of any process (including your own) higher than your own base priority. The following user privileges enable you to perform the indicated functions:

- The GROUP privilege allows you to change the priority of other processes in your group.

- The WORLD privilege allows you to change the priority of any other processes in the system.

- The ALTPRI privilege allows you to set the priority of any process whose priority you have privilege to change (see GROUP and WORLD privilege explanations) higher than your own base priority. If you do not have the ALTPRI privilege, you can set the priority of any process whose priority you have privilege to set only equal to or lower than your own base priority.

There are two ways to change the base priority of a process:

- At the command level with the command:

  ```
  $ SET PROCESS/PRIORITY=n
  ```

- At the program level with the Set Priority ($SETPRI) system service

The Set Priority system service is probably more useful to real-time programmers than the SET PROCESS/PRIORITY command, because the system service enables you to set process base priorities dynamically according to the program's logic. This service has the following general formats:

**MACRO Format**

```
$SETPRI [pidadr],[prcnam],pri,[prvpri]
```

**High-Level Language Format**

```
SYS$SETPRI([pidadr],[prcnam],pri,[prvpri])
```

The _VAX/VMS System Services Reference Manual_ has a detailed explanation of the Set Priority system service.

# CHAPTER 2

# CONTROLLING THE PROGRAM EXECUTION ENVIRONMENT

The VAX/VMS system gives you considerable control over the execution context of your applications, provided you have suitable user privileges. Each application runs in the context of one or more processes and can control that context in the following ways:

- Create processes (subprocesses or detached processes) to divide the work into related segments
- Set each process's base priority to achieve real-time responsiveness
- Control each process's use of physical memory

You can use these features to ensure that all components of a real-time application receive adequate processor time and physical memory when they need them.

Process base priority is discussed in Section 1.6. Process creation and control of physical memory are discussed in this chapter.

The DISPLAY utility allows you to monitor system activity, and thus to obtain information that can guide you in using features discussed in this chapter. The VAX/VMS System Manager's Guide explains the functions and operation of the DISPLAY utility.

The Get Job/Process Information ($GETJPI) system service can also be used to obtain information about one or more processes. The VAX/VMS System Services Reference Manual explains the Get Job/Process Information system service, including the "wild card" process searching capability.

## 2.1 PROCESS CREATION

Real-time applications are often divided into a number of programs. Each program might run concurrently with one or more others, and each might run conditionally (for example, only when certain events occur).

The VAX/VMS system allows you to create processes to run these programs. These created processes can be subprocesses or detached processes, depending on your purpose and user privileges.

You can create either type of process with the Create Process ($CREPRC) system service or with the RUN command, although real-time applications frequently create subprocesses with the $CREPRC system service and detached processes with the RUN command (often within a command procedure at the start of the application). Section 2.1.3 discusses the $CREPRC system service, and Section 2.1.4 discusses the RUN (Process) command.

### 2.1.1 Subprocesses and Detached Processes

Subprocesses and detached processes are treated the same by the scheduling and swapping components of the operating system. For example, each process of either type has a base priority that the system uses in scheduling processes, allocating CPU time, and deciding which process to swap out if necessary. Both types of process are shown in the displays generated by the SHOW SYSTEM command and the DISPLAY utility.

Subprocesses and detached processes differ, however, in their degree of independence from their creator and in the privileges and quotas required to use them. Table 2-1 summarizes the major differences between a subprocess and a detached process.

Table 2-1
Subprocess versus Detached Process

| Subprocess | Detached Process |
|---|---|
| 1. Shares creator's resources and its deductible and pooled quotas | 1. Has own resources and quotas |
| 2. Must terminate before its creator; automatically terminated when its creator is deleted | 2. Termination is independent of its creator's |
| 3. No privilege required to create a subprocess | 3. DETACH privilege required to create a detached process |
| 4. Number of subprocesses is limited by creator's PRCLM quota | 4. Number of detached processes is limited only by the system's maximum total process count (SYSGEN parameter MAXPROCESSCNT) |
| 5. Can access devices allocated by its creator | 5. Must allocate devices it needs to reserve for exclusive use |

A process does not need GROUP privilege to use system services or commands that affect any subprocess it creates (for example, to change the subprocess's priority). A process does need GROUP or WORLD privilege, however, to affect a detached process (GROUP if the detached process is in its group, otherwise WORLD).

### 2.1.2 Real-Time Uses of Detached Processes and Subprocesses

Real-time applications often create detached processes to perform highly privileged functions and subprocesses to perform functions requiring little or no privilege. Isolating privileged code as a detached process makes it easier to debug and affords greater protection for the system as a whole. Once it is created, a detached process is more insulated than a subprocess from any errors its creator may incur, because a detached process terminates independently of its creator's termination, whereas a subprocess is automatically deleted under the following conditions:

- If the subprocess was created by a process that is using the command interpreter (for example, by the process created for you at login time), the subprocess is deleted when its creating process logs out.

- If the subprocess was created by a process that is not using the command interpreter (for example, by another subprocess or a detached process executing a single image), that subprocess is deleted when its creator is deleted.

A process can explicitly delete itself or, if it has suitable privilege, another process by using the Delete Process ($DELPRC) system service. The WORLD privilege allows you to delete any process in the system; the GROUP privilege allows you to delete other processes in your own group.

### 2.1.3 Create Process System Service

The Create Process ($CREPRC) system service gives you program-level control over the creation of subprocesses and detached processes. For example, you might simply create a process at the beginning of the program and control that created process's activity through the hibernation or suspension mechanisms (see Chapter 3). On the other hand, you might need to test values within your program or wait for some external event before creating another process. In any case, process creation is relatively time consuming, and therefore should be used prudently in real-time programs.

The Create Process system service has the following general formats:

**MACRO Format**

```
$CREPRC [pidadr],[image],[input],[output],[error],
        [prvadr],[quota],[prcnam],[baspri],[uic],
        [mbxunt],[stsflg]
```

**High-Level Language Format**

```
SYS$CREPRC([pidadr],[image],[input],[output],[error],
           [prvadr],[quota],[prcnam],[baspri],[uic],
           [mbxunt],[stsflg])
```

The following arguments to $CREPRC are of special interest to real-time users:

- UIC - Determines whether the created process is a subprocess (no UIC specified -- UIC same as creator) or a detached process (UIC specified).

- PRVADR - Allows you to specify privileges for the created process. To give the created process any privilege the creator does not have, you must have the SETPRV privilege.

- BASPRI - Allows you to specify a base priority for the created process. To assign the created process a base priority higher than the creator's own, you must have the ALTPRI privilege.

- STSFLG - Allows you to specify various options for the created process.

For a detailed explanation of the Create Process system service, see the VAX/VMS System Services Reference Manual.

### 2.1.4 RUN (Process) Command

The RUN command creates a subprocess or detached process to run a specified program if you enter any of the process-related command qualifiers (that is, any qualifier other than /DEBUG or /NODEBUG). The general format for the RUN command to create a subprocess or detached process is listed as follows:

```
$ RUN/command-qualifiers  program-file-spec
```

Each of the process-related command qualifiers is optional, although you must enter at least one. The presence of the /UIC command qualifier determines whether the created process is a detached process (qualifier specified) or a subprocess (qualifier not specified). The process-related command qualifiers and their default values are listed below.

| **Qualifier** | **Default (if applicable)** |
|---|---|
| /[NO]ACCOUNTING | /ACCOUNTING |
| /AST_LIMIT=quota | 10 (outstanding ASTs) |
| /[NO]AUTHORIZE | |
| /BUFFER_LIMIT=quota | 10240 (bytes) |
| /DELAY=delta time | |
| /ERROR=file-spec | |
| /FILE_LIMIT=quota | 20 (files) |
| /INPUT=file-spec | |
| /INTERVAL=delta-time | |
| /IO_BUFFERED=quota | 6 (outstanding requests) |
| /IO_DIRECT=quota | 6 (outstanding requests) |
| /MAILBOX=unit | |
| /MAXIMUM_WORKING_SET=quota | 200 (pages) |
| /OUTPUT=file-spec | |
| /PRIORITY=n | (same as creator) |
| /PRIVILEGES=privilege-list | (same as creator) |
| /PROCESS_NAME=process-name | (null name) |
| /QUEUE_LIMIT=quota | 8 (outstanding timer queue requests) |
| /[NO]RESOURCE_WAIT | /RESOURCE_WAIT |
| /SCHEDULE=absolute-time | |
| /[NO]SERVICE_FAILURE | /NOSERVICE_FAILURE |
| /SUBPROCESS_LIMIT=quota | 8 (subprocesses) |
| /[NO]SWAPPING | /SWAPPING |
| /TIME_LIMIT=limit | 0 (that is, no limit) |
| /UIC=uic | |
| /WORKING_SET=default | 200 (pages) |

The /UIC, /PRIVILEGES, and /PRIORITY qualifiers serve the same purposes as the UIC, PRVADR, and BASPRI arguments to the Create Process system service (see Section 2.1.3).

The VAX/VMS Command Language User's Guide has a complete explanation of the RUN command and the process-related qualifiers.

You may want to include RUN commands for process creation in command procedures. The following example shows a command procedure that prompts for information and then creates a subprocess.

```
$INQUIRE DEVICE "Device name"         !Specify input device
$INQUIRE TEST "Test name"             !Specify program to be run
$INQUIRE INTERVAL "How often should it be reported?  (0:mm:ss)"
$RUN/PROCESS_NAME='TEST'/PRIORITY=19/INPUT='DEVICE'/OUTPUT=OPA0:-
        /INTERVAL='INTERVAL'  'TEST'
```

## 2.2 PHYSICAL MEMORY CONTROL

Physical memory is one of the most valuable system resources to a real-time user. Programs execute faster when the code and data they need at any given instant are already in memory and do not need to be retrieved from disk storage.

In brief, VAX/VMS memory management operates in the following way. The pages of a process that are currently in physical memory (usually a subset of all the process's pages) constitute that process's working set. The maximum number of physical memory page frames a process can occupy is determined by its current working set limit. When the number of page frames in use reaches the working set limit and the process needs additional pages, the system pages the process against itself. That is, the system releases pages in the working set (placing each one on the free page list or the modified page list) and then reads the pages it needs from disk or finds them in memory (on the free page list or the modified page list). If and when the working set requirements of all processes in the balance set (that is, processes currently in memory) exceed the available physical memory, one or more lower-priority processes are swapped out (temporarily removed from the balance set) and their page frames are made available for use by other processes. For more detailed information on VAX/VMS memory management, see the VAX/VMS Summary Description and Glossary or the VAX-11/780 Technical Summary. For information on parameters to the SYSGEN utility affecting memory management, see the VAX/VMS System Manager's Guide.

Several system services allow you to control the operating system's allocation of physical memory to the process. The following services are most pertinent to real-time manipulation of physical memory:

- Adjust Working Set Limit ($ADJWSL)
- Lock Pages in Memory ($LCKPAG)
- Lock Pages in Working Set ($LKWSET)
- Set Process Swap Mode ($SETSWM)

The subsections that follow give brief descriptions and general formats for these services. For more detailed information, see the VAX/VMS System Services Reference Manual.

### 2.2.1 Adjusting the Working Set Limit ($ADJWSL)

The Adjust Working Set Limit ($ADJWSL) system service allows you to increase or decrease the maximum number of physical memory pages your process can occupy. You can also use this system service to find your current working set limit. (You can change and find out your working set limit at the command level with the SET WORKING_SET and SHOW WORKING_SET commands.)

The VAX/VMS system normally performs automatic working set adjustment. However, automatic working set adjustment is inhibited for all processes if you specified WSINC=0 to the SYSGEN utility, and automatic working set adjustment is inhibited for a given process if the process has a real-time priority (16 through 31) or if the process's working set default value is equal to its working set quota (maximum) value. The VAX/VMS System Manager's Guide explains automatic working set adjustment and the SYSGEN parameters that affect its operation.

One of the simplest forms of memory management is to change the working set limit at different points in your program. Large programs usually proceed in phases; for example, a program might perform a heavily I/O-bound setup phase, then settle into localized compute-bound processing, then do discontiguous array processing, and so forth. If your code has definable phases, you may want to call the $ADJWSL system service at logical points to increase or decrease the working set limit.

Another use of this system service is to prevent the excessive paging activity that occurs when a program runs in too small a working set.

You should avoid excessive use of this system service, however, because it incurs overhead for your process and perhaps for other processes in the system.

No user privilege is required to use the $ADJWSL system service. However, you cannot set a process's working set limit lower than the system's minimum limit (determined by the SYSGEN parameter MINWSCNT) or higher than the process's maximum working set size (determined by its WSQUOTA entry in the UAF or specified when the process was created).

The Adjust Working Set Limit system service has the following general formats:

**MACRO Format**

```
     $ADJWSL [pagcnt],[wsetlm]
```

**High-Level Language Format**

```
     SYS$ADJWSL([pagcnt],[wsetlm])
```

### 2.2.2 Keeping Pages in the Working Set ($LKWSET)

The Lock Pages in Working Set ($LKWSET) system service allows you to specify that a page or range of pages should not be replaced in the working set, perhaps because these pages are heavily used or because the code in them must gain control and execute quickly whenever it is needed. If the specified pages are not already in the working set, they are brought into memory if necessary and locked in the working set. Pages locked in the working set remain so until they are unlocked by the Unlock Pages from Working Set ($ULWSET) system service.

Pages locked in the working set can be removed from physical memory, however, if their process is swapped out (that is, if the process's working set is removed from the balance set). To prevent this from happening, use the Set Process Swap Mode ($SETSWM) system service to disable swapping (see Section 2.2.4).

Locking pages in the working set is normally sufficient to guarantee that their contents are accessible, especially if swapping is disabled for the process. However, in a few cases you may need to lock the pages in memory using the Lock Pages in Memory ($LCKPAG) system service (see Section 2.2.3), to guarantee that the physical location of the contents never changes. These cases include the following:

- The process must lock pages for a routine that will execute at an elevated interrupt priority level (IPL). Section 4.6.1 discusses interrupt priority levels.

- The process is not using the VAX/VMS I/O system and must lock pages for direct I/O operations.

If you use the $LKWSET system service, be careful not to lock so many pages that the remaining pages in the working set incur too many page faults. If excessive page faulting occurs, you may need to increase the working set limit with the Adjust Working Set Limit ($ADJWSL) service (see Section 2.2.1).

The Lock Pages in Working Set system service has the following general formats:

**MACRO Format**

```
$LKWSET inadr,[retadr],[acmode]
```

**High-Level Language Format**

```
SYS$LKWSET(inadr,[retadr],[acmode])
```

The general format of the Unlock Pages from Working Set system service is the same as the above, except that $ULWSET or SYS$ULWSET is used instead of $LKWSET or SYS$LKWSET.

### 2.2.3 Keeping Pages in Memory ($LCKPAG)

The Lock Pages in Memory ($LCKPAG) system service locks a virtual page or range of virtual pages in physical memory. If the specified virtual pages are not already in memory, they are brought into the working set and then locked in memory. Locked pages are not available for page replacement until they are unlocked by the Unlock Pages from Memory ($ULKPAG) system service or until the program terminates (locked pages are unlocked automatically at image exit). You must have the PSWAPM user privilege to lock pages in memory.

It is usually not necessary to lock pages in memory; locking them in the working set is often sufficient. (Section 2.2.2 discusses cases in which pages should be locked in memory.) Use caution, however, because locking any pages in memory reduces by that number the pages that VAX/VMS memory management can allocate among other processes in the system.

Locked pages remain in memory even if their process is swapped out. To prevent the process from being swapped out, use the Set Process Swap Mode ($SETSWM) system service to disable swapping (see Section 2.2.4).

The Lock Pages in Memory system service has the following general formats:

**MACRO Format**

```
$LCKPAG inadr,[retadr],[acmode]
```

**High-Level Language Format**

```
SYS$LCKPAG(inadr,[retadr],[acmode])
```

The general format of the Unlock Pages in Memory system service is the same as the above, except that $ULKPAG or SYS$ULKPAG is used instead of $LCKPAG or SYS$LCKPAG.

### 2.2.4 Keeping the Process in Memory ($SETSWM)

The Set Process Swap Mode ($SETSWN) system service enables you to prevent your process from being swapped out of memory or to allow it to be swapped out of memory. You must have the PSWAPM user privilege to alter process swap mode.

An example of real-time use of setting process swap mode is a process running an image that must respond quickly to some external event (such as an interrupt), but has nothing to do until the event occurs. After it is activated, the image can lock critical pages in its working set (see Section 2.2.2), disable swapping for the process, and hibernate. (It is important to disable swapping, because being in a hibernate state normally makes a process a good candidate for outswapping.) When the event occurs, an AST service routine (see Section 3.3) can awaken the process.

The Set Process Swap Mode system service has the following general formats:

**MACRO Format**

```
$SETSWM [swpflg]
```

**High-Level Language Format**

```
SYS$SETSMW([swpflg])
```

The SWPFLG argument can be a value of 0 (the default, to allow swapping) or 1 (to inhibit swapping).

# CHAPTER 3

# COMMUNICATING AND SHARING BETWEEN PROCESSES

Real-time applications often consist of related programs running as several processes. These processes may be detached processes, or they may be a detached process with one or more subprocesses. These processes usually need to communicate with each other and to share common code or data. Interprocess communication often consists of event notification (for example, that an I/O operation is complete), although it can also involve transmission of messages or other data. Processes within the application can synchronize their operations through effective communication. Processes can also share code or data to reduce the application's physical memory requirements.

Table 3-1 lists several VAX/VMS features that can be used to communicate between user processes, synchronize their operations, or share code and data.

Table 3-1
Features for Communication, Synchronization, and Sharing

| Feature | Main Use |
|---|---|
| Common event flags | Notify process of event completion; synchronize access to a resource |
| Mailboxes | Pass messages or other data between processes |
| AST service routines | Execute desired routine in response to an external event, regardless of when the event occurs |
| Hibernation and suspension | Activate subprocesses and detached processes only when they are needed |
| Global sections | Share data or code |
| Shareable images | Share data or code |

Each feature listed in Table 3-1 is often used with one or more other features. For example, an AST service routine executing at I/O completion might write a message to a mailbox to be read by another process or might set an event flag for which another process is waiting.

## 3.1 COMMON EVENT FLAGS

Common event flags provide a simple and convenient means for event notification. Cooperating processes can set, clear, and wait for flags in a common event flag cluster.

Common event flags can be used to synchronize access to a resource by multiple processes. Appendix A discusses and illustrates the use of a common event flag as a mutual exclusion (mutex) semaphore to lock a resource.

Event flags are status-posting bits maintained by VAX/VMS for general programming use. Each process can manipulate up to 128 event flags, numbered 0 through 127. The event flags are grouped into four clusters of 32 flag bits each; however, whenever you set, clear, or wait for an event flag, you specify the flag number, not a cluster number or name. (The significance of the cluster name for common event flag clusters is discussed later in this section.)

The first two clusters, flags 0 through 31 and 32 through 63, are called local event flags because they are available only to a single process. Two additional clusters, flags 64 through 95 and 96 through 127, are called common event flag clusters because they can be used by cooperating processes. Table 3-2 summarizes local and common event flag clusters.

Table 3-2
Summary of Event Flag Clusters

| Event Flag Numbers | Description | Restriction |
|---|---|---|
| 0-23<br>32-63 | Local event flag clusters for general use by a process | Event flags 24 through 31 are reserved for system use |
| 64-95<br>96-127 | Common event flag clusters | Must be associated before use |

Common event flag clusters are either temporary or permanent (depending on the PERM argument value in the Associate Common Event Flag Cluster system service call).

Temporary common event flag clusters:

- Do not require any special user privilege, but do use part of the calling process's timer queue entries (TQELM) quota.
- Are deleted when all processes associated with the cluster have disassociated from it. A process can disassociate explicitly using the Disassociate Common Event Flag Cluster ($DACEFC) service, or it can disassociate implicitly at image exit.

Permanent common event flag clusters:

- Require the creating process to have the PRMCEB user privilege.
- Continue to exist until they are explicitly marked for deletion with the Delete Common Event Flag Cluster ($DLCEFC) service and no processes are associated with them.

This section will present general formats and focus on aspects pertinent to real-time applications. Chapter 5 discusses special considerations for common event flag clusters in shared (multiport) memory.

The VAX/VMS System Services Reference Manual has a chapter on event flag usage and detailed description of event flag services.

### 3.1.1 Creating and Associating with Clusters

To create or associate with a common event flag cluster, use the Associate Common Event Flag Cluster ($ASCEFC) system service, which has the following general formats:

**MACRO Format**

```
$ASCEFC efn,name, [prot],[perm]
```

**High-Level Language Format**

```
SYS$ASCEFC(efn,name,[prot],[perm])
```

The first process specifying a given name creates the cluster and associates with it; any other processes specifying this name associate with the existing cluster. All processes associating with the same common event flag cluster must specify the same name, but they do not have to specify event flag numbers in the same 32-bit grouping. You can allow any other process in your group to associate with the cluster (the default) or restrict association to processes with your UIC (by specifying a PROT argument value of 1). You can make the cluster temporary (the default) or permanent (by specifying a PERM argument value of 1).

### 3.1.2 Setting Event Flags

You can set event flags in a variety of ways. The following system services accept an optional EFN argument, which specifies an event flag to be set when the operation is completed:

- Queue I/O Request ($QIO and $QIOW forms, $INPUT and $OUTPUT macros)
- Set Timer ($SETIMR)
- Update Section File on Disk ($UPDSEC)
- Get Job/Process Information ($GETJPI)

Note that each of the above system services clears the specified event flag before it begins the requested operation.

You can also set an event flag using the Set Event Flag ($SETEF) system service. To clear an event flag, use the Clear Event Flag ($CLREF) system service. Both the $SETEF and $CLREF system services accept only one argument: EFN, a value indicating the flag to be set or cleared.

### 3.1.3 Waiting for Event Flags

If a process needs to be activated only in response to one or more events, you can use one of the following system services to place the process in a wait state until it must execute:

- $WAITFR - The Wait for Single Event Flag system service places the process in a wait state until a single specified event flag has been set.
- $WFLOR - The Wait for Logical OR of Event Flags system service places the process in a wait state until any one of a specified group of event flags has been set.
- $WFLAND - The Wait for Logical AND of Event Flags system service places the process in a wait state until all of a specified group of event flags have been set.

During this wait state the process can still receive asynchronous system trap (AST) interrupts, but after the AST service routine completes, the process automatically reexecutes the "Wait for..." service call.

After the flag or flags have been set and the process has responded to the event(s), the process can reenter the wait state by looping back to the appropriate system service call.

## 3.2 MAILBOXES

A mailbox is a record-oriented virtual I/O device that cooperating processes can use to send messages, status information, return codes; or other data to each other. A mailbox must be created using the Create Mailbox and Assign Channel ($CREMBX) system service. Any other process that needs to use the mailbox simply assigns an I/O channel to the mailbox using the $CREMBX system service or the Assign I/O Channel ($ASSIGN) system service. Actual data transfer (reading and writing) involving the mailbox is accomplished by using I/O system services, RMS, or high-level language I/O statements.

Mailboxes are suited to sending messages that cannot be conveyed by the simpler and faster operations of setting and clearing event flags. Mailboxes can hold multiple messages, which are read on a first-in first-out (FIFO) basis, whereas with an event flag you cannot determine from a flag's current status how many times it has been set or cleared. Some overhead is involved, however, with the use of mailboxes. Therefore, to pass and read messages faster you can use a global section (see Section 3.5) to hold the messages and common event flags to notify processes that messages are ready to be read.

A special use of a mailbox is as a process termination mailbox, which receives a process termination message for the creating process when a subprocess or detached process is deleted. Process termination mailboxes are discussed in the VAX/VMS System Services Reference Manual.

Mailboxes are either temporary or permanent. Table 3-3 contrasts the two types.

Table 3-3
Temporary versus Permanent Mailboxes

| Temporary | Permanent |
|---|---|
| 1. TMPMBX user privilege required to create | 1. PRMMBX user privilege required to create |
| 2. Creating process's buffered I/O byte count (BYTLM) quota is reduced (see Section 3.2.1) | 2. No process quotas affected |
| 3. Logical name entered in group logical name table | 3. Logical name entered in system logical name table |
| 4. Automatically deleted when no more channels are assigned to it | 4. Must be explicitly marked for deletion with the Delete Mailbox ($DELMBX) service |

Chapter 5 discusses mailboxes in shared (multiport) memory. The chapter on the mailbox driver in the VAX/VMS I/O User's Guide contains information on the use of mailboxes and a programming example.

### 3.2.1 Creating a Mailbox

The Create Mailbox and Assign Channel system service creates a mailbox or, if the specified mailbox already exists, assigns a channel to it. This service has the following general formats:

**MACRO Format**

```
$CREMBX [prmflg],chan,[maxmsg],[bufquo],[promsk],
        [acmode],[lognam]
```

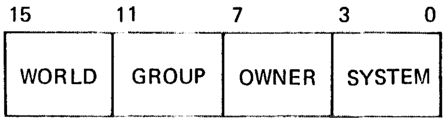
**High-Level Language Format**

```
SYS$CREMBX([prmflg],chan,[maxmsg],[bufquo],[promsk],
        [acmode],[lognam])
```

The PRMFLG argument determines whether the mailbox is temporary (the default) or permanent (value of 1). If the mailbox is temporary, the process's buffered I/O byte count (BYTLM) quota is reduced by the sum of the following until the mailbox is deleted:

- The number of bytes of system dynamic memory that can be used to buffer messages sent to the mailbox
- The size of the mailbox unit control block

The PROMSK argument allows you to restrict access to the mailbox by setting specific bits in a protection mask. This mask contains four 4-bit fields:



The bits are read from right to left in each field and indicate, when they are set, that read, write, execute, and delete access (in that order) are denied to the particular category of user. Only read and write access, however, are meaningful for mailbox protection. The default setting of 0 (all bits cleared) indicates that all users have read and write access to the mailbox.

The ACMODE argument allows a process executing at a more privileged access mode to associate a less privileged access mode with the channel assigned to the mailbox. (Kernel mode is the highest; user mode is the lowest.) The access modes and their corresponding values are listed below. The symbolic names for the values are defined by the $PSLDEF macro.

| **Access Mode** | **Value** | **Symbolic Name** |
|---|---|---|
| Kernel | 0 | PSL$C_KERNEL |
| Executive | 1 | PSL$C_EXEC |
| Supervisor | 2 | PSL$C_SUPER |
| User | 3 | PSL$C_USER |

Any ACMODE value you specify is maximized with your current access mode; that is, the channel is associated with the less privileged of the specified mode and your current mode.

The LOGNAM argument allows you to specify the logical name associated with the mailbox. Processes using a mailbox must specify the same logical name to identify that mailbox. When the mailbox is created, the logical name is entered in the group logical name table if the mailbox is temporary and in the system logical name table if the mailbox is permanent.

### 3.2.2 Other Mailbox Services

To use an existing mailbox, your process must assign it an I/O channel using the Create Mailbox system service or the Assign I/O Channel system service. (A high-level language program, however, need only issue an OPEN statement specifying the logical name of the mailbox.) The Assign I/O Channel system service has the following general formats:

**MACRO Format**

```
$ASSIGN devnam,chan,[acmode],[mbxnam]
```

**High-Level Language Format**

```
SYS$ASSIGN(devnam,chan,[acmode],[mbxnam])
```

The DEVNAM argument must specify the mailbox logical name. The ACMODE argument has the same meaning as in the Create Mailbox service. The VAX/VMS System Services Reference Manual describes the Assign I/O Channel system service in detail.

To delete a permanent mailbox, you must mark it for deletion using the Delete Mailbox ($DELMBX) system service. Actual deletion occurs, however, when all processes have deassigned the I/O channels connecting them to the mailbox or closed the file in a high-level language program. To deassign the I/O channel, use the Deassign I/O Channel ($DASSGN) system service.

### 3.2.3 Example Using a Mailbox

Figure 3-1 is a simple illustration of cooperating processes using a mailbox.

```
        PROGRAM MASTERPROC
        INTEGER*4 SYS$CREMBX,SYS$CREPRC,STATUS,CHAN

C-- Create a mailbox and call it      BOX'

     ❶  STATUS = SYS$CREMBX(,CHAN,,,,,'MAILBOX')
        IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

C-- Create a subprocess running program 'SUBPROC' and assign its input to be
C-- the mailbox and its output to be our terminal

     ❷  STATUS = SYS$CREPRC(,'SUBPROC','MAILBOX','TTD6:',,,,,%VAL(2),,,)
        IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

C-- Send the subprocess a message (in this case the number 12345)

     ❸  OPEN(UNIT=1,NAME='MAILBOX',STATUS='NEW')
        WRITE(1,*) 12345
        END


        PROGRAM SUBPROC

C-- Read the message from the mailbox and, in this case, just display it

     ❹  ACCEPT *,MESSAGE
        TYPE 10,MESSAGE
10      FORMAT(' The message was: ',I5)
        END
```

Figure 3-1 Using a Mailbox to Communicate

Notes on Figure 3-1:

❶ One process creates a mailbox.

❷ The process creates a subprocess.

❸ The creating process writes a message to the mailbox.

❹ The subprocess reads the message.

## 3.3 ASYNCHRONOUS SYSTEM TRAP SERVICE ROUTINES

An asynchronous system trap (AST) is a software-simulated interrupt used for event notification within a process. An AST service routine is a user-written routine that receives control when an AST is "delivered" after being queued to the process. The AST is delivered to the process (that is, interrupts the process execution flow) as soon as no higher-priority process is executable, unless specific conditions temporarily prevent it from being delivered (see Section 3.3.2). When the AST service routine completes, the current image continues executing from the point at which it was interrupted. ASTs are thus a mechanism to allow asynchronous operations.

### 3.3.1 **System Services with AST Service Routine Arguments**

Several system services allow you to specify an AST service routine to be executed when the requested operation is completed. The call to the service initiates the request, and an AST is queued to the process when the request is completed. These services are as follows:

- Queue I/O Request ($QIO)
- Update Section File on Disk ($UPDSEC)
- Get Job/Process Information ($GETJPI)

The Set Timer ($SETIMR) system service allows you to specify (1) an absolute or delta time for an AST to be queued to the process, and (2) the address of an AST service routine.

The Set Power Recovery AST ($SETPRA) system service specifies the address of an AST service routine to receive control after a power recovery is detected.

The Declare AST ($DCLAST) system service allows a process to queue an AST for itself at the same or a less privileged access mode and to specify an AST service routine. This service is particularly useful for testing an AST service routine and for initiating actions that must be performed in an AST service routine.

The VAX/VMS System Services Reference Manual contains a chapter on AST services, including a discussion on writing an AST service routine.

### 3.3.2 **Access Modes and AST Delivery**

ASTs are queued for a process by access mode. An AST for a more privileged access mode always takes precedence over one for a less privileged access mode; that is, an AST will interrupt any AST service routine executing at a less privileged mode. Normally, AST service routines that you specify execute at user access mode; however, the process can receive ASTs from more privileged access modes (for example, a kernel-mode AST at I/O completion).

Figure 3-2 shows a program interrupted by a user-mode AST, and the user-mode AST service routine interrupted by a kernel-mode AST.

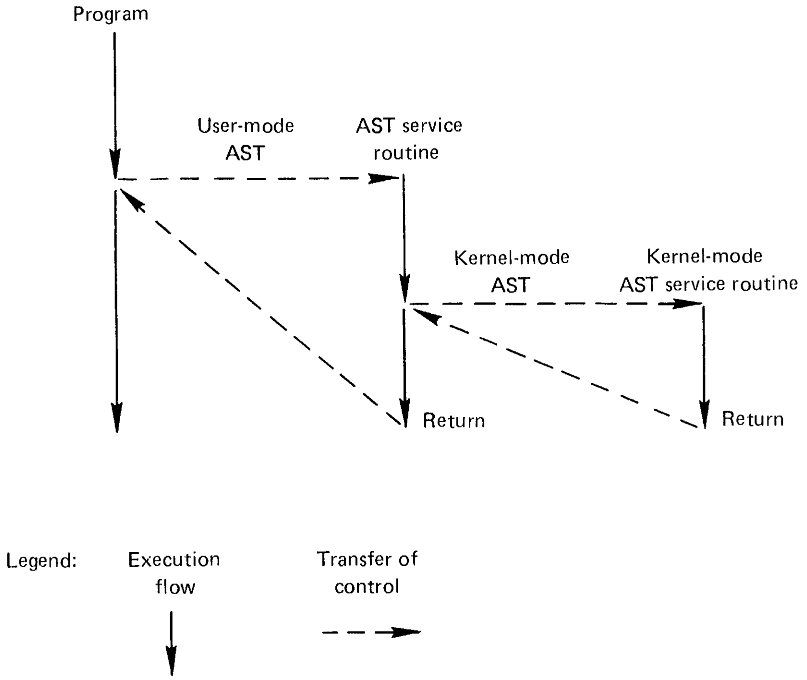


Figure 3-2 Access Modes and AST Delivery

An AST cannot be delivered to a process, however, while any of the following conditions are true:

- An AST service routine is currently executing at the same or a more privileged access mode.
- The current image is executing at a more privileged access mode than the mode for which the AST is declared.
- You have explicitly disabled AST delivery using the Set AST Enable ($SETAST) system service.
- The process is suspended (see Section 3.4).

## 3.4 HIBERNATION AND SUSPENSION

Hibernation and suspension are two synchronization mechanisms that allow a process to control when it or another process becomes active. Hibernation and suspension both temporarily halt the execution of a process; however, there are differences in how the mechanisms operate. Table 3-4 contrasts hibernation and suspension.

Table 3-4
Hibernation versus Suspension

| Hibernation | Suspension |
|---|---|
| 1. Process can cause only itself to hibernate | 1. Process can suspend itself or another process, depending on privilege |
| 2. Interruptible; ASTs can be delivered to the process | 2. Not interruptible; ASTs can be queued but not delivered |
| 3. Reversed by $WAKE system service | 3. Reversed by $RESUME system service |
| 4. Process can wake itself or be awakened by another process | 4. Process cannot cause itself to resume; another process must cause resumption |
| 5. Process can schedule wakeup at absolute time or fixed time interval ($SCHDWK service) | 5. Process cannot schedule resumption |
| 6. Hibernate/wake complete quickly and require little system overhead | 6. $SUSPEND service uses system dynamic memory; resumption takes longer |

The next two subsections provide coding examples illustrating two common uses of hibernate/wake:

- Activating a process as needed
- Activating a process at fixed intervals

Note that in both examples the process to be awakened is identified by process identification number rather than by process name. Either method is acceptable; however, when a process is identified by process identification number, the system service executes slightly faster, because it does not have to search the process name table.

### 3.4.1 Example 1: Wakeups as Needed

PROCESS1 creates PROCESS2 as a subprocess or detached process, but wants the created process to run only when certain events occur or certain conditions are true. Therefore, PROCESS1 sets bit 5 in the STSFLG argument to the Create Process system service call, causing PROCESS2 to hibernate immediately after it is created. PROCESS2 is activated only when PROCESS1 so requests, and PROCESS2 returns to hibernation immediately after it does whatever the specific application requires (for example, writing information to a mailbox used by both processes).

**PROCESS 1   Wakes PROCESS2 whenever necessary**

```
PROCESS2_ID:    .BLKL   1       ;RECEIVE ID OF CREATED PROCESS
PROCESS2_NAME:  .ASCID  /PROCESS2/ ;NAME OF CREATED PROCESS
          .
          .
          .
        $CREPRC_S  PIDADR=PROCESS2_ID,-         ;CREATE PROCESS2
                   PCRNAM=PROCESS2_NAME,-       ;SPECIFY NAME
                   STSFLG=#^B10000,-    ;PROCESS2 STARTS IN HIBERNATION
                      .
                      .                 ;(OTHER ARGUMENTS, AS NEEDED)
                      .
        BSBW    ERROR              ;BRANCH TO ERROR-CHECKING ROUTINE
          .
          .
          .
        $WAKE_S  PIDADR=PROCESS2_ID     ;WAKE PROCESS2
        BSBW    ERROR              ;BRANCH TO ERROR-CHECKING ROUTINE
          .
          .
          .
        $WAKE_S PIDADR=PROCESS2_ID ;WAKE PROCESS2
        BSBW    ERROR              ;BRANCH TO ERROR-CHECKING ROUTINE
          .
          .
          .
```

**PROCESS2   Awakens, performs functions, then goes back to sleep**

```
        .ENTRY  START,0            ;IMAGE ENTRY POINT & MASK
          .
          .                        ;(PERFORM FUNCTIONS)
          .
        RET                        ;BACK TO HIBERNATION
```

### 3.4.2 Example 2: Wakeups at Fixed Intervals

PROCESS1, a process with a priority in the timesharing range, creates PROCESS2 as a subprocess or detached process with a real-time base priority. PROCESS2 will run only at a fixed interval, in this case every hour, although its priority helps to ensure that when it does run it will run without interruption.

PROCESS2 hibernates immediately after it is created. PROCESS1 used the Schedule Wakeup ($SCHDWK) system service to schedule a wakeup for PROCESS2 in one hour (DAYTIM argument) and every hour thereafter (REPTIM argument). When PROCESS2 is activated, it performs its tasks and returns to a state of hibernation.

**PROCESS1 Process with timesharing priority**

```
PROCESS2_ID:     .BLKL   1        ;RECEIVE ID OF CREATED PROCESS
PROCESS2_NAME:   .ASCID  /PROCESS2/ ;NAME OF CREATED PROCESS
A1HOUR: .ASCID  /0 01:00:00.00/ ;ONE HOUR (DELTA TIME) IN ASCII
B1HOUR: .BLKQ  1                ;QUADWORD TO HOLD BINARY TIME VALUE
          .
          .
          .
        $CREPRC_S  PIDADR=PROCESS2_ID,...-       ;CREATE PROCESS2
                   ,PCRNAM=PROCESS2_NAME,-
                   BASPRI=#17,...                ;REAL-TIME PRIORITY
        BSBW    ERROR                   ;BRANCH TO ERROR-CHECKING ROUTINE
        $BINTIM_S  TIMBUF=A1HOUR,-      ;CONVERT TIME TO BINARY
                   TIMADR=B1HOUR
        BSBW    ERROR                   ;BRANCH TO ERROR-CHECKING ROUTINE
        $SCHDWK_S  PIDADR=PROCESS2_ID,- ;SCHEDULE WAKEUP FOR PROCESS2
                   DAYTIM=B1HOUR,-      ;   IN ONE HOUR,
                   REPTIM=B1HOUR        ;   AND EVERY HOUR THEREAFTER
        BSBW    ERROR                   ;BRANCH TO ERROR-CHECKING ROUTINE
          .
          .                             ;(CONTINUE PROGRAM EXECUTION)
          .
```

**PROCESS2 High priority real-time process**

```
          .
          .
          .
        .ENTRY  START,0         ;IMAGE ENTRY POINT & MASK
SLEEP:  $HIBER_S                ;SLEEP TILL NEXT SCHEDULED WAKEUP
        BSBW    ERROR           ;BRANCH TO ERROR-CHECKING ROUTINE
          .
          .                     ;(PERFORM HIGH-PRIORITY TASKS)
          .
        BRW     SLEEP           ;BACK TO SLEEP (FOR ONE HOUR)
```

A specific application of this example might involve a routine that needs to run periodically to gather and process status information. The routine might run for only a very short time, for example, a few seconds every hour. To prevent the routine from being interrupted, you can assign its process a real-time base priority and use any of the other methods discussed in Chapter 2.

## 3.5 GLOBAL SECTIONS

A global section is an area of memory containing data or code that can be shared by cooperating processes. One process "creates" the section; subsequent processes establish their right to use the section by "mapping" to it. The data or code in the section can be from a disk file (disk file section) or in physical memory or I/O space (page frame section). This section discusses disk file sections. Physical page frame sections are treated in Chapter 4 in the discussion of connecting to an interrupt vector.

In many real-time applications, such as data acquisition or industrial process-control, response time is so critical that control variables and data readings must remain in memory. Frequently, many different processes must use this data simultaneously. Global sections provide a convenient mechanism for fast access to the data and for the rapid passing of data from one process to another.

Global sections can be temporary or permanent. Temporary sections are deleted when no processes are mapped to them, but permanent sections must first be explicitly marked for deletion with the Delete Global Section ($DGBLSC) system service. Most global sections that you create from within your programs should be temporary, so that the system resources associated with the section can be freed as soon as they are no longer needed. Temporary global sections in real-time applications usually contain data rather than code. Permanent global sections, on the other hand, usually contain routines common to several programs. In fact, most of the permanent global sections in the system are shareable images installed by the system manager as known images. (Shareable images are discussed in Section 3.6. The INSTALL utility is explained in the VAX/VMS System Manager's Guide.)

VAX-11 Record Management Services (VAX-11 RMS), with its file-sharing capabilities, provides an alternative to global sections in some cases as a mechanism for sharing disk file data. Each method has its advantages; however, global sections provide the faster access that many real-time applications require. Table 3-5 shows the trade-offs involved in choosing between a global section and VAX-11 RMS for sharing disk file data.

Table 3-5
Global Sections versus VAX-11 RMS

| Global Sections | VAX-11 RMS |
|---|---|
| 1. Faster access to data | 1. Access to data slowed by file-system overhead |
| 2. More programming effort required; user must define and keep track of service arguments and other data | 2. Programming simplified by VAX-11 RMS or high-level language macros; most internal operations and data structures transparent to the user |
| 3. Greater burden on the user to protect data and synchronize access | 3. Automatic file protection and synchronization of access, based on parameters supplied by user |
| 4. Especially suited for small files | 4. Especially suited for large files |

Chapter 5 discusses global sections in shared (multiport) memory.

### 3.5.1 Creating and Mapping a Global Section

The Create and Map Section ($CRMPSC) system service creates a section or maps to an existing section. The VAX/VMS System Services Reference Manual has a detailed description of this service and a lengthy discussion of sections in general. The present manual gives only the general format for calling the service and discusses a few arguments especially significant to real-time users.

The Create and Map Section system service has the following general formats:

**MACRO Format**

```
     $CRMPSC   [inadr],[retadr],[acmode],[flags],[gsdnam],[ident]
               ,[relpag],[chan],[pagcnt],[vbn],[prot],[pfc]
```

**High-Level Language Format**

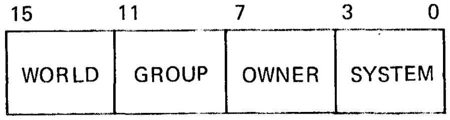
```
     SYS$CRMPSC([inadr],[retadr],[acmode],[flags],[gsdnam],[ident]
               ,[relpag],[chan],[pagcnt],[vbn],[prot],[pfc])
```

The FLAGS argument specifies a mask defining the section type and characteristics. This mask is the logical OR of the flag bits you want to set. (The $SECDEF macro defines the symbolic names for the flag bits in the mask.) To specify a global section, you must set the SEC$M_GBL flag bit. You can set additional flag bits as needed. The flag bit meanings and the default values they override are listed below.

| Flag | Meaning | Default Attribute |
|---|---|---|
| SEC$M_GBL | Global section | Private section |
| SEC$M_CRF | Pages are copy-on-reference | Pages are shared |
| SEC$M_DZRO | Pages are demand-zero pages | Pages are not zeroed when copied |
| SEC$M_EXPREG | Map into first available space | Map into range specified by INADR argument |
| SEC$M_WRT | Read/write section | Read-only section |
| SEC$M_PERM | Permanent | Temporary |
| SEC$M_PFNMAP | Physical page frame section | Disk file section |
| SEC$M_SYSGBL | System global section | Group global section |

The PROT argument specifies a numeric value representing the protection mask to be applied to the section. To deny read or write access to the section to one or more types of user, you must specify the appropriate protection mask. If you do not specify this argument, all users have read and write access to the section.

The protection mask has four 4-bit fields:



Bits are read from right to left in each field and indicate, when they are set, that read, write, execute, and delete access (in that order) are denied for that particular category of user. However, the following considerations apply to any protection mask you specify:

- Only read and write access are meaningful for section protection. Denying execute or delete access has no effect.

- For group global sections the "World" field has no effect, because only members of the creator's group are permitted to map to the section. The "World" field does apply, however, to system global sections.

For example, to allow the owner of a group global section to read and write to the section but allow other members of the group only to read the section (that is, to deny them write access), specify a protection mask of 0200 (hexadecimal).

### 3.5.2 Other Section-Related System Services

The following system services are often used with global sections:

- Map Global Section ($MGBLSC). Maps an existing global section.

- Update Section File on Disk ($UPDSEC). Writes the modified pages of a section back to the disk file. This system service is especially useful for periodically updating a data base that is being modified by multiple processes.

- Delete Virtual Address Space ($DELTVA). "Unmaps" a global section by deleting the process's virtual addresses into which the section was mapped.

- Delete Global Section ($DGBLSC). Marks a global section for deletion. Actual deletion occurs when no processes are mapped to the section.

## 3.6 SHAREABLE IMAGES

Shareable images can be used to share frequently used code or data among multiple processes. A shareable image might contain routines that are common to several programs. If a shareable image is installed in the system as a permanent global section (as is normally the case), other programs can share its contents by linking with it. The benefits of using shareable images include reductions in disk storage space, physical memory use, and system paging activity. The VAX-11 Linker Reference Manual explains the benefits and uses of shareable images in detail.

In the airline reservation example in Chapter 7, the reservation data base is a shareable image.

To use a shareable image effectively, you must create the shareable image and then permit other programs to use it.

To create a shareable image, you must perform the following steps:

1. Code the program containing the routine or data to be shared. Design this program to meet the needs of all other programs that will be using it (that is, all programs that will be linked to the shareable image). Follow the programming conventions discussed in the chapter on shareable images in the VAX-11 Linker Reference Manual.

2. Assemble or compile the program containing the shareable code or data. For example:

   ```
   $ MACRO SHCODE
   ```

   This command generates the object module SHCODE.OBJ in your default directory (assume that this is DB1:[SMITH] for this and the remaining steps).

3. Link the object module to produce a shareable image, using the /SHAREABLE command qualifier. For example:

   ```
   $ LINK/SHAREABLE SHCODE
   ```

   This command generates the shareable image SHCODE.EXE in your default directory.

To permit other programs to use the shareable image, you must perform the following steps:

1. Create a linker options file. Identify the shareable image to be used with the /SHAREABLE file qualifier. For example, create a file named A.OPT containing the following line:

   ```
   DB1:[SMITH]SHCODE/SHAREABLE
   ```

2. Link each program that will use the shareable image, identifying the linker options file with the /OPTIONS file qualifier. For example:

   ```
   $LINK PROGRAM1,A/OPTIONS
   ```

   This command generates an executable image named PROGRAM1 that is linked with the shareable image SHCODE.

To permit multiple processes to use the same copy of the shareable image, install it as a known image, using the INSTALL utility. (The VAX/VMS System Manager's Guide explains the INSTALL utility.) It is recommended that you copy the shareable image file to the directory identified by the logical name SYS$SHARE (which by default is [SYSLIB] on the system disk), and then run INSTALL:

```
$ RUN SYS$SYSTEM:INSTALL
INSTALL>SYS$SHARE:SHCODE/OPEN/SHARED
```

The example above designates the shareable image as a permanent global section, that is, a permanently open section potentially available to all users of the system.

Note that the VAX/VMS image activator assumes that shareable images linked with the executable image being run are located in SYS$SHARE. To have the image activator look for a shareable image in a different location, define the shareable image file name as a logical name with the file specification as the equivalence name before running the executable image. For example:

```
$ DEFINE SHCODE DB1:[SMITH]SHCODE
```

# CHAPTER 4

# PERFORMING I/O OPERATIONS

A real-time VAX/VMS process can use the VAX/VMS I/O system to perform I/O operations, or it can bypass most of the I/O system by manipulating device registers and responding to device interrupts directly. Before you can optimize I/O operations for a real-time application, however, you must understand the components that form the VAX/VMS I/O system and how they interact.

## 4.1 OVERVIEW OF THE VAX/VMS I/O SYSTEM

The VAX/VMS I/O system has the following major components:

- The Queue I/O Request system service
- Device drivers
- Ancillary control processes (ACPs)
- The I/O posting routine

The following subsections describe the main functions of these components.

### 4.1.1 Queue I/O Request System Service

Every I/O request issued by a process under VAX/VMS results directly or indirectly in the invocation of the Queue I/O Request system service. For example, both a FORTRAN READ statement and a VAX-11 RMS $GET request from a VAX-11 MACRO program cause the Queue I/O Request system service to be called.

You can call the Queue I/O Request system service specifying one of three types of function code: physical, logical, or virtual. The service validates the device-independent portions of the I/O request. The device driver or ancillary control process (ACP) performs any necessary validation of the device-dependent portions of the I/O request.

The VAX/VMS I/O User's Guide lists the valid function codes for each device driver or ACP and provides guidelines for choosing among function codes when alternatives are available.

### 4.1.2 Ancillary Control Processes

An ancillary control process (ACP) is a VAX/VMS process that performs I/O-related functions associated with file structures and protocol, rather than functions related to the actual transfer of data. VAX/VMS supplies at least five ACPs:

- Two or more ACPs for Files-11 structured disk devices
- One ACP for ANSI magnetic tapes
- NETACP for network functions
- REMACP for remote terminal I/O functions

The use of ACPs is normally transparent to your programs. VAX-11 RMS issues the necessary Queue I/O Request system services for virtual functions on your behalf. You can, however, issue Queue I/O Request system service calls directly for Files-11 disk and magnetic tape ACPs to request such functions as the following:

- File creation
- File access
- Reading and writing of virtual blocks
- File deletion

The VAX/VMS I/O User's Guide describes the use of ACPs by user processes.

When a user process or VAX-11 RMS issues a Queue I/O Request system service for an ACP function, the Queue I/O Request system service passes the request to the appropriate ACP. The ACP processes the request (if necessary), converts the function from virtual to logical (if necessary), and queues the request to the appropriate device driver. The driver performs the transfer, as described in Section 4.1.3.

### 4.1.3 Device Drivers

Device drivers are responsible for taking the information that the Queue I/O Request system service provides about an I/O request and performing the I/O operation. To accomplish these tasks, a driver contains the following main routines:

- Device activation routine
- Interrupt service routine
- I/O completion routine

Drivers also contain other routines to handle request validation and such contingencies as power failure and device timeout, as described in the VAX/VMS Guide to Writing a Device Driver.

The device activation routine obtains the device controller resources needed to perform the transfer (for example, the controller data channel), sets up device registers in I/O space, and initiates the transfer. Once the transfer is initiated, the device activation routine issues a wait request that temporarily suspends the device driver.

When the transfer is complete, the device requests an interrupt and the system activates the driver's interrupt service routine to handle the interrupt. (Section 4.6 discusses interrupt handling.) In addition to handling the interrupt, the interrupt service routine may program the device for another transfer or may activate the I/O completion routine in the driver to perform device-dependent I/O completion. The driver's I/O completion routine, in turn, passes control to the VAX/VMS I/O posting routine.

### 4.1.4 I/O Posting Routine

Once the device driver has finished the device-dependent portions of the I/O request, it calls the I/O posting routine. I/O posting consists of completing the device-independent portions of the I/O request, setting a designated event flag (flag 0 by default), and queuing a kernel mode AST for the process that initiated the I/O request.

The next time the system schedules this process for execution, the kernel mode AST routine executes. This routine completes the I/O request by performing the following functions:

- If requested, writes the status of the I/O request into a user-specified I/O status block.

- If requested, queues an AST at the access mode of the Queue I/O request for the process to execute a user-specified routine.

- For read requests that were buffered in system space, copies the data from system space into the user's buffer. Device drivers determine whether the data is read directly into the user buffer (direct I/O) or buffered first in system space (buffered I/O).

The driver's I/O posting routine has a lower priority than the driver's start I/O routine. Therefore, if a new I/O request is queued for the device before the existing I/O request is completed, the new I/O is started. This method of operation keeps the device as busy as possible.

## 4.2 USER INTERFACE TO THE I/O SYSTEM

The design of the VAX/VMS I/O system allows user-written programs to interface with the system at a number of levels:

- VAX-11 Common Run-Time Procedure Library routines

- VAX-11 Record Management Services (VAX-11 RMS)

- Queue I/O Request system service for a device or ACP function

- Connecting to a device interrupt vector

In addition, users can write device drivers to support devices not supported by VAX/VMS and incorporate those devices into the system.

Programs written in VAX-11 MACRO can interface with the I/O system by using VAX-11 RMS, by using the Queue I/O Request system service, or by mapping to I/O space and connecting to a device interrupt vector. Programs written in a high-level language can interface with the I/O system using the same methods as a VAX-11 MACRO program, or they can issue the I/O statements specific to that language. In the latter case, the program interfaces with the I/O system by means of the VAX-11 Common Run-Time Procedure Library.

The following steps occur when a high-level language program, in this case VAX-11 FORTRAN, issues a read request under VAX/VMS:

- When the program executes, the read statement results in a call to the Run-Time Library read procedure to initiate the read operation. To initiate the read, the procedure issues a VAX-11 RMS $GET request.

- VAX-11 RMS gains control and, in turn, issues the appropriate Queue I/O Request system service.

- The Queue I/O Request system service processes the request (as described in Section 4.1.1) and queues it to the driver or ACP.

- Once the driver activates the device and completes the I/O operation, it calls the VAX/VMS I/O posting routine.

- The VAX/VMS I/O posting routine then performs device-independent I/O completion, returns status to the user program, and, if requested, queues an AST or sets an event flag.

A user program can interface with the I/O system at one of several levels, depending on its requirements. At each level, the user program makes trade offs between ease of use and execution speed. As a general rule, the closer to the VAX/VMS executive that a user program interfaces, the less overhead is involved in the I/O operation. This manual focuses on the following lower levels of interface: the Queue I/O Request system service, the Create and Map Section system service, and the connect-to-interrupt capability.

### 4.2.1 VAX-11 RMS Features of Interest to Real-Time Users

VAX-11 Record Management Services has several features that may permit certain applications to take advantage of VAX-11 RMS and still meet their throughput and response requirements. Listed below are descriptions of these features, with the VAX-11 RMS mechanism associated with each feature. Complete descriptions of the features and mechanisms are given in the VAX-11 Record Management Services Reference Manual.

| **Mechanism** | **Feature** |
|---|---|
| $FAB ALQ=quantity | Preallocation of enough blocks to hold the entire file. Avoids time-consuming file extensions and ACP window turns; prevents discontiguous file extensions. |
| $FAB FAC=BIO | Block I/O (for $PUT operations). Faster I/O because no RMS buffer is used. |
| $FAB FOP=CTG | Contiguous files. Faster access, especially for random access and/or files with many segments. |
| $RAB MBF=buffers | Multibuffering. Improves throughput. |
| $RAB ROP=RAH<br>$RAB ROP=WBH | Read-ahead and write-behind. Improve throughput (done by default by certain high-level language compilers). |
| $RAB MBC=blocks | Multiblock I/O. Reduces number of disk accesses for record operations. |

## 4.3 USING THE QUEUE I/O REQUEST SYSTEM SERVICE

The Queue I/O Request ($QIO) system service gives programmers in any supported language a low-level, flexible interface with the VAX/VMS I/O system. You must first assign an I/O channel to the device using the Assign I/O Channel ($ASSIGN) system service. Your call to the Queue I/O Request system service must specify this channel and a function code identifying the operation to be performed. The optional arguments to the Queue I/O Request service allow you to do the following:

- Perform asynchronous ($QIO form) or synchronous ($QIOW form) I/O
- Set an event flag at I/O completion (EFN argument)
- Receive the final completion status (IOSB argument)
- Specify an AST service routine (ASTADR argument) to be executed when the I/O completes and pass a parameter (ASTPRM argument) to that routine
- Specify function-specific or device-specific parameters (P1, P2, etc.)

There are two forms of this service: Queue I/O Request ($QIO) and Queue I/O Request and Wait for Event Flag ($QIOW). The $QIO form returns control to the program immediately after queuing the I/O request and without waiting for the I/O to be completed; this form allows your program to perform asynchronous I/O. The $QIOW form waits until the I/O is completed before returning control to your program. (The $INPUT and $OUTPUT macros are special forms of $QIOW.)

The Queue I/O Request system service has the following general formats:

**MACRO Format**

```
     $QIO[W]     [efn],chan,func,[iosb],[astadr],[astprm],
                 [p1],[p2],[p3],[p4],[p5],[p6]
```

**High-Level Language Format**

```
     SYS$QIO[W]([efn],chan,func,[iosb],[astadr],[astprm],
                 [p1],[p2],[p3],[p4],[p5],[p6])
```

The VAX/VMS System Services Reference Manual has additional general information on this system service and some examples of its use. The VAX/VMS I/O User's Guide has specific information and examples of this system service for each of the device drivers it discusses.

## 4.4 INTERRUPT-GENERATED I/O

A process with suitable privileges can connect to a device interrupt vector and/or map the processor's I/O space into process virtual address space. Connecting to a device interrupt vector allows your process to respond to interrupts from the device with minimal overhead. Mapping processor I/O space allows your process to access device registers from the main program or from an AST service routine.

A process normally uses these features for devices that do not have VAX/VMS drivers. These devices must not be direct memory access (DMA) devices, and they must be attached to the UNIBUS. Examples of such devices are the AD11-K the DR11-B, and the KW11-P.

You can use the Queue I/O Request ($QIO) system service with an appropriate function code to connect to a device interrupt vector and to specify a user-supplied routine, called an interrupt service routine (ISR), that VAX/VMS executes when the designated device interrupts. Connecting to a device interrupt vector allows you to do the following:

- Respond to an interrupt within a very short time
- Preempt other system processing to handle a real-time event, for example, a clock interrupt
- Buffer data from a device in real time and return the data to the process at a later time
- Set an event flag or queue an AST to your process after receiving the interrupt

The effect of user-written interrupt service routines is to allow you to perform some of the functions normally done by a device driver, but without requiring that you write a full device driver and without requiring that the routine be loaded into the VAX/VMS operating system (device drivers are part of VAX/VMS).

If you must access device registers from user mode (that is, from the main program or a user-mode AST service routine), you must use the Create and Map Section ($CRMPSC) system service to map I/O space, specifying page frame number (PFN) mapping. The service creates a global or private section that maps the specified I/O pages into your process's virtual address space. The process can then gain access to I/O space using virtual addresses.

You do not need to map I/O space to access device registers from any of the following routines specified in the $QIO call connecting to an interrupt vector: device initialization routine, start I/O routine, interrupt service routine, and cancel I/O routine. These routines execute in system space and thus can access UNIBUS I/O space, which is mapped as part of system space.

The sections that follow explain how to map the VAX-11 processor's I/O space and how to connect to a device interrupt vector.

## 4.5 MAPPING I/O SPACE

On a VAX-11/780 processor, I/O space is assigned physical address locations of 20000000 (hexadecimal) and higher. I/O space contains device registers that a driver or user process can read and write to control a device. Each device controller has an associated control/status register in I/O space. Device registers for each device are located at an offset from the device's control/status register (CSR).

The $IO780DEF macro defines the following symbols describing the layout of VAX-11/780 I/O space:

| **Symbol** | **Meaning** | **Hexadecimal value** |
|---|---|---|
| IO780$AL_IOBASE | Start of I/O space | 20000000 |
| IO780$AL_UBOSP | Start of address space for first UNIBUS | 20100000 |

These symbols are contained in SYS$LIBRARY:LIB.MLB.

The number of registers and their locations vary for different devices. The PDP-11 Peripherals Handbook provides the necessary information for devices supplied by DIGITAL. The VAX-11/780 Hardware Handbook contains information about the layout of I/O space.

On a VAX-11 processor, the address of a physical memory location has the format illustrated in Figure 4-1.

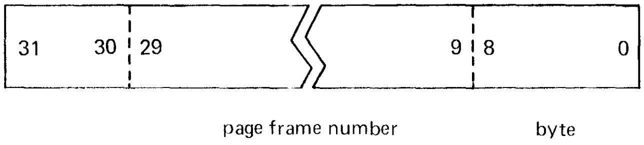


Figure 4-1 Physical Address

The page frame number (bits 9 through 29) specifies the number of a physical page in memory. Bit 29 is clear to indicate a physical memory address and set to indicate an address in I/O space. Bits 0 through 8 specify the byte address within the page.

For a process to gain access to I/O space or to any page of physical memory, it must map that page into its virtual address space. When your VAX/VMS process maps a page by specifying its page frame number, it completely bypasses VAX/VMS memory management and creates its own window to the page. As a result, the protection functions that VAX/VMS normally performs are not performed for mapping by page frame number:

- No checks are performed to ensure that no other VAX/VMS processes are mapped to the page and modifying it.
- No reference count is maintained. A process can delete a global section mapped by page frame numbers when other processes are still using it; this is not the case when VAX/VMS performs the mapping.

Modifying pages mapped by page frame numbers can have unpredictable results and can adversely affect system operation, especially if the operating system is also using these pages. Because of the unprotected nature of mapping by page frame numbers, you must have the PFNMAP user privilege to use this capability.

### 4.5.1 Page Frame Number (PFN) Mapping

When used for mapping by page frame number, the Create and Map Section system service designates the specified page(s) as a global or private section and maps the section into the requesting process's virtual address space. The pages can be located anywhere in the VAX-11 processor's local memory, or in MA780 memory (if a multiport memory unit is connected to the system), or in I/O space.

The format and conventions for mapping by page frame number (that is, mapping a physical page frame section) are similar to those for mapping a disk file section. The Create and Map Section system service has the following general formats:

**MACRO Format**

```
     $CRMPSC    [inadr] ,[retadr] ,[acmode] ,[flags] ,[gsdnam] ,[ident]
                ,[relpag] ,[chan] ,[pagcnt] ,[vbn] ,[prot] ,[pfc]
```

**High-Level Language Format**

```
     SYS$CRMPSC([inadr] ,[retadr] ,[acmode] ,[flags] ,[gsdnam] ,[ident]
                ,[relpag] ,[chan] ,[pagcnt] ,[vbn] ,[prot] ,[pfc])
```

The RELPAG, CHAN, and PFC arguments are not applicable in mapping by page frame number. The INADR, RETADR, ACMODE, GSDNAM, IDENT, and PROT arguments have the same functions regardless of whether you specify page frame number mapping; these arguments are described in the VAX/VMS System Services Reference Manual.

The following arguments are affected by PFN mapping:

flags

Mask defining the section type and characteristics. This mask is the logical OR of the flag bits you want to set. The $SECDEF macro defines symbolic names for the flag bits in the mask.

The SEC$M_PFNMAP flag bit must be set to indicate mapping by page frame number. The SEC$M_PFNMAP flag setting identifies the memory for the section as starting at the page frame number specified in the VBN argument and extending for the number of pages specified in the PAGCNT argument.

If appropriate, the following flags can also be set:

| **Flag** | **Meaning** | **Default** |
|---|---|---|
| SEC$_GBL | Global section | Private section |
| SEC$M_WRT | Read/write section | Read-only section |
| SEC$M_PERM | Permanent section | Temporary section |
| SEC$M_SYSGBL | System global section | Group global section |
| SEC$M_EXPREG | Expand the process's virtual address space as needed to contain the section. | Map into range specified by INADR argument |

Neither the SEC$M_CRF (copy-on-reference) nor the SEC$M_DZRO (demand-zero) bit can be set when mapping by page frame number.

The VAX/VMS System Services Reference Manual provides additional information about the use of the flag settings.

pagcnt

Number of pages in the section; the value of this argument must not be zero.

vbn

Page frame number of the first page to be mapped (as opposed to this argument's normal usage identifying the starting virtual block number within a disk file). When you are mapping more than one page with a single Create and Map Section system service request, the pages are physically contiguous starting with the specified page.

**Notes**

1. An error in mapping UNIBUS I/O space or a reference to a nonexistent UNIBUS address causes a UNIBUS adaptor error. However, this error does not cause a system failure. Rather, an entry is made in the system error log file and the user program continues executing (probably with erroneous results). The process is not notified of the UNIBUS adapter error.

2. If a power failure occurs on the UNIBUS, the system continues to run. However, if a user process accesses UNIBUS I/O space from user mode during a UNIBUS power failure, the process receives a machine check exception. To handle this condition, the process must have a condition handler to deal with machine check exceptions. The VAX/VMS System Services Reference Manual discusses condition handlers in detail.

3. During recovery from a UNIBUS adaptor power failure, the processor spends a considerable amount of time (perhaps 10 to 60 milliseconds) at interrupt priority level (IPL) 31. This action blocks user processes from executing during the recovery.

### 4.5.2 Programming Conventions for Addressing Device Registers

Once you have mapped to I/O space, you can read data from a device data buffer register or enable interrupts by setting a bit in a control/status register, because the device registers are now addressable as part of your process's virtual memory. The UNIBUS adapter performs the actual mapping of VAX-11 virtual addresses to 18-bit UNIBUS addresses that correspond to device registers.

Because UNIBUS devices are one word (16 bits) long, all instructions referring to these registers must be word-context instructions (for example, BISW, MOVW, and ADDW3), unless the register is byte addressable. Instructions referring to byte-addressable registers should be byte-context instructions, such as BISB and MOVB. Unaligned references and references using a length attribute other than the length of the register may produce unpredictable results; for example, a byte reference to a word-addressable register does not necessarily respond by supplying or modifying the byte addressed. A longword reference to a UNIBUS location causes a machine check.

Instructions that use a UNIBUS device register as a source operand must not be interruptible instructions. In some cases when a device register is being copied, interrupting and restarting an instruction may cause a character to be lost. To guarantee a noninterruptible sequence, use only the instructions listed in Appendix C of the VAX-11/780 Hardware Handbook, and do not use autoincrement deferred addressing mode or any of the displacement deferred addressing modes. You should always store the address of a device control register in a general register and then gain access to the device indirectly through the general register.

The example below defines symbolic word offsets for each device register and gains access to them using displacement mode addressing from R4.

```
;
; Device register offsets
;

LP_CSR  = 0                             ; CSR offset
LP_DBR  = 2                             ; Buffer address offset
    .
    .
    .
    MOVL    CSR_VA,R4                   ; Get CSR address
    .
    .
    .
    TSTW    LP_CSR(R4)                  ; Is printer online?
```

The following restrictions also apply to instructions addressing device registers:

- Operand types of floating, double, field, queue, or quadword are not allowed, nor can the position, size, length, or base of an operand be from I/O space. For example, a field instruction cannot be used to test a bit in a device register.

- You cannot have more than one modify or write destination, and this modify or write destination must be the last operand.

- Instructions referring to I/O space must not cause an exception after the first I/O space reference. This restriction includes deferred references to I/O space.

## 4.6 CONNECTING TO AN INTERRUPT VECTOR

On a VAX-11 processor, peripheral devices have interrupt vectors associated with them. When a device interrupt occurs, the action taken by the processor depends on the interrupt priority level (IPL) associated with the device.

Connecting to an interrupt vector differs from the standard method of programming a peripheral device. Programming a peripheral device is normally a 3-step loop:

1. The device driver starts the device and enables interrupts from the device.
2. The device generates an interrupt.
3. The device driver fields the interrupt, collects status and data, and clears the interrupt condition.

Under the VAX/VMS operating system, a user program normally requests I/O by means of a Queue I/O Request ($QIO) system service call. A device driver, executing as part of the operating system, controls and responds to the device. The driver returns status and data to the requesting user process.

However, real-time application programmers can connect to an interrupt vector to control and respond to a device without writing a full VMS device driver, and without issuing $QIO calls for each device interaction. Instead, you issue a connect-to-interrupt $QIO call that specifies code to be executed to control the device, and a data area that the program and the device control code can share. You subsequently control and respond to the device without additional $QIO calls.

The timings involved in different system activities associated with connecting to an interrupt vector are as follows:

- The time between when the device generates an interrupt and when the process's interrupt service routine receives control depends upon the IPL of the processor at the time of the interrupt. If the processor is executing at an IPL below that of the device (as is the usual case), the interrupt service routine gains control within a few microseconds. However, if the processor is executing at an IPL above that of the device, the interrupt service routine does not gain control until the executing code lowers the IPL below the device IPL. (Section 4.6.1 discusses IPLs.)
- The time from the user interrupt service routine's exit to the execution of the AST routine specified in the $QIO call depends on the priority of the process and whether a context switch is required.

### 4.6.1 Interrupt Priority Levels

VAX-11 processors define 32 hardware interrupt priority levels. These interrupt priority levels establish the order in which peripheral devices, error condition reporting, and various components of VAX/VMS gain access to the processor; that is, interrupt priority levels are a synchronization mechanism. (Interrupt priority is not related to process priority, which is discussed in Section 1.6.) VAX/VMS and VAX-11 processors assign the interrupt priority levels (IPLs) as follows:

- User mode programs run at IPL 0; this is the lowest IPL.
- VAX/VMS routines and device driver processes request interrupts at IPLs 1 through 15. (Device drivers execute as fork processes under VAX/VMS, as described in the VAX/VMS Guide to Writing a Device Driver.)
- Peripheral devices generate interrupts at IPLs 16 through 19. UNIBUS peripherals generate interrupts of IPLs 20 through 23 (corresponding to UNIBUS BR levels 4 through 7).
- Processor error conditions and the system clock generate interrupts at IPLs 20 through 31.

Because of the way in which priority levels are assigned, device interrupts almost always receive immediate service from the processor and VAX/VMS.

A VAX-11 processor always executes the code associated with the highest IPL for which an interrupt has been requested. For example, if the processor is executing a driver process and a device requests an interrupt, the processor stops executing the driver, saves the driver's context for subsequent reactivation, and activates the interrupt service routine for the interrupting device. When that interrupt service routine terminates, VAX/VMS activates the code associated with the next lower IPL for which an interrupt has been requested. The routine activated can be either of the following:

- A routine that had already started execution but was interrupted by a higher level interrupt
- A routine for which an interrupt has been pending while the processor executed at a higher IPL but which had not been executed previously

### 4.6.2 Performing the Connect-To-Interrupt

Connecting to a device interrupt vector allows your program to receive notification of an interrupt from a designated device by any combination of the following means:

- By execution of a user-supplied interrupt service routine
- By the setting of an event flag
- By execution of an AST routine that is to gain control in process context

In addition, you can specify a cancel routine that is to be executed when the process disconnects from the interrupt vector or is deleted.

Before your program can run, the system manager must have done the following at system generation time:

- Specify the REALTIME_SPTS parameter to the SYSGEN utility, reserving system page table entries for use by real-time processes. These system page table entries are used to map process-specified buffers in system space (see the P1 argument description in Section 4.6.5). The REALTIME_SPTS parameter value must be greater than or equal to the number of pages in buffers specified by processes connected to interrupt vectors.

- Configure the real-time device by issuing a CONNECT command to the SYSGEN utility. This command names the device; its vector, register, and adapter addresses; and a skeletal driver (CONINTERR) for the device.

The CONNECT command to the SYSGEN utility is explained in the VAX/VMS System Manager's Guide.

At run time the process calls the $ASSIGN system service to associate a channel with the device. The process can also map the page in UNIBUS I/O space containing the device registers (see Section 4.5). To connect to the device interrupt vector, the process issues a $QIO call specifying the IO$_CONINTREAD or IO$_CONINTWRITE function code and as many of the following as are appropriate:

- An interrupt service routine to be executed when the device generates an interrupt

- A buffer containing code to be executed in system context and/or data (This buffer must be contiguous in the process's address space.)

- An AST service routine to execute and/or an event flag to be set after the interrupt service routine (if any) completes (If an AST service routine is specified, an AST parameter may also be specified.)

- A device initialization routine

- A start I/O routine

- A cancel I/O routine

A nonprivileged process (that is, lacking the CMKRNL privilege) can also connect to an interrupt vector, but it can only specify an AST service routine to be executed or an event flag to be set (or both) when an interrupt is generated. Section 4.6.5 explains the $QIO format for connecting to an interrupt vector.

### 4.6.3 The Connect-To-Interrupt Driver

The VAX/VMS connect-to-interrupt driver (CONINTERR) provides a driver interface to the system on behalf of the process. CONINTERR connects the process to the device by executing the following steps:

1. Validates the $QIO system service parameters, such as the process's access to the specified buffer, and the number of the optional event flag.

2. Locks the physical pages of the buffer into physical memory, and maps the pages using system page table entries allocated by the REALTIME_SPTS parameter to the SYSGEN utility.

3. Constructs argument lists and calling interfaces to the process-specified routines by storing values in the device's unit control block (UCB).

4. Allocates the specified number of AST control blocks to the process, and inserts each block in a queue in the device's UCB.

5. Transfers control to VAX/VMS to queue the connect to interrupt I/O packet to CONINTERR start I/O routine.

When the CONINTERR start I/O routine gains control, it passes control, by means of a user-specified JSB or CALLS interface, to the process-specified start-device routine. This routine usually initializes the device and may also start activity on the device.

When the device generates an interrupt, the interrupt service routine in CONINTERR gains control. This routine transfers control to the process-supplied interrupt service routine.

### 4.6.4 The Interrupt and AST Service Routines

The interrupt service routine that you specify, like those supplied by VAX/VMS, has the following characteristics:

- It is mapped in system space.

- It executes on the interrupt stack.

- It executes at the IPL of the device that requested the interrupt.

Because of these characteristics, the interrupt service routine executes as part of the VAX/VMS operating system rather than in the context of your user process. As part of the operating system, the interrupt service routine has access to system data bases not available to user processes. However, because an interrupt service routine has these capabilities and executes at a raised IPL, you must code it carefully to avoid disrupting the system. Section 4.6.9 discusses conventions for process-specified interrupt service routine.

The interrupt service routine that you specify usually performs one or more of the following steps:

1. Copies data from a device register

2. Writes to a device register to clear the interrupt condition

3. Restarts the device, or returns an offset, a byte count, or actual data as an AST parameter

4. Returns an interrupt status to the VAX/VMS connect-to-interrupt driver (CONINTERR)

Depending on the interrupt status, the CONINTERR interrupt service routine queues a fork process to run at a lower IPL. Then the interrupt service routine exits from the interrupt with an REI instruction. When the CONINTERR fork process gains control, it queues an AST or posts an event flag to the process (or both).

The AST service routine that you specify gains control in process context. This routine usually performs one or more of the following steps:

1. Reads or writes device registers if the process mapped I/O space (see Section 4.5).

2. Interprets data. Use caution, however, because any processing done by the AST service routine can be interrupted by a device interrupt, which might store more data or modify the buffer's contents.

3. Calls the Cancel I/O on Channel ($CANCEL) system service to disconnect the process from the interrupt.

### 4.6.5 Queue I/O Request System Service for Connect-To-Interrupt

The format of the Queue I/O Request ($QIO) system service to connect to an interrupt vector is given below. The explanation is limited to connecting to an interrupt vector. For a detailed description of the $QIO system service, see the VAX/VMS System Services Reference Manual.

**MACRO Format**

```
$QIO [efn] ,[chan] ,func ,[iosb] ,[astadr] ,[astprm]
                   ,[p1] ,[p2] ,[p3] ,[p4] ,[p5] ,[p6]
```

**High-Level Language Format**

```
SYS$QIO([efn] ,[chan] ,func ,[iosb] ,[astadr] ,[astprm]
                      ,[p1] ,[p2] ,[p3] ,[p4] ,[p5] ,[p6])
```

efn
iosb
astadr
astprm

These arguments apply to the $QIO system service completion, not to device interrupt actions. For an explanation of these arguments, see the $QIO service description in the VAX/VMS System Services Reference Manual.

func

Function code of IO$_CONINTREAD or IO$_CONINTWRITE. The IO$_CONINTWRITE function code allows locations in the buffer pointed to by the Pl argument to be modified; the IO$_CONINTREAD function code makes the buffer contents read-only.

pl

Address of a descriptor for the buffer containing code and/or data. The first longword records the number of bytes in the buffer; the second longword records the address of the buffer. (Note: The buffer size must not exceed 64K bytes.)

p2

Address of an entry point list. The list consists of four longwords that contain offsets into the buffer (specified in the Pl argument) of entry points of process-specified routines. These longwords and their contents are as follows:

| | |
|---|---|
| CIN$L_INIDEV | Offset to device initialization routine |
| CIN$L_START | Offset to start device routine |
| CIN$L_ISR | Offset to interrupt service routine |
| CIN$L_CANCEL | Offset to cancel I/O routine |

Note: Symbols starting with CIN$ are defined by the $CINDEF macro. The definitions are in the library SYS$LIBRARY:LIB.MLB.

p3

Longword containing flags and an optional event flag number specification. The low-order word contains the logical OR of flags describing options to the connect-to-interrupt facility. The flags and their meanings are as follows:

| | |
|---|---|
| CIN$M_EFN | Set event flag on interrupt |
| CIN$M_USECAL | Use CALL interface to process-specified routines (default is JSB interface) |
| CIN$M_REPEAT | Leave process connected to the interrupt vector until the connection is canceled |
| CIN$M_INIDEV | Process-specified device initialization routine is in the buffer specified in the P1 argument |
| CIN$M_START | Process-specified start I/O routine is in buffer |
| CIN$M_ISR | Process-specified interrupt service routine is in buffer |
| CIN$M_CANCEL | Process-specified cancel I/O routine is in buffer |

The high-order word specifies the number of the event flag to be set when an interrupt occurs. This number is expressed as an offset to CIN$V_EFNUM.

For example, to specify that your interrupt service routine is in the buffer and to set event flag 4, code P3 as follows:

```
P3 = CIN$M_ISR!CIN$M_EFN!4@CIN$V_EFNUM>
```

See the "Notes" later in this section for additional information on the flags.

p4

Address of the entry mask of an AST service routine to be called as the result of an interrupt.

p5

AST parameter to be passed to the AST completion routine (used as the AST parameter only if the process-supplied interrupt service routine does not overwrite the value).

p6

Number of AST control blocks to preallocate in anticipation of fast, recurrent interrupts from the device.

**Return Status**

SS$_NORMAL

System service successfully completed.

SS$_ACCVIO

The caller does not have the appropriate access to the buffer specified in the P1 argument or to the entry point list specified in the P2 argument.

SS$_BADPARAM

The size of the buffer specified in the P1 argument exceeds 64K bytes, or the number of preallocated AST control blocks specified in the P6 argument exceeds 65767.

SS$_DISCONNECT

A connection is already outstanding for the device, or a condition described in note 2.b (see "Notes") has occurred.

SS$_EXQUOTA

The process has exceeded its direct I/O limit quota or its AST limit quota.

SS$_ILLEFC

An illegal event flag number was specified.

SS$_INSFMEM

Insufficient system dynamic memory is available to complete the system service.

SS$_INSFSPTS

Insufficient system page table entries are available to double map the process buffer. (The value of the REALTIME_SPTS parameter to the SYSGEN utility must be increased.)

SS$_NOPRIV

The process does not have the CMKRNL privilege. This privilege is only required if the user specifies a buffer to be used by the process and the process-specified kernel mode routines.

SS$_UNASEFC

The process is not associated with the cluster containing the specified event flag.

### Privilege Restrictions

The connect-to-interrupt $QIO call does not require privileges if no shared buffer is specified. If the request specifies a buffer descriptor argument, the process must have the CMKRNL privilege.

### Resources Required/Returned

A connect-to-interrupt request updates the process quota values as follows:

- Subtracts the number of preallocated AST control blocks in the P6 argument from the number of outstanding ASTs remaining for the process (ASTCNT)

- Subtracts 1 (for the $QIO) from the direct I/O count (DIOCNT)

**Notes**

1. After the $QIO call is issued, the operation is not completed until the process or the connect-to-interrupt driver cancels I/O on the channel.

2. The connect-to-interrupt driver can cancel I/O on the channel for a number of reasons, including the following:

   a. The driver cannot set the specified event flag, perhaps because the process disassociated from the common event flag cluster after requesting that a flag in that cluster be set.

   b. The driver cannot reallocate AST control blocks quickly enough. This condition can occur because not enough AST control blocks (P6 argument) were specified, because not enough pool space is available for the requested AST control blocks, or because the process ASTCNT quota is exhausted.

   c. The driver cannot queue the AST to the process.

3. If no event flag setting was requested in the P3 argument and if no AST service routine was specified in the P4 argument, P6 if ignored and no AST control blocks are preallocated. If you requested an event flag be set and/or an AST service routine but did not preallocate any AST control blocks (that is, P6 is zero), one AST control block is automatically preallocated, because the system needs one control block to set any event flag or to deliver any ASTs.

   If you request an event flag and/or an AST service routine and if you preallocate any AST control blocks, the CIN$M_REPEAT bit is set automatically in the longword specified in the P3 argument. Thus, as long as you preallocate any AST control blocks, your process will automatically remain connected to the interrupt vector to receive repeated interrupts until the process is disconnected from the interrupt vector.

   If the CIN$M_REPEAT flag is not set, the process is disconnected from the interrupt vector after the first successful interrupt, and a status code of SS$_NORMAL is returned.

### 4.6.6 Conventions for Process-Specified Routines

Any routines that the process specifies in the connect-to-interrupt call are double-mapped, once in process space and once in system space. Each routine executes in kernel mode at an appropriate IPL:

- Device initialization routine after power recovery - IPL 31 (IPL$_POWER)
- Start_I/O routine - IPL 6 (IPL$_QUEUEAST)
- Interrupt service routine - device IPL (assumed to be IPL 22)
- Cancel routine - IPL 6 (IPL$_QUEUEAST)

The process must have the CMKRNL user privilege.

Each routine must:

- Be position independent
- Follow the rules for accessing I/O space (see Section 4.5.3)
- Access only data within the buffer or non-pageable locations in system space
- Perform any necessary synchronization of access to data in the shared buffer
- Save any registers it uses (unless otherwise noted in the remaining sections of this chapter)
- Exit properly
- Not incur exceptions
- Not perform lengthy processing
- Not dispatch to code outside the buffer specified in the Pl argument to the Queue I/O Request call

Sections 4.6.8 through 4.6.11 discuss conventions for specific process specified routines. Section 4.6.12 describes several program examples of connecting to an interrupt vector.

The VAX/VMS Guide to Writing a Device Driver explains how to write a device initialization routine, a start I/O routine, an interrupt service routine, and a cancel I/O routine. That manual also discusses the I/O data structures used by these routines.

### 4.6.7 Programming Language Constraints

Only VAX-11 MACRO or VAX-11 BLISS-32 should be used to code process-specified routines in system space (see Section 4.6.6) or any references to I/O space. There is no assurance that the code generated by compilers for other languages will satisfy all the constraints described in this section.

The following constraints apply to process-specified routines in system space (that is, in the buffer specified in the Pl argument to the $QIO call that establishes the connection to the interrupt vector):

- The compiler must generate position independent code for the routines.

- The generated code and data must be contiguous in virtual space.

- No calls can be made to any procedure outside the buffer. (This restriction includes calls to routines in the VAX-11 Common Run-Time Procedure Library.)

For any references to I/O space, the generated code must follow the rules for accessing I/O space (see Section 4.5.2). Device register access from high-level languages usually requires that the variable equivalent to the register be a 16-bit integer data type. You may need to check the assembly-language code generated by compilers for languages other than VAX-11 MACRO or VAX-11 BLISS-32 to determine whether it follows all necessary conventions.

### 4.6.8 Process-Specified Device Initialization Routine

During recovery from a power failure, VAX/VMS calls the connect-to-interrupt driver's device initialization routine. This routine marks the device as online in the UCB$W_STS field, stores the UCB address in the IDB$L_OWNER field, and then transfers control to the process-specified device initialization routine. The process-specified routine executes in system context at IPL 31 (IPL$_POWER).

If the process specified a JSB interface, the process routine gains control with the following register settings:

| | |
|---|---|
| R0 | address of the unit control block (UCB) |
| R4 | address of the device status register (CSR) |
| R5 | address of the interrupt dispatch block (IDB) |
| R6 | address of the device data block (DDB) |
| R8 | address of the channel request block (CRB) |

If the process specified a CALL interface, the process routine gains control with an argument list pointed to by AP:

| | |
|---|---|
| 0(AP) | argument count of 5 |
| 4(AP) | address of the device status register (CSR) |
| 8(AP) | address of the interrupt dispatch block (IDB) |
| 12(AP) | address of the device data block (DDB) |
| 16(AP) | address of the channel request block (CRB) |
| 20(AP) | address of the unit control block (UCB) |

The process-specified routine may initialize device registers. However, it must not lower IPL, and it must preserve all registers except R0 through R3.

The routine exits with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface). The stack must be as it was when the routine was entered.

### 4.6.9 Process-Specified Start I/O Routine

The process-specified start I/O routine executes in system context at IPL 6 (IPL$_QUEUEAST). It is entered from the connect-to-interrupt driver's start I/O routine. The input to the process-specified start I/O routine is as follows:

| | |
|---|---|
| R2 | address of the counted argument list |
| R3 | address of the I/O request packet (IRP) |
| R5 | address of the unit control block (UCB) |

| | |
|---|---|
| 0(AP) | argument count of 4 |
| 4(AP) | system-mapped address of the process buffer |
| 8(AP) | address of the I/O request packet (IRP) |
| 12(AP) | system-mapped address of the device's CSR |
| 16(AP) | address of the unit control block (UCB) |

The process-specified start I/O routine may set up device registers. It can raise IPL but must not lower it below 6, and must exit at IPL 6. It must preserve all registers except R0 through R4.

The routine exits with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface). The stack must be as it was when the routine was entered.

### 4.6.10 Process-Specified Interrupt Service Routine

A process-specified interrupt service routine is entered when an interrupt from the device occurs. This routine executes in system context at device IPL. The input to the process-specified interrupt service routine is as follows:

| | |
|---|---|
| R2 | address of the counted argument list |
| R4 | address of the interrupt dispatch block (IDB) |
| R5 | address of the unit control block (UCB) |

| | |
|---|---|
| 0(AP) | argument count of 5 |
| 4(AP) | system-mapped address of the process buffer |
| 8(AP) | address of the AST parameter |
| 12(AP) | system-mapped address of the device status register (CSR) |
| 16(AP) | address of the interrupt dispatch block (IDB) |
| 20(AP) | address of the unit control block (UCB) |

This routine is responsible for clearing the interrupt condition (by writing to some device register, for example) if such an operation is required for the device. In addition, the routine may copy the contents of device registers into the shared buffer or into the AST parameter. The routine must also follow these conventions:

- Maintain an IPL equal to or higher than device IPL (If the IPL is raised, the current IPL should first be saved on the stack for later use in restoring IPL.)

- Save and restore all registers other than R0 through R4 used in the routine

- Restore the stack to its original state before exiting

- Place one of the following status values in R0 before exiting:

  low bit clear -- dismiss interrupt (process is not notified of interrupt)

  low bit set -- set event flag if CIN$M_EFN bit is set in P3 argument, and queue AST if P4 specifies an AST service routine

- Exit with a RET instruction (CALL interface) or RSB instruction (JSB interface)

### 4.6.11 Process-Specified Cancel I/O Routine

When the user process issues a cancel I/O request for a device connected to the process, the connect-to-interrupt driver's cancel I/O routine first checks to determine whether the process can indeed cancel I/O for this device. If it can, the process-specified cancel I/O routine is given control. This routine executes in system context at IPL 8 (IPL$_FORK).

If a JSB interface was specified for the process-supplied cancel I/O routine, the following registers are inputs:

| | |
|---|---|
| R2 | negated value of the channel index number |
| R3 | address of the current I/O request packet (IRP) |
| R4 | address of the process control block (PCB) for the process canceling the I/O |
| R5 | address of the unit control block (UCB) |

If a CALL interface was specified, the argument list is as follows:

| | |
|---|---|
| 0(AP) | argument list count of 4 |
| 4(AP) | negated value of the channel index number |
| 8(AP) | address of the current I/O request packet (IRP) |
| 12(AP) | address of the process control block (PCB) for the process canceling the I/O |
| 16(AP) | address of the unit control block (UCB) |

The process-specified cancel I/O routine must not lower IPL below 6 and must exit at IPL 6. It may clear device registers. It must preserve all registers except R0 and R3, and must place a completion status in R0-R1 (which VAX/VMS will place in the I/O status block associated with the connect-to-interrupt $QIO call).

The process-specified cancel I/O routine should not rely on the channel index number unless it checks the UCB$M_BSY bit in UCB$W_STS to confirm that the process is still connected to the device. The routine may set the UCB$M_CANCEL bit in UCB$W_STS.

The routine exits with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface). The stack must be as it was when the routine was entered.

### 4.6.12 **Real-Time Applications Examples**

To understand how the connect-to-interrupt facility is useful for programming real-time devices, consider devices used in three types of real-time applications:

1. Asynchronous event reporting without data: devices that generate an interrupt as the result of an external event not initiated by a programmed request.

2. Program-driven data collection: devices that generate an interrupt as the result of a programmed request, and make the result of the request available as data in a device register at the time of the interrupt.

3. Asynchronous event reporting with data: one device triggers another device by generating an interrupt that causes a programmed request to be sent to the other device, which in turn generates an interrupt.

Examples of these three types of real time applications and models of programs to handle the devices follow.

NOTE

> The configurations described in the examples in this section are not officially supported; DIGITAL does not provide device driver, UETP, or diagnostic support for certain devices mentioned. The examples are provided merely as possible models for users who wish to design real-time applications using unsupported devices or configurations.

Chapter 6 contains a program example illustrating data definitions and coding used to connect to a device interrupt vector.

4.6.12.1 **Example 1: KW11-W Watchdog Timer** - This type of device reports asynchronous external events: it generates an interrupt as a result of an external event not initiated by a programmed request. The only data of interest to be passed to the user process is the occurrence of the external event. Such devices include contact and/or solid state interrupts, and clocks or counters. The program may need to initiate clock and counter devices by means of a programmed request, but any subsequent interrupts are the result of external events only.

In this example, a dual-processor system uses two KW11-W watchdog timers connected back-to-back to monitor CPU failures. Each processor must arm its timer at regular intervals to prevent the timer from operating a relay that outputs an alarm signal. The alarm output of each timer is connected to the receive input of the other watchdog. If processor A fails and its watchdog times out, the alarm output generates an interrupt on processor B via the second watchdog timer.

The watchdog control program on each processor simply addresses the timer at regular intervals. If the interval passes without the timer being addressed, the timer operates an output relay that generates an interrupt to the second CPU. For this example, assume that the interval is 5 seconds (Example 3 later in this section addresses the problem of a much smaller time interval).

The watchdog control program on processor A executes as follows:

1. Assigns a channel to the device

2. Calls $CRMPSC to map to the I/O page in order to address the device registers

3. Issues a connect-to-interrupt $QIO call to connect the program to the watchdog timer for processor B; specifies the addresses of an interrupt service routine and an AST routine

4. Writes a value to a device register to start the timer

5. Calls $SETIMR to request that an event flag be set after a specified interval (for example, 5 seconds)

6. Calls $WAITFR to wait for the event flag

7. When the event flag is set, writes a value to a device register to reset the timer

8. Loops to step 5

The same control program runs on processor B except that it connects to the watchdog timer for processor A. If either processor fails, the watchdog timer generates an interrupt on the other processor.

The standby processor that receives the interrupt gains control in the VAX/VMS connect-to-interrupt driver (CONINTERR), which calls a process-supplied interrupt service routine (defined in step 3 above) that handles the interrupt as follows:

1. Sets the KW11-W switch relay register to clear the timer interrupt condition

2. Sets a status flag that will cause an AST to be delivered to the control program that connected to the interrupt

3. Returns to CONINTERR

CONINTERR completes the interrupt handling as follows:

1. Schedules a fork process at a lower IPL. This fork process, when it gains control, will queue an AST to the user program.

2. Executes an REI instruction to return from the interrupt

The timer control program on the standby processor regains control in an AST routine. This routine responds to the other processor's failure by switching over and assuming control of the other processor's tasks (or whatever is appropriate).

4.6.12.2 **Example 2: AD11-K, AM11-K; A/D Converter with Multiplexer Connected to the UNIBUS** - This type of device provides program-driven data collection: it generates an interrupt as the result of a programmed request to the device, and makes the result of the request available as data in a device register. Typical devices include A/D converters and digital I/O registers.

The data collection operation is usually repetitive for such applications. Therefore, the interrupt service routine must be capable of buffering data from the device in order to ensure that no data is lost due to the high speed data transfer rate. A typical buffer size for this sampling technique might be 32 16-bit words.

In this example, a user program controls an AD11-K/AM11-K combination that accepts analog data from thermocouples. The AD11-K converts analog data to digital data and returns the data in a device register. Every 10 seconds, the program samples 16 to 32 out of 64 channels at gain settings that may vary based on the thermocouple type and previous samplings.

To collect data efficiently, the program buffers data in a process-specified interrupt service routine, and requests delivery of an AST to the user process when all the requested channels have been sampled. To perform variable sampling, the program passes parameters to the interrupt service routine.

The program establishes a protocol to communicate between the program and the interrupt service routine. The protocol defines a data area shared by the main program, the interrupt service routine, and the AST routine. The data area contains parameters from the program and data from the AD11-K. The data area is a 98-word array used as follows:

1. Elements 1-2 of the data area contain an index to the next buffer location to be filled, and a count indicating the number of samplings still to be taken. The main program initializes these values before starting the device. The interrupt service routine reads and modifies these values in the process of copying data and determining when to stop sampling.

2. Elements 3-66 of the data area are reserved for interrupt service routine parameters. Each pair of elements contains the number of a channel, and a gain value. The main program loads these parameters before starting the device.

3. Elements 67-98 of the data area receive the data that the interrupt service routine reads from the AD11-K data buffer register. The AST routine later reads data from this part of the buffer.

The program sets up for the sampling as follows:

1. Assigns a channel to the device

2. Calls $CRMPSC to map to the I/O page in order to address the device registers

3. Initializes the data area by writing a 67 (the index to the next buffer location to be filled) into element 1, and the number of samples to take into element 2 of the data area; zeroes elements 3-98 of the data area

4. Writes channel numbers and gain values into the parameter section of the data area

5. Issues a connect-to-interrupt $QIO call to connect the process to the A/D converter; specifies the addresses of the area to be double mapped, an offset to the ISR, and an AST routine

6. Sets the start and interrupt enable bits in the AD11-K status register to start the A/D converter

7. Calls $HIBER to place the process in a wait state

As soon as the AD11-K has converted the first sample, the device generates an interrupt. The VAX/VMS CONINTERR routine calls the process-specified interrupt service routine. This process-specified routine executes as follows:

1. Computes the next location to be written in the buffer by reading the first element in the data area

2. Reads 12 bits of data from the A/D buffer register into the next location in the buffer

3. Updates the buffer offset and count elements at the beginning of the data area

4. If all requested samples have been collected, writes the address of the data area into the AST parameter, sets a status flag that will cause an AST to be delivered to the control program, and returns to the CONINTERR routine

5. Otherwise, sets the start bit in a device register to restart the device and returns to the CONINTERR routine with a status flag requesting no AST delivery or event flag setting

Based on the interrupt status from the process-specified interrupt service routine, the CONINTERR routine completes the interrupt processing by queuing a fork process that will queue an AST to the user process. When the process gains control in the AST service routine, this routine processes the samples in the following steps:

1. Clears the interrupt enable bit in the device status register

2. Examines the data collected in order to adjust channel selection and/or gain values for the next sampling

3. Copies the data to a file

4. Reinitializes the data area

5. Calls $SCHDWK to wake the process after a short interval (for example, 10 seconds)

6. Returns

When the time interval elapses, the process regains control. The program can then restart the sampling process by again setting the start and interrupt enable bits in the AD11-K status register.

4.6.12.3 **Example 3: KW11-P Real Time Clock and AD11-K Converter Connected to the UNIBUS** - This type of device reports asynchronous external events by collecting data: one device triggers another device by generating an interrupt that causes a programmed request to be sent to the other device, which in turn generates an interrupt. A typical example is a clock-driven A/D operation for precise time sampling as required in signal processing. This processing technique is often used in laboratories. The amount of data collected in such a timed sampling might typically be 200 to 1000 16-bit words.

In this example, the main program sets up the real-time clock to generate interrupts periodically. At regular intervals, the clock interrupt triggers a programmed request for an A/D conversion operation. The AD11-K collects a sample, and interrupts the CPU with a "done" interrupt and 12 bits of data. The AD11-K interrupt service routine buffers the data and, if the buffer is full, causes an AST to be delivered to the process. The process, gaining control in an AST routine, copies the buffered data to another buffer or to disk.

Programming these device functions is slightly more complicated than the previous example. The main program must specify a large buffer to be used in ring fashion to guarantee that data is not lost between clock-driven samplings. In addition, the program must connect to two device interrupts -- one for the clock and one for the A/D converter.

The protocol used by the main program, the interrupt service routine, and the AST routine is similar to the previous example. The data area is larger: 4K words of buffer area follow the parameter area. The A/D converter interrupt service routine and the AST routine treat the 4K-word buffer as four buffer sections of 1K words per section. The first element in each 1K buffer section is a flag indicating whether the section is in use. The AST resets the flag value after copying the contents of the buffer. The interrupt service routine uses a buffer section only if the section's flag value indicates that the buffer has been emptied.

The main program starts the sampling with the following steps:

1. Assigns channels to the clock and to the A/D converter.

2. Calls $CRMPSC to map to the I/O page in order to address the device registers.

3. Initializes the data buffer by writing a 67 (the index to the next buffer location to be filled) into element 1, and the number of samples to take into element 2 of the data area; zeroes elements 3-4096 of the data area; flags each page of the buffer as available.

4. Writes channel numbers and gain values into the parameter segments of the data area.

5. Issues a connect-to-interrupt $QIO call to connect the process to the clock, and specifies the address of an interrupt service routine.

6. Issues a connect-to-interrupt $QIO call to connect the process to the A/D converter; and specifies the addresses of the area to be double mapped, an offset to the interrupt service routine and an AST routine.

7. Sets the sampling interval by writing a 16-bit value into the KW11-P count set buffer register.

8. Starts the clock by setting the run, mode, rate selection, and interrupt enable bits in the KW11-P control and status register. Setting the mode bit causes repeated interrupts generated at a rate specified in the time interval.

9. Calls $HIBER to place the process in a wait state.

The clock interrupts when zero (underflow) occurs during a count-down from the preset interval count. The VAX/VMS CONINTERR routine calls the process-specified clock interrupt service routine. This process-specified routine starts the A/D conversion as follows:

1. Starts the A/D converter by setting the start and interrupt enable bits in the AD11-K status register

2. Sets interrupt status that prevents AST delivery or event flag setting as a result of this interrupt

3. Returns to CONINTERR

Starting the A/D converter results in an interrupt from the AD11-K, and control passes, via CONINTERR, to the AD11-K interrupt service routine. This routine executes as follows:

1. If this sample is the first sample for a new buffer (indicated by a flag in the data area), the routine moves to the next buffer section (branches to error handling if the buffer is still full), and sets up the first two elements of the data area to indicate the buffer section to be written next. Then, it sets the flag at the start of the new buffer section and sets a flag in the data area to indicate that sampling is occurring.

2. The routine computes the next location to be written in the buffer by reading the first location in the data area.

3. The routine reads 12 bits of data from the A/D buffer register into the next location in the buffer.

4. The routine updates the buffer offset and count values in the data area.

5. If this sample fills the data sector, the routine writes the offset of the filled sector from the start of the 4K-word buffer into the AST parameter, sets a status flag that will cause an AST to be delivered to the control program, and sets a flag indicating that a new data section is to be started.

6. The routine returns to CONINTERR.

The AST routine copies and zeroes the next buffer section to indicate that the section is again available to the interrupt service routine. When the next clock interrupt occurs, the data can be written to the next buffer section, even if the AST routine has not yet emptied the previous buffer section.

# CHAPTER 5

# USING SHARED MEMORY

The MA780 is a multiport memory unit that can be attached to VAX-11/780 processors. Each VAX-11/780 processor can support up to two MA780s. Each MA780 has four ports, thereby allowing up to four VAX-11/780 processors to be attached to it. Figure 5-1 illustrates two VAX-11/780 processors attached to an MA780.

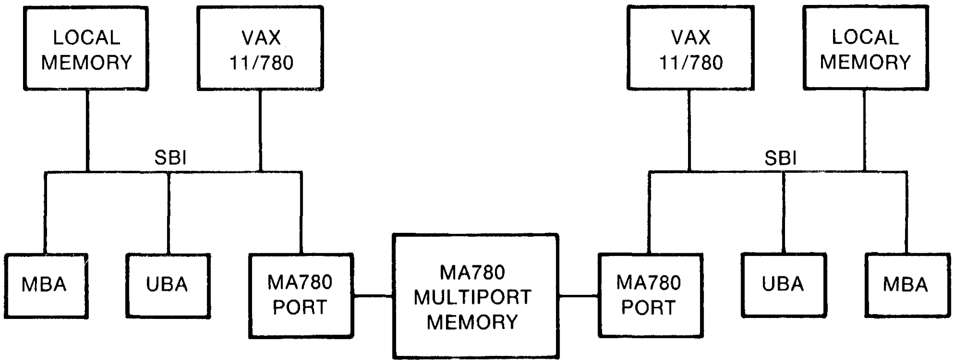


Figure 5-1 Two VAX-11/780s Attached to an MA780

Using one or more multiport memory units, an application can consist of multiple processes running on different VAX-11/780 processors. Regardless of the processor on which they are running, these processes can communicate the completion of an event, send messages, and share common data and code by means of the shared memory.

## 5.1 PREPARING MULTIPORT MEMORY FOR USE

Before an application using multiport memory can execute under VAX/VMS, the system manager must activate the VAX/VMS operating system in processors connected to the multiport memory unit and initialize that memory. The VAX/VMS System Manager's Guide explains the system management responsibilities associated with a multiport memory unit; the present section summarizes the system management functions for the benefit of the application programmer.

First, the system manager activates the VAX/VMS operating system in a VAX-11/780 and initializes the multiport memory unit. These actions cause the following to occur:

- The uninitialized shared memory is connected to the VAX/VMS system running in the processor.

- A name is defined that all processes running in all processors can use to refer to the shared memory (see Section 5.3)
- Limits are set for the following resources in this multiport memory unit:
  - Common event flag clusters: the total number that can be created, and the number that can be created by processes running on this processor
  - Mailboxes: the total number that can be created, and the number that can be created by processes running on this processor
  - Global sections: the total number that can be created, and the number that can be created by processes running on this processor

Then the system manager activates the VAX/VMS operating system in other processors connected to the multiport memory unit. The system manager then connects the initialized shared memory to the VAX/VMS system running in each of these processors and sets limits for the number of common event flag clusters, global sections, and mailboxes that processes on each processor can create in the multiport memory.

The system manager can also install global sections in shared memory just as they are installed in local memory. The INSTALL utility can be used to create shared memory global sections for known files. Once the global sections are installed, a process running in any processor connected to the multiport memory can map to the section, if the process has the appropriate privilege. The process can gain access to the global section either by using a logical name defined by the system manager or by using the section name specified when the global section was created. In the latter case, the section name must be unique on this processor.

## 5.2 PRIVILEGES REQUIRED FOR SHARED MEMORY USE

To use facilities in memory shared by multiple processors, you must have all of the user privileges required to use the equivalent facility in local memory. For example, to create a permanent global section, you must have the PRMGBL privilege, and to create a temporary or permanent mailbox, you must have the TMPMBX or PRMMBX privilege, respectively.

In addition to any other required privileges, you must have the SHMEM privilege to create or delete a common event flag cluster, mailbox, or global section in memory shared by multiple processors. However, you do not need the SHMEM privilege to use an existing cluster, mailbox, or global section in multiport memory.

## 5.3 NAMING FACILITIES IN SHARED MEMORY

To allow access to facilities in memory shared by multiple processors, the system manager and application programmers define names that application programs use to refer to individual shared memory units. During system installation, the system manager defines the name that processes on that particular processor use to refer to the shared memory itself. Application programs define the names that they use to refer to common event flag clusters, global sections, and mailboxes located in the shared memory.

By convention, facilities in shared memory have a name string in the following format:

```
[memory-name:]facility-name
```

memory-name

Name assigned by the system manager during system installation to the shared memory containing the facility. VAX/VMS requires the memory name when you specify a common event flag cluster or mailbox. The colon is recognized as a delimiter separating the two parts of the name string.

facility-name

Logical name assigned to the event flag cluster, global section, or mailbox. The name must contain 15 or fewer characters, and can consist only of alphabetic characters, numeric characters, the dollar sign ($), and the underline (_).

Examples of facility names are:

| | |
|---|---|
| SHRMEM:GS_DATA | Identifies the global section GS_DATA in the shared memory named SHRMEM |
| SHRMEM:MAILBX | Identifies the mailbox MAILBX in the same shared memory |

## 5.4 ASSIGNING LOGICAL NAMES AND LOGICAL NAME TRANSLATION

You can define a logical name for a shared memory facility with the DEFINE or ASSIGN command or the Create Logical Name ($CRELOG) system service. Application programs can then refer to the facility using the logical name; for example, a process can invoke the Create Mailbox and Assign Channel ($CREMBX) system service specifying the logical name for an existing mailbox to which a channel is to be assigned.

When translating a logical name for a shared memory facility, the VAX/VMS operating system uses a slightly different approach from that used for other logical names. The purpose of this approach is to allow programmers to specify either the complete name (memory name and facility name) or a logical name that the system will translate to the complete name. If you define logical names properly, a program that uses a given facility in local memory can be run without change to use the facility in shared memory.

Whenever VAX/VMS encounters the name of a common event flag cluster, mailbox, or global section, it performs the following special logical name translation sequence:

1. Inserts one of the following prefixes to the name (or to the part of the name before the colon if a colon is present):

   CEF$ for common event flag clusters
   MBX$ for mailboxes
   LIB$ for global sections

2. Subjects the resultant string to logical name translation. If translation does not succeed (that is, the original name did not use a logical name), passes the original name string to the system service. If translation does succeed, goes to step 3.

3. Appends the part of the original string after the colon (if any) to the translated name.

4. Repeats steps 1 to 3 (up to nine more times, if necessary) until logical name translation fails. When translation fails, passes the string to the system service.

For example, assume that you have made the following logical name assignment:

```
$ DEFINE MBX$CHKPNT SHRMEM$1:CHKPNT
```

Assume also that your program refers to the mailbox name as CHKPNT in a system service argument. The following logical name translation takes place:

1. MBX$ is prefixed to CHKPNT.

2. MBX$CHKPNT is translated to SHRMEM$1:CHKPNT.

3. No further translation is successful; therefore, the string SHRMEM$1:CHKPNT is passed to the system service.

The logical name definition in the preceding example allows a program that used a mailbox named CHKPNT in local memory to run using the mailbox in shared memory, without being recompiled or relinked.

Note that if a process creates one or more subprocesses and they use a mailbox or common event flag cluster in shared memory, the creator should place the logical name in the group logical name table (for example, specify the /GROUP qualifier with the DEFINE command). If the name is defined in the process logical name table (the default), the subprocesses will not receive the correct equivalence name, because each subprocess has its own process logical name table.

There are two exceptions to the logical name translation method discussed in this section:

- If the facility name starts with an underline (_), the VAX/VMS system strips the underline and considers the resultant string to be the actual name (that is, no further translation is performed).

- If the facility is a global section with a name in the format name_nnn, VAX/VMS first strips the underline and the digits (nnn), then translates the resultant name according to the sequence discussed in this section, and finally reappends the underline and digits. The system uses this method with known images and shared files installed by the system manager.

## 5.5 HOW VAX/VMS FINDS FACILITIES IN SHARED MEMORY

After the VAX/VMS system performs the logical name translation described in Section 5.4, the final equivalence name must be the name of a facility in either the processor's local memory or in shared memory. If the equivalence name specifies the name of a shared memory (that is, the name is in the format name:facility-name), VAX/VMS searches for the facility in the appropriate data base of the specified memory.

If the equivalence name specifies a common event flag cluster or mailbox and does not specify a memory name, VAX/VMS searches through the common event flag cluster data base or the mailbox data base until it locates the specified cluster or mailbox. Absence of a memory name as part of a common event flag cluster name or mailbox name indicates that the facility is located in local memory.

If the equivalence name specifies a global section and does not specify a memory name, VAX/VMS looks for the section as follows:

1. First, it searches the global section tables for sections in the processor's local memory.

2. Then, it searches the global section tables for each initialized shared memory connected to the processor in the order in which they were connected and recognized by the processor.

The result of searching in this order is that global sections in the processor's local memory take precedence over those in shared memories. Thus, absence of a memory name as part of a global section name is not used as an indication of where the global section is located.

## 5.6 USING COMMON EVENT FLAGS IN SHARED MEMORY

Under VAX/VMS, any process can associate with up to two common event flag clusters (event flag numbers 64 through 95 and 96 through 127). These clusters can be located in shared memory or in local memory. To create and associate with a common event flag cluster in shared memory and manipulate flags in the cluster, you use the same steps as you would to associate with a common event flag cluster in local memory:

1. Issue the Associate Common Event Flag Cluster ($ASCEFC) system service to create the cluster or to associate with an existing cluster.

2. Issue any of the services that set, clear, and wait for designated event flags, as appropriate.

As with local memory clusters, the first process among cooperating processes to issue the Associate Common Event Flag Cluster ($ASCEFC) system service causes the cluster to be created. Any other process calling this service and specifying the same cluster associates with that cluster. VAX/VMS implicitly qualifies cluster names with the group number of the creator's UIC; therefore, other cooperating processes must belong to the same group.

All of the event flag system services, with the exception of Associate Common Event Flag Cluster and Disassociate Common Event Flag Cluster, function identically regardless of whether they are used with local or shared memory clusters. The only difference with the associate and disassociate system services is that to specify a cluster in shared memory, you must provide the memory name as well as the cluster name. That is, after VAX/VMS performs logical name translation of the name argument, the cluster name must have the following format:

```
memory-name:cluster-name
```

Section 5.3 describes the name format, and Section 5.4 explains the logical name translation performed by the system.

Section 3.1 discusses common event flags and related system services. The VAX/VMS System Services Reference Manual describes all of the event flag services in detail.

## 5.7 USING MAILBOXES IN SHARED MEMORY

The first process on each processor to refer to a shared memory mailbox must use the Create Mailbox and Assign Channel ($CREMBX) system service to create the mailbox and assign a channel to it. Any $CREMBX system service call referring to a shared memory mailbox must specify a mailbox name that has or translates to the following format (Section 5.4 explains the logical name translation procedure):

```
memory-name:mailbox-name
```

When the mailbox is created, the $CREMBX system service also creates the mailbox-name portion of the name string as a logical name with an equivalence name in the format MBn. For example, if the complete name string is SHMEM:MAILBOX, the system service will create MAILBOX as a logical name with an equivalence name of, for example, MBB005.

The Assign I/O Channel ($ASSIGN) and Deassign I/O Channel ($DASSGN) system services require that you specify only the mailbox-name portion of a shared memory mailbox name string. Likewise, any high-level language program statements that open, close, read from, or write to a shared memory mailbox must specify only the mailbox-name portion.

Figure 5-2 shows two VAX-11 FORTRAN programs using a shared memory mailbox. The memory-name in this example is SHMEM. The programs are running in processes on separate processors.

```
      PROGRAM   ONE
      INTEGER*4 SYS$CREMBX,STATUS,CHAN

      STATUS = SYS$CREMBX(,CHAN,,,,,'SHMEM:MAILBOX')
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

C-- Open the mailbox using the mailbox-name; write a message.

      OPEN (UNIT=1,NAME='MAILBOX',STATUS='NEW')
      WRITE (1,*)  MESSAGE
          .
          .
          .
      END



      PROGRAM   TWO
      INTEGER*4 SYS$CREMBX,STATUS,CHAN

      STATUS = SYS$CREMBX(,CHAN,,,,,'SHMEM:MAILBOX')
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

C-- Open the mailbox using the mailbox-name; read the message.

      OPEN (UNIT=1,NAME='MAILBOX',STATUS='OLD')
      READ (1,*)  MESSAGE
          .
          .
          .
      END
```

Figure 5-2  Using a Shared Memory Mailbox

A mailbox in shared memory cannot be used as process termination mailbox.

Section 3.2 discusses mailboxes and related system services, and includes a programming example.

## 5.8 USING GLOBAL SECTIONS IN SHARED MEMORY

Under VAX/VMS, processes can map global sections located in local memory or in shared memory. A global section in shared memory can be mapped to an image file or a data file, just like a global section in local memory. To create a global section in shared memory, you perform the same steps as you would to create a global section in local memory:

1. Using VAX-11 RMS, open the file to be mapped.

2. Issue the Create and Map Section ($CRMPSC) system service.

The file to be mapped must reside on a disk device attached to the local processor. Once the section is created, however, processes on all processors attached to the shared memory can map the section.

To map an existing global section in shared memory, you issue a Map Global Section ($MGBLSC) system service specifying the name of the section. Once the section is mapped, processes gain access to shared memory global sections in the same manner as they do to local memory sections. VAX/VMS thus makes use of the shared memory unit transparent to the process.

VAX/VMS treats the pages of a global section in shared memory differently from pages in local memory. When a process creates a shared-memory global section, VAX/VMS brings all of the pages of the mapped image or data file into memory. Any process mapped to that global section can gain access to those pages without incurring a page fault because the pages are already in physical memory. Unlike process pages in local memory, global section pages in shared memory are not included in the working sets of the processes that map the section.

Because no paging occurs, VAX/VMS never writes the contents of shared memory global section pages back to their disk file. For read/write global sections in which you want to maintain an updated file while the application executes, you must issue an Update Section File on Disk ($UPDSEC) system service. The process issuing the update request must execute on the same processor as the process that created the global section. You can update the disk file periodically during execution of the application as a checkpoint precaution. The disk file is automatically updated when the section is deleted.

Each process that has mapped a global section in shared memory can unmap the section in either of the following ways:

- Issue a Delete Virtual Address Space ($DELTVA) system service to delete the process's virtual address space that maps the section.

- Terminate the current image, thereby causing VAX/VMS to unmap the process from the section automatically.

Deleting a global section in shared memory requires an explicit deletion request, because all global sections in shared memory must be permanent sections. The deletion request can be either a Delete Global Section ($DGBLSC) system service issued by the application or a deletion request issued by the system manager. In either case, VAX/VMS does not perform the actual deletion until all processes that have mapped the section unmap it.

The VAX/VMS System Services Reference Manual provides information on the use of the VAX/VMS system services used with global sections, that is, memory management system services. Section 5.8.1 of the present manual provides information specifically related to creating and mapping a global section in shared memory. The $CRMPSC, $MGBLSC, $DGBLSC, and $UPDSEC system services are the only memory management system services for which the shared memory has any direct implications.

### 5.8.1 Create and Map Section System Service

The Create and Map Section System Service has the following general formats when issued to create and/or map a global section in multiport memory.

**MACRO Format**

```
    $CRMPSC  [inadr],  [retadr],  [acmode],  [flags],  gsdnam
            ,[ident],  [relpag],  [chan],  [pagcnt],  [vbn],  [prot]
```

**High-Level Language Format**

```
    SYS$CRMPSC  ([inadr],  [retadr],  [acmode],  [flags],  gsdnam
            ,[ident],  [relpag],  [chan],  [pagcnt],  [vbn],  [prot])
```

With the exception of the FLAGS, GSDNAM, and PFC arguments, the arguments of this service are not affected by MA780 considerations.

flags

Mask defining the section type and characteristics. Of the flags defined, the following two must be set.

| Flag | Meaning |
|---|---|
| SEC$M_GBL | Global section |
| SEC$M_PERM | Permanent section |

That is, sections in shared memory must be permanent global sections.

If appropriate, the following flags also can be set.

| Flag | Meaning | Default |
|---|---|---|
| SEC$M_DZRO | Pages are demand-zero pages | Pages are not zeroed when copied |
| SEC$M_WRT | Read/write section | Read-only |
| SEC$M_SYSGBL | System global section | Group global section |
| SEC$M_EXPREG | Map section into the first free range of virtual addresses large enough to hold the section | Map section according to the INADR argument |

Neither SEC$M_CRF (copy-on-reference) nor SEC$M_PFNMAP (page frame number mapping) can be set when using the Create and Map Section system service to create global sections in shared memory. If SEC$M_CRF is set, VAX/VMS places the global section in local memory.

gsdnam

Address of a character string descriptor pointing to the text name string for the global section. This argument is required for creating sections in shared memory.

The string can be either the name of a global section or the logical name of a global section. VAX/VMS performs logical name translation as described in Section 5.4.

VAX/VMS implicitly qualifies global section names with an identification. For group global sections, the section name is also implicitly qualified by the group number of the process creating the global section.

pfc

Page fault cluster size for local memory sections. This argument is ignored for global sections in shared memory, because VAX/VMS reads the file into memory when it creates the section and does not allow paging for sections in shared memory.

# CHAPTER 6

# PRIVILEGED SHAREABLE IMAGES

A privileged shareable image is a shareable image containing one or more routines that nonprivileged users can call to perform privileged functions. The creator of the privileged shareable image codes, compiles or assembles, links, and installs the routine; other users can then call this routine in their programs using the standard CALL interface, provided they have linked their object module(s) with the privileged shareable image. Privileged shareable images thus provide a vehicle for users, in effect, to write and use their own system services.

Because privileged shareable images can be written for any purpose, their use is not limited to real-time applications. However, privileged shareable images can provide real-time users with a suitable vehicle for special-purpose routines that nonprivileged processes in applications can use.

## 6.1 CODING THE PRIVILEGED SHAREABLE IMAGE

The following requirements must be met in coding a privileged shareable image:

- It must contain a special change-mode vector identifying a kernel-mode and/or executive-mode dispatcher.

- Its entry point must be followed by a CHMK or CHME instruction with a negative operand.

- Any kernel-mode or executive-mode dispatcher pointed to in the change-mode vector must validate the CHMK or CHME operand, and must be followed by one or more routines that perform the desired function(s).

- The privileged shareable image (or each routine in it) must enable any necessary user privileges and disable them when they are no longer needed. The Set Privileges ($SETPRV) system service is used to enable and disable user privileges.

Each of the preceding considerations is discussed in the following sections.

### 6.1.1 Change-Mode Vector

One of the program sections in a privileged shareable image must start with a change-mode vector. The purpose of this vector is to point (by means of self-relative offsets) to the start of the kernel-mode or executive-mode dispatch routine within the privileged shareable image.

The program section containing the change-mode vector must be assigned the VEC attribute. (See the VAX-11 MACRO Language Reference Manual or the VAX-11 Linker Reference Manual for a discussion of program section attributes.)

The change-mode vector must have the format shown in Figure 6-1. The offsets from the base of the vector to specific items are expressed by symbols starting with PLV$L_. These symbols are defined by the $PLVDEF macro and are contained in SYS$LIBRARY:LIB.MLB.

| | |
|---|---|
| Vector Type Code<br>(PLV$C_TYP_CMOD) | PLV$L_TYPE |
| System Version Number<br>(SYS$K_VERSION) | PLV$L_VERSION |
| Kernel Mode Dispatcher Offset | PLV$L_KERNEL |
| Exec Mode Entry Offset | PLV$L_EXEC |
| Reserved | |
| Reserved | |
| RMS Dispatcher Offset | PLV$L_RMS |
| Address Check | PLV$L_CHECK |

Figure 6-1 Change-Mode Vector Format

The significant offsets in the change-mode vector and their contents are as follows:

- PLV$L_TYPE - Contains the type code PLV$C_TYP_CMOD, identifying this as a change-mode vector.

- PLV$L_VERSION - Contains the system version number (expressed by the value SYS$K_VERSION). When the privileged shareable image is linked, the linker inserts the value of SYS$K_VERSION into this location. Before the privileged shareable image is used at run time, the VAX/VMS image activator compares this value with the current version number of SYS.EXE; and if the two do not match, the privileged shareable image is not used and an error status is returned.

- PLV$L_KERNEL - Contains a self-relative pointer to the user-supplied kernel-mode dispatcher. ("Self-relative" means relative to the start of the longword field.) A zero value indicates there is no kernel-mode dispatcher.

- PLV$L_EXEC - Contains a self-relative pointer to the user-supplied executive-mode dispatcher. A zero value indicates there is no executive-mode dispatcher.

- PLV$_RMS - Contains a self-relative pointer to the dispatcher for VAX-11 RMS services. A zero value indicates there is no user-supplied VAX-11 RMS dispatcher. Only one privileged shareable image should specify the VAX-11 RMS vector, because only the last value will be used. This field is intended for use only by DIGITAL.
- PLV$L_CHECK - Contains a value to verify that a privileged shareable image that is not position independent is located at the proper virtual address. If the image is position independent, this field should contain zero. If the image is not position independent, this field should contain its own address.

### 6.1.2 Entry Point to the Privileged Shareable Image

The entry point of a privileged shareable image must be an entry mask followed by a CHMK (change mode to kernel) or CHME (change mode to executive) instruction, depending on whether you want control transferred to a kernel-mode or executive-mode dispatcher (specified in the vector). The operand of the CHMK or CHME instruction must be a negative value, because positive values are reserved for calling system services supplied by DIGITAL.

### 6.1.3 Kernel-Mode or Executive-Mode Dispatcher

The kernel-mode or executive-mode dispatch code that you write must:

- Validate the CHMK or CHME operand, handling any invalid operands.
- Transfer control to the appropriate coding segment if the privileged shareable image contains functionally separate coding segments. The CASE instruction in VAX-11 MACRO or a computed GO-TO-type statement in a high-level language provides a convenient mechanism for determining where to transfer control.
- Precede the coding segment(s) performing the function(s) the privileged shareable image was designed to perform.

### 6.1.4 Enabling and Disabling User Privileges

A privileged shareable image must enable any privileges that it needs but that the nonprivileged user of the privileged shareable image lacks. The privileged shareable image must also disable any such privileges before the nonprivileged user receives control again.

To enable or disable a set of privileges, use the Set Privileges ($SETPRV) system service. The following example shows the operator (OPER) and physical I/O (PHY_IO) privileges being enabled.

```
PRVMSK:  .LONG   <1@PRV$V_OPER>!<1@PRV$V_PHY_IO> ;OPER AND PHY_IO
         .LONG   0       ;QUADWORD MASK REQUIRED.  NO BITS SET IN
                         ;HIGH-ORDER LONGWORD FOR THESE PRIVILEGES.
            .
            .
            .
         $SETPRV_S  ENBFLG=#1,-         ;1=enable, 0=disable
                    PRVADR=PRVMSK       ;Identifies the privileges
```

Any code executing in executive or kernel mode is granted an implicit SETPRV privilege.

The VAX/VMS System Services Reference Manual contains an explanation of the Set Privileges ($SETPRV) system service.

## 6.2 LINKING THE PRIVILEGED SHAREABLE IMAGE

The following conventions apply when you link (that is, create) a privileged shareable image:

- Use the /SHAREABLE command qualifier to identify the image to be created as shareable.
- Use the /PROTECT command qualifier or the PROTECT= option to identify the entire image or specific clusters, respectively, as protected against user-mode or supervisor-mode write access (see Section 6.2.1 for further information).
- Define the privileged shareable image's entry point as a universal symbol, using the UNIVERSAL= option.

The listings in Section 6.5 include the LINK command and linker options file used to create the sample privileged shareable image.

### 6.2.1 Specifying Protection for the Image or Clusters

The VAX-11 Linker allows you to protect all or part of a privileged shareable image from write access by code executing in user or supervisor mode. The /PROTECT command qualifier causes all image sections to be so protected. The PROTECT= option in a linker options file permits you to specify protection for individual clusters, thus allowing privileged shareable images to contain parts into which the nonprivileged user can write.

The linker option takes the form PROTECT=YES or PROTECT=NO and precedes the specifications for clusters that are to be protected or unprotected, respectively. The following example shows the linker options file entries to designate clusters A, B, and D as protected, and cluster C as unprotected.

```
PROTECT=YES
CLUSTER=A,,,MODULE1,MODULE2
CLUSTER=B,,,MODULE3,MODULE4,MODULE5
PROTECT=NO
CLUSTER=C,,,MODULE6,MODULE7
PROTECT=YES
CLUSTER=D,,,MODULE8,MODULE9
```

The VAX-11 Linker Reference Manual discusses linker options files and explains each available option.

## 6.3 INSTALLING THE PRIVILEGED SHAREABLE IMAGE

To make a privileged shareable image usable by nonprivileged programs, you must install it as a protected permanent global section. The following procedure is recommended:

1. Move the privileged shareable image to a protected directory, such as SYS$SHARE.

2. Run the INSTALL utility, specifying the /PROTECT, /OPEN, and /SHARED qualifiers. You can also specify the /HEADER_RESIDENT qualifier. The following entry could be used to install the privileged shareable image presented in Section 6.5 (the image name is USS):

   ```
   $ RUN SYS$SYSTEM:INSTALL
   INSTALL>SYS$SHARE:USS/PROTECT/OPEN/SHARED/HEADER_RES
   ```

   The INSTALL utility is discussed in the VAX/VMS System Manager's Guide.

## 6.4 USING THE PRIVILEGED SHAREABLE IMAGE

To the nonprivileged user of a privileged shareable image there is no difference between using it and using an ordinary shareable image. To use a privileged shareable image, the user must:

- Call the privileged shareable image.

- Link the privileged shareable image into the executable image being created. Note: If the shareable image was installed as writeable, you cannot link it into an executable image. You must link an uninstalled copy of the writeable shareable image into the executable image.

## 6.5 PROGRAM LISTINGS

The rest of this chapter contains listings of modules in a privileged shareable image and of a module that calls the privileged shareable image.

USSDISP.LIS

```
USER_SYS_DISP                 - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42        Page    0
Table of contents
```

```
USER_SYS_DISP                 - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42        Page    1
V1.0                                                                   10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

                                   0000     1         .TITLE  USER_SYS_DISP - Example of user system service dispatcher
                                   0000     2         .IDENT  /V1.0/
                                   0000     3 ;
                                   0000     4 ; Copyright (C) 1980
                                   0000     5 ; Digital Equipment Corporation, Maynard, Massachusetts 01754
                                   0000     6 ;
                                   0000     7 ; This software is furnished under a license for use only on a single
                                   0000     8 ; computer  system  and  may be copied only with the inclusion of the
                                   0000     9 ; above copyright notice. This software, or any other copies thereof,
                                   0000    10 ; may not be provided or otherwise made available to any other person
                                   0000    11 ; except for use on such system and to one who agree to these license
                                   0000    12 ; terms.  Title to  and  ownership of the software shall at all times
                                   0000    13 ; remain in DEC.
                                   0000    14 ;
                                   0000    15 ; The information in the software is subject to change without notice
                                   0000    16 ; and should  not  be construed  as a commitment by Digital Equipment
                                   0000    17 ; Corporation.
                                   0000    18 ;
                                   0000    19 ; DEC assumes  no  responsibility  for the use or  reliability of its
                                   0000    20 ; software on equipment which is not supplied by DEC.
                                   0000    21 ;
                                   0000    22 ;
                                   0000    23 ; Facility: Example of User Written System Services
                                   0000    24 ;++
                                   0000    25 ; Abstract:
                                   0000    26 ;       This module contains an example dispatcher for user written
                                   0000    27 ;       system services along with several sample services.  It is a
                                   0000    28 ;       template intend to serve as the starting point for implementing
                                   0000    29 ;       a privileged shareable image containing your own services.  When used as
                                   0000    30 ;       a template, the definitions and code for the sample services
                                   0000    31 ;       should be removed.
```

```
0000     ;
0000     ; Overview:
0000     ;       User written system services are contained in privileged shareable
0000     ;       images that are linked into user program images in exactly the
0000     ;       same fashion as any shareable image.  The creation and installation
0000     ;       of a privileged, shareable image is slightly different from that
0000     ;       of an ordinary shareable image.  These differences are:
0000     ;
0000     ;               1. A vector defining the entry points and providing other
0000     ;                  control information to the image activator.  This vector
0000     ;                  is a the lowest address in an image section with the VEC
0000     ;                  attribute.
0000     ;
0000     ;               2. The shareable image is linked with the /PROTECT option
0000     ;                  that marks all of the image sections so that they will
0000     ;                  protected and given EXEC mode ownership by the image
0000     ;                  activator.
0000     ;
0000     ;               3. The shareable image MUST be installed /SHARE /PROTECT
0000     ;                  with the INSTALL utility in order for the image activator
0000     ;                  to connect the privileged shareable image to the change mode
0000     ;                  dispatchers.
0000     ;
0000     ;       A privileged shareable image implementing user written system services is
0000     ;       comprised of the following major components:
0000     ;


USER_SYS_DISP            - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42        Page   2
V1.0                                                              10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

0000     ;               1. A transfer vector containing all of the entry points and
0000     ;                  collecting them at the lowest virtual address in the shareable
0000     ;                  image.  This formalism enables revision of the shareable
0000     ;                  image without necessitating the relinking of images that
0000     ;                  use it.
0000     ;
0000     ;               2. A Privileged Library Vector in a PSECT with the VEC attribute
0000     ;                  that describes the entry points for dispatching EXEC and
0000     ;                  KERNEL mode services along with validation information.
0000     ;
0000     ;               3. A dispatcher for kernel mode services.  This code will
0000     ;                  be called by the VMS change mode dispatcher when it
0000     ;                  fails to recognize a kernel mode service request.
0000     ;
0000     ;               4. A dispatcher for executive mode services.  This code will
0000     ;                  be called by the VMS change mode dispatcher when it fails
0000     ;                  to recognize an executive mode service request.
0000     ;
0000     ;               5. Service routines to perform the various services.
0000     ;
```

```
0000    78 ;       The first four components are contained in this template and are
0000    79 ;       most easily implemented in MACRO, while the service routines can
0000    80 ;       be implemented in BLISS or MACRO. Other languages may be usable
0000    81 ;       but are not recommended -- particularly if they require runtime
0000    82 ;       support routines or are extravagant in their use of stack or are
0000    83 ;       unable to generate PIC code.
0000    84 ;
0000    85 ;       This example is position-independent (PIC) and it is good practice
0000    86 ;       to implement shareable images this way whenever possible.
0000    87 ;--
0000    88 ;
0000    89 ; Link Command File Example:
0000    90 ;
0000    91 ;       $!
0000    92 ;       $!      Command file to link User System Service example.
0000    93 ;       $!
0000    94 ;       $ LINK/PROTECT/NOSYSSHR/SHARE=USS/MAP=USS/FULL SYS$INPUT/OPTIONS
0000    95 ;       !
0000    96 ;       !       Options file for the link of User System Service example.
0000    97 ;       !
0000    98 ;               SYS$SYSTEM:SYS.STB/SELECTIVE
0000    99 ;       !
0000   100 ;       !       Create a separate cluster for the transfer vector.
0000   101 ;       !
0000   102 ;       CLUSTER=TRANSTER_VECTOR,,,SYS$DISK:[]USSDISP
0000   103 ;       !
0000   104 ;       GSMATCH=LEQUAL,1,1
0000   105 ;
0000   106 ;--
```

```
USER_SYS_DISP                    - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page   3
V1.0                             Declarations and Equates                 10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

0000   108         .SBTTL  Declarations and Equates
0000   109 ;
0000   110 ;       Include Files
0000   111 ;
0000   112
0000   113         .LIBRARY "SYS$LIBRARY:LIB.MLB"  ; Macro library for system structure
0000   114                                         ;  definitions
0000   115 ;
0000   116 ;       Macro Definitions
0000   117 ;
0000   118 ;       DEFINE_SERVICE - A macro to make the appropriate entries in several
0000   119 ;                        different PSECTs required to define an EXEC or KERNEL
0000   120 ;                        mode service.  These include the transfer vector,
0000   121 ;                        the case table for dispatching, and a table containing
0000   122 ;                        the number of required arguments.
0000   123 ;
0000   124 ;       DEFINE_SERVICE Name,Number_of_Arguments,Mode
```

```
          0000       ;
          0000               .MACRO  DEFINE_SERVICE,NAME,NARG=0,MODE=KERNEL
          0000               .PSECT  $$$TRANSFER_VECTOR,PAGE,NOWRT,EXE,PIC
          0000               .ALIGN  QUAD                    ; Align entry points for speed and style
          0000               .TRANSFER       NAME            ; Define name as universal symbol for entry
          0000               .MASK   NAME                    ; Use entry mask defined in main routine
          0000               .IF     IDN MODE,KERNEL
          0000               CHMK    #<KCODE_BASE+KERNEL_COUNTER> ; Change to kernel mode and execute
          0000               RET                             ; Return
          0000               KERNEL_COUNTER=KERNEL_COUNTER+1 ; Advance counter
          0000
          0000               .PSECT  KERNEL_NARG,BYTE,NOWRT,EXE,PIC
          0000               .BYTE   NARG                    ; Define number of required arguments
          0000
          0000               .PSECT  USER_KERNEL_DISP1,BYTE,NOWRT,EXE,PIC
          0000               .WORD   2+NAME-KCASE_BASE       ; Make entry in kernel mode CASE table
          0000
          0000               .IFF
          0000               CHME    #<ECODE_BASE+EXEC_COUNTER> ; Change to executive mode and execute
          0000               RET                             ; Return
          0000               EXEC_COUNTER=EXEC_COUNTER+1     ; Advance counter
          0000
          0000               .PSECT  EXEC_NARG,BYTE,NOWRT,EXE,PIC
          0000               .BYTE   NARG                    ; Define number of required arguments
          0000
          0000               .PSECT  USER_EXEC_DISP1,BYTE,NOWRT,EXE,PIC
          0000               .WORD   2+NAME-ECASE_BASE       ; Make entry in exec mode CASE table
          0000               .ENDC                           ;
          0000               .ENDM   DEFINE_SERVICE          ;
          0000       ;
          0000       ;       Equated Symbols
          0000       ;
          0000
          0000               $PHDDEF                         ; Define process header offsets
          0000               $PLVDEF                         ; Define PLV offsets and values
          0000               $PRDEF                          ; Define processor register numbers
          0000       ;
          0000       ;       Initialize counters for change mode dispatching codes
          0000       ;
00000000  0000       KERNEL_COUNTER=0                        ; Kernel code counter
```

```
USER_SYS_DISP            - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page   4
V1.0                     Declarations and Equates                 10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

           00000000  0000  EXEC_COUNTER=0                           ; Exec code counter
                     0000
                     0000  ;
                     0000  ;       Own Storage
                     0000  ;
               00000000          .PSECT  KERNEL_NARG,BYTE,NOWRT,EXE,PIC
                     0000  KERNEL_NARG:                             ; Base of byte table containing the
                     0000                                           ;  number of required arguments.
               00000000          .PSECT  EXEC_NARG,BYTE,NOWRT,EXE,PIC
                     0000  EXEC_NARG:                               ; Base of byte table containing the
                     0000                                           ;  number of required arguments.


USER_SYS_DISP            - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page   5
V1.0                     Transfer Vector and Service Definitions  10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

                     0000        .SBTTL  Transfer Vector and Service Definitions
                     0000  ;++
                     0000  ; The use of transfer vectors to effect entry to the user written system services
                     0000  ; enables some updating of the shareable image containing them without necessitating
                     0000  ; a re-link of all programs that call them.  The PSECT containinng the transfer
                     0000  ; vector will be positioned at the lowest virtual address in the shareable image
                     0000  ; and so long as the transfer vector is not re-ordered, programs linked with
                     0000  ; one version of the shareable image will continue to work with the next.
                     0000  ;
                     0000  ; Thus as additional services are added to a privileged shareable image, their
                     0000  ; definitions should be added to the end of the following list to ensure that
                     0000  ; programs using previous versions of it will not need to be re-linked.
                     0000  ; To completely avoid relinking existing programs the size of the privileged
                     0000  ; shareable image must not change so some padding will be required to provide
                     0000  ; the opportunity for future growth.
                     0000  ;--
                     0000        DEFINE_SERVICE  USER_GET_TODR,1,KERNEL  ; Service to get value of time
                     0002                                                ;  of day register
                     0002        DEFINE_SERVICE  USER_SET_PFC,2,KERNEL   ; Service to set value of process
                     0004                                                ;  default pagefault cluster
                     0004        DEFINE_SERVICE  USER_NULL,0,EXEC        ; Null exec service
                     0002
                     0002  ;
                     0002  ; The base values used to generate the dispatching codes should be negative for
                     0002  ; user services and must be chosen to avoid overlap with any other privileged
                     0002  ; shareable images that will be used concurrently.  Their definition is
                     0002  ; deferred to this point in the assembly to cause their use in the preceding
                     0002  ; macro calls to be forward references that guarantee the size of the change
                     0002  ; mode instructions to be four bytes.  This satisfies an assumption that is
                     0002  ; made by for services that have to wait and be retried.  The PC for retrying
                     0002  ; the change mode instruction that invokes the service is assumed to be 4 bytes
                     0002  ; less than that saved in the change mode exception frame.  Of course, the

                     0002  ; particular service routine determines whether this is possible.
                     0002  ;
           FFFFFC00  0002  KCODE_BASE=-1024                         ; Base CHMK code value for these services
           FFFFFC00  0002  ECODE_BASE=-1024                         ; Base CHME code value for these services
```

```
          0002          .SBTTL  Change Mode Dispatcher Vector Block
          0002  ;++
          0002  ; This vector is used by the image activator to connect the privileged shareable
          0002  ; image to the VMS change mode dispatcher.  The offsets in the vector are self-
          0002  ; relative to enable the construction of position independent images.  The system
          0002  ; version number will be used by the image activator to verify that this shareable
          0002  ; image was linked with the symbol table for the current system.
          0002  ;
          0002  ;                       Change Mode Vector Format
          0002  ;
          0002  ;          +-------------------------------------------+
          0002  ;          !             Vector Type Code              !   PLV$L_TYPE
          0002  ;          !              (PLV$C_TYP_CMOD)             !
          0002  ;          +-------------------------------------------+
          0002  ;          !          System Version Number            !   PLV$L_VERSION
          0002  ;          !              (SYS$K_VERSION)              !
          0002  ;          +-------------------------------------------+
          0002  ;          !      Kernel Mode Dispatcher Offset        !   PLV$L_KERNEL
          0002  ;          !                                           !
          0002  ;          +-------------------------------------------+
          0002  ;          !         Exec Mode Entry Offset            !   PLV$L_EXEC
          0002  ;          !                                           !
          0002  ;          +-------------------------------------------+
          0002  ;          !                Reserved                   !
          0002  ;          !                                           !
          0002  ;          +-------------------------------------------+
          0002  ;          !                Reserved                   !
          0002  ;          !                                           !
          0002  ;          +-------------------------------------------+
          0002  ;          !          RMS Dispatcher Offset            !   PLV$L_RMS
          0002  ;          !                                           !
          0002  ;          +-------------------------------------------+
          0002  ;          !             Address Check                 !   PLV$L_CHECK
          0002  ;          !                                           !
          0002  ;          +-------------------------------------------+
          0002  ;
          0002  ;
      00000000          .PSECT  USER_SERVICES,PAGE,VEC,PIC,NOWRT,EXE
          0000
00000001  0000          .LONG   PLV$C_TYP_CMOD           ; Set type of vector to change mode
                                                         ; dispatcher
00000000' 0004          .LONG   SYS$K_VERSION            ; Identify system version
00000005' 0008          .LONG   KERNEL_DISPATCH-.        ; Offset to kernel mode dispatcher
00000001' 000C          .LONG   EXEC_DISPATCH-.          ; Offset to executive mode dispatcher
00000000  0010          .LONG   0                        ; Reserved.
00000000  0014          .LONG   0                        ; Reserved.
00000000  0018          .LONG   0                        ; No RMS dispatcher
00000000  001C          .LONG   0                        ; Address check - PIC image
```

```
                                 0020          .SBTTL  Kernel Mode Dispatcher
                                 0020  ;++
                                 0020  ; Input Parameters:
                                 0020  ;
                                 0020  ;       (SP) - Return address if bad change mode value
                                 0020  ;
                                 0020  ;        R0  - Change mode argument value.
                                 0020  ;
                                 0020  ;        R4  - Current PCB Address. (Therefore R4 must be specified in all
                                 0020  ;                register save masks for kernel routines.)
                                 0020  ;
                                 0020  ;        AP  - Argument pointer existing when the change
                                 0020  ;              mode instruction was executed.
                                 0020  ;
                                 0020  ;        FP  - Address of minimal call frame to exit
                                 0020  ;              the change mode dispatcher and return to
                                 0020  ;              the original mode.
                                 0020  ;--
                             00000000          .PSECT  USER_KERNEL_DISP0,BYTE,NOWRT,EXE,PIC
                                 0000  KACCVIO:                                ; Kernel access violation
         50    0000'8F    3C     0000          MOVZWL  #SS$_ACCVIO,R0          ; Set access violation status code
                          04     0005          RET                             ;  and return
                                 0006  KINSFARG:                               ; Kernel insufficient arguments.
         50    0000'8F    3C     0006          MOVZWL  #SS$_INSFARG,R0         ; Set status code and
                          04     000B          RET                             ;  return
                          05     000C  KNOTME: RSB                             ; RSB to forward request
                                 000D
                                 000D  KERNEL_DISPATCH::                       ; Entry to dispatcher
         51    0400 C0    9E     000D          MOVAB   W^-KCODE_BASE(R0),R1    ; Normalize dispatch code value
                    F8    19     0012          BLSS    KNOTME                  ; Branch if code value too low
               02   51    B1     0014          CMPW    R1,#KERNEL_COUNTER      ; Check high limit
                    F3    1E     0017          BGEQU   KNOTME                  ; Branch if out of range
                                 0019  ;
                                 0019  ; The dispatch code has now been verified as being handled by this dispatcher,
                                 0019  ; now the argument list will be probed and the required number of arguments
                                 0019  ; verified.
                                 0019  ;
       51    0000'CF41    9A     0019          MOVZBL  W^KERNEL_NARG[R1],R1    ; Get required argument count
  51    00000004 9F41     DE     001F          MOVAL   @#4[R1],R1              ; Compute byte count including arg count
                                 0027          IFNORD  R1,(AP),KACCVIO         ; Branch if arglist not readable
      0400'CF40     6C    91     002D          CMPB    (AP),W^<KERNEL_NARG-KCODE_BASE>[R0] ; Check for required number
                    D1    1F     0033          BLSSU   KINSFARG                ;  of arguments
                    50    AF     0035          CASEW   R0,-                    ; Case on change mode
                                 0037          -                               ; argument value
                                 0037                  #KCODE_BASE,-           ; Base value
         01    FC00 8F           0037                  #<KERNEL_COUNTER-1>     ; Limit value (number of entries)
                                 003B  KCASE_BASE:                             ; Case table base address for DEFINE_SERVICE
                                 003B  ;
                                 003B  ;       Case table entries are made in the PSECT USER_KERNEL_DISP1 by
                                 003B  ;       invocations of the DEFINE_SERVICE macro.  The three PSECTS,
                                 003B  ;       USER_KERNEL_DISP0,1,2 will be abutted in lexical order at link-time.
                                 003B  ;
                             00000000          .PSECT  USER_KERNEL_DISP2,BYTE,NOWRT,EXE,PIC
                             05  0000          RSB                             ; Return to reject out of
                                 0001                                          ; range value
```

```
USER_SYS_DISP                   - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page   8
V1.0                            Executive Mode Dispatcher                10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

                                     0001   318         .SBTTL  Executive Mode Dispatcher
                                     0001   319 ;++
                                     0001   320 ; Input Parameters:
                                     0001   321 ;
                                     0001   322 ;       (SP) - Return address if bad change mode value
                                     0001   323 ;
                                     0001   324 ;        R0  - Change mode argument value.
                                     0001   325 ;
                                     0001   326 ;        AP  - Argument pointer existing when the change
                                     0001   327 ;              mode instruction was executed.
                                     0001   328 ;
                                     0001   329 ;        FP  - Address of minimal call frame to exit
                                     0001   330 ;              the change mode dispatcher and return to
                                     0001   331 ;              the original mode.
                                     0001   332 ;--
                                 00000000   333         .PSECT  USER_EXEC_DISP0,BYTE,NOWRT,EXE,PIC
                                     0000   334 EACCVIO:                                ; Exec access violation
                  50   0000'8F   3C  0000   335         MOVZWL  #SS$_ACCVIO,R0          ; Set access violation status code
                                 04  0005   336         RET                             ;  and return
                                     0006   337 EINSFARG:                               ; Exec insufficient arguments.
                  50   0000'8F   3C  0006   338         MOVZWL  #SS$_INSFARG,R0         ; Set status code and
                                 04  000B   339         RET                             ;  return
                                 05  000C   340 ENOTME: RSB                             ; RSB to forward request
                                     000D   341
                                     000D   342 EXEC_DISPATCH::                         ; Entry to dispatcher
                  51   0400 C0   9E  000D   343         MOVAB   W^-ECODE_BASE(R0),R1    ; Normalize dispatch code value
                            F8   19  0012   344         BLSS    ENOTME                  ; Branch if code value too low
                       01   51   B1  0014   345         CMPW    R1,#EXEC_COUNTER        ; Check high limit
                            F3   1E  0017   346         BGEQU   ENOTME                  ; Branch if out of range
                                     0019   347 ;
                                     0019   348 ; The dispatch code has now been verified as being handled by this dispatcher,
                                     0019   349 ; now the argument list will be probed and the required number of arguments
                                     0019   350 ; verified.
                                     0019   351 ;
                51   0000'CF41   9A  0019   352         MOVZBL  W^EXEC_NARG[R1],R1      ; Get required argument count
            51   00000004 9F41   DE  001F   353         MOVAL   @#4[R1],R1              ; Compute byte count including arg count
                                     0027   354         IFNORD  R1,(AP),EACCVIO         ; Branch if arglist not readable
                0400'CF40   6C   91  002D   355         CMPB    (AP),W^<EXEC_NARG-ECODE_BASE>[R0] ; Check for required number
                            D1   1F  0033   356         BLSSU   EINSFARG                ;  of arguments
                            50   AF  0035   357         CASEW   R0,-                    ; Case on change mode
                                     0037   358                 -                       ; argument value
                                     0037   359                 #ECODE_BASE,-           ; Base value
                  00   FC00 8F       0037   360                 #<EXEC_COUNTER-1>       ; Limit value (number of entries)
                                     003B   361 ECASE_BASE:                             ; Case table base address for DEFINE_SERVICE
                                     003B   362 ;
                                     003B   363 ;       Case table entries are made in the PSECT USER_EXEC_DISP1 by
                                     003B   364 ;       invocations of the DEFINE_SERVICE macro.  The three PSECTS,
                                     003B   365 ;       USER_EXEC_DISP0,1,2 will be abutted in lexical order at link-time.
                                     003B   366 ;
                                 00000000   367         .PSECT  USER_EXEC_DISP2,BYTE,NOWRT,EXE,PIC
                                 05  0000   368         RSB                             ; Return to reject out of
                                     0001   369                                         ; range value
```

```
USER_SYS_DISP                   - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page   9
V1.0                            Get Time of Day Register Value           10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

                                   0001   371         .SBTTL  Get Time of Day Register Value
                                   0001   372 ;++
                                   0001   373 ; Functional Description:
                                   0001   374 ;       This routine reads the content of the hardware time of day
                                   0001   375 ;       processor register and stores the resulting value at the
                                   0001   376 ;       specified address.
                                   0001   377 ;
                                   0001   378 ; Input Parameters:
                                   0001   379 ;       04(AP) - Address to return time of day value
                                   0001   380 ;       R4 - Address of current PCB
                                   0001   381 ;
                                   0001   382 ; Output Parameters:
                                   0001   383 ;       R0 - Completion Status Code
                                   0001   384 ;--
                             001C  0001   385         .ENTRY  USER_GET_TODR,^M<R2,R3,R4>
              51    04 AC    D0  0003   386         MOVL    4(AP),R1                ; Get address to store time of day register
                                   0007   387         IFNOWRT #4,(R1),10$             ; Branch if not writable
                   61    1B    DB  000D   388         MFPR    #PR$_TODR,(R1)          ; Return current time of day register
        50   00000000'8F    D0  0010   389         MOVL    #SS$_NORMAL,R0          ; Set normal completion status
                             04  0017   390         RET                             ;  and return
                                 0018   391
            50     0000'8F    3C  0018   392 10$:    MOVZWL  #SS$_ACCVIO,R0          ; Indicate access violation
                             04  001D   393         RET                             ;

USER_SYS_DISP                   - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page  10
V1.0                            Set Page Fault Cluster Factor            10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

                                   001E   395         .SBTTL  Set Page Fault Cluster Factor
                                   001E   396 ;++
                                   001E   397 ; Functional Description:
                                   001E   398 ;       This routine sets the page fault cluster to the specified value
                                   001E   399 ;       and returns the previous value.
                                   001E   400 ;
                                   001E   401 ; Input Parameters:
                                   001E   402 ;       04(AP) - New value for Page Fault Cluster factor
                                   001E   403 ;       08(AP) - Address to return previous value
                                   001E   404 ;                (0 means none)
                                   001E   405 ;       R4 - PCB address of current process
                                   001E   406 ;
                                   001E   407 ; Output Parameters:
                                   001E   408 ;       R0 - Completion Status code
                                   001E   409 ;--
                             0030  001E   410         .ENTRY  USER_SET_PFC,^M<R4,R5>
        55   00000000'9F    D0  0020   411         MOVL    @#CTL$GL_PHD,R5         ; Get address of process header
              51    08 AC    D0  0027   412         MOVL    8(AP),R1                ; Get address to store previous value
                        0A    13  002B   413         BEQL    10$                     ; Branch if none
                                 002D   414         IFNOWRT #4,(R1),30$             ; Branch if not writable
              61    34 A5    9A  0033   415         MOVZBL  PHD$B_DFPFC(R5),(R1)    ; Return current value
           7F 8F    04 AC    91  0037   416 10$:    CMPB    4(AP),#127              ; Check for legal value
                        04    1B  003C   417         BLEQU   20$                     ; Branch if legal
              50    7F 8F    90  003E   418         MOVB    #127,R0                 ; Set to maximum value
              34 A5    50    90  0042   419 20$:    MOVB    R0,PHD$B_DFPFC(R5)      ; Set new value into PHD
        50   00000000'8F    D0  0046   420         MOVL    #SS$_NORMAL,R0          ; Set normal completion status
                             04  004D   421         RET                             ;  and return
                             004E   422
   50   0000'8F       3C     004E   423  30$:     MOVZWL  #SS$_ACCVIO,R0          ; Indicate access violation
                      04     0053   424           RET                             ;
                             0054   425

USER_SYS_DISP               - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42        Page  11
V1.0                         Null Service                            10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

                             0054   427           .SBTTL  Null Service
                             0054   428  ;++
                             0054   429  ; Functional Description:
                             0054   430  ;
                             0054   431  ; Input Parameters:
                             0054   432  ;
                             0054   433  ; Output Parameters:
                             0054   434  ;
                             0054   435  ;--
                             0054   436
                    0000     0054   437           .ENTRY  USER_NULL,^M<>          ; Entry definition
   50   0000'8F·      3C     0056   438           MOVZWL  #SS$_NORMAL,R0          ; Set normal completion status
                      04     005B   439           RET                             ;  and return
                             005C   440
                             005C   441           .END
```

```
USER_SYS_DISP                  - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page  12
Symbol table                                                            10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

BIT...         = 00000000              PHD$L_R13         000000B8              PHD$W_WSQUOTA     00000018
CTL$GL_PHD       ********   X  0C      PHD$L_R2          0000008C              PLV$C_TYP_CMOD  = 00000001
EACCVIO          00000000 R    0B      PHD$L_R3          00000090              PLV$C_TYP_MSG   = 00000002
ECASE_BASE       0000003B R    0B      PHD$L_R4          00000094              PLV$L_CHECK       0000001C
ECODE_BASE     = FFFFFC00              PHD$L_R5          00000098              PLV$L_EXEC        0000000C
EINSFARG         00000006 R    0B      PHD$L_R6          0000009C              PLV$L_KERNEL      00000008
ENOTME           0000000C R    0B      PHD$L_R7          000000A0              PLV$L_MSGDSP      00000008
EXEC_COUNTER   = 00000001              PHD$L_R8          000000A4              PLV$L_RMS         00000018
EXEC_DISPATCH    0000000D RG   0B      PHD$L_R9          000000A8              PLV$L_TYPE        00000000
EXEC_NARG        00000000 R    04      PHD$L_REFERFLT    00000014              PLV$L_VERSION     00000004
GBL...         = 00000000              PHD$L_RESLSTH     000000F0              PR$S_SID_ECO    = 00000008
KACCVIO          00000000 R    09      PHD$L_SPARE       0000013C              PR$S_SID_PL     = 00000004
KCASE_BASE       0000003B R    09      PHD$L_SSP         0000007C              PR$S_SID_SN     = 0000000C
KCODE_BASE     = FFFFFC00              PHD$L_TIMREF      00000100              PR$S_SID_TYPE   = 00000008
KERNEL_COUNTER = 00000002              PHD$L_USP         00000080              PR$V_SID_ECO    = 00000010
KERNEL_DISPATCH  0000000D RG   09      PHD$L_WSL         00000180              PR$V_SID_PL     = 0000000C
KERNEL_NARG      00000000 R    03      PHD$M_DALCSTX   = 00000002              PR$V_SID_SN     = 00000000
KINSFARG         00000006 R    09      PHD$M_PFMFLG    = 00000001              PR$V_SID_TYPE   = 00000018
KNOTME           0000000C R    09      PHD$M_WSPEAKCHK= 00000004              PR$_ACCR        = 00000029
PHD$B_ASTLVL     000000CB              PHD$Q_AUTHPRIV    000000DC              PR$_ACCS        = 00000028
PHD$B_CPUMODE    0000005C              PHD$Q_IMAGPRIV    000000E4              PR$_ASTLVL      = 00000013
PHD$B_DFPFC      00000034              PHD$Q_PRIVMSK     00000000              PR$_CADR        = 00000025
PHD$B_PAGFIL     0000001F              PHD$S_ASTLVL    = 00000008              PR$_CAER        = 00000027
PHD$B_PGTBPFC    00000035              PHD$S_P0LR      = 00000018              PR$_CMIERR      = 00000017
PHD$C_LENGTH     00000180              PHD$V_ASTLVL    = 00000018              PR$_CSRD        = 0000001D
PHD$C_PHDPAGCTX= 00000008              PHD$V_DALCSTX   = 00000001              PR$_CSRS        = 0000001C
PHD$K_LENGTH     00000180              PHD$V_P0LR      = 00000000              PR$_CSTD        = 0000001F
PHD$L_BIOCNT     00000054              PHD$V_PFMFLG    = 00000000              PR$_CSTS        = 0000001E
PHD$L_CPULIM     00000058              PHD$V_WSPEAKCHK= 00000002              PR$_ESP         = 00000001
PHD$L_CPUTIM     00000038              PHD$W_ASTLM       00000040              PR$_ICCS        = 00000018
PHD$L_DIOCNT     00000050              PHD$W_BAK         00000044              PR$_ICR         = 0000001A
PHD$L_ESP        00000078              PHD$W_CWSLX       000000DA              PR$_IPL         = 00000012
PHD$L_FREP0VA    00000028              PHD$W_DFWSCNT     0000001A              PR$_ISP         = 00000004
PHD$L_FREP1VA    00000030              PHD$W_EMPTPG      000000D4              PR$_KSP         = 00000000
PHD$L_FREPTECNT  0000002C              PHD$W_EXTDYNWS    00000072              PR$_MAPEN       = 00000038
PHD$L_IMGCNT     000000F4              PHD$W_FLAGS       00000036              PR$_MCESR       = 00000026
PHD$L_KSP        00000074              PHD$W_PHVINDEX    00000042              PR$_NICR        = 00000019
PHD$L_P0BR       000000C4              PHD$W_PRCLM       0000003E              PR$_P0BR        = 00000008
PHD$L_P0LRASTL   000000C8              PHD$W_PST         00000020              PR$_P0LR        = 00000009
PHD$L_P1BR       000000CC              PHD$W_PSTBASMAX   00000046              PR$_P1BR        = 0000000A
PHD$L_P1LR       000000D0              PHD$W_PSTFREE     00000026              PR$_P1LR        = 0000000B
PHD$L_PAGEFLTS   00000048              PHD$W_PSTLAST     00000024              PR$_PCBB        = 00000010
PHD$L_PAGFIL     0000001C              PHD$W_PTCNTACT    0000006C              PR$_PME         = 0000003D
PHD$L_PC         000000BC              PHD$W_PTCNTLCK    00000068              PR$_RXCS        = 00000020
PHD$L_PCB        00000074              PHD$W_PTCNTMAX    0000006E              PR$_RXDB        = 00000021
PHD$L_PFLREF     000000FC              PHD$W_PTCNTVAL    0000006A              PR$_SBIER       = 00000034
PHD$L_PFLTRATE   000000F8              PHD$W_QUANT       0000003C              PR$_SBIFS       = 00000030
PHD$L_PGFLTIO    0000004C              PHD$W_REQPGCNT    000000D8              PR$_SBIMT       = 00000033
PHD$L_PSL        000000C0              PHD$W_RESPGCNT    000000D6              PR$_SBIQC       = 00000036
PHD$L_PSTBASOFF  00000020              PHD$W_WSAUTH      0000000A              PR$_SBIS        = 00000031
PHD$L_PTWSLELCK  00000060              PHD$W_WSDYN       0000000E              PR$_SBISC       = 00000032
PHD$L_PTWSLEVAL  00000064              PHD$W_WSFLUID     00000070              PR$_SBITA       = 00000035
PHD$L_R0         00000084              PHD$W_WSLAST      00000012              PR$_SBR         = 0000000C
PHD$L_R1         00000088              PHD$W_WSLIST      00000008              PR$_SCBB        = 00000011
PHD$L_R10        000000AC              PHD$W_WSLOCK      0000000C              PR$_SID         = 0000003E
PHD$L_R11        000000B0              PHD$W_WSLX        00000046              PR$_SID_TYP750  = 00000002
PHD$L_R12        000000B4              PHD$W_WSNEXT      00000010              PR$_SID_TYP780  = 00000001
```

```
USER_SYS_DISP                 - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page  13
Symbol table                                                           10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

PR$_SID_TYP7ZZ = 00000003
PR$_SID_TYPMAX = 00000003
PR$_SIRR       = 00000014
PR$_SISR       = 00000015
PR$_SLR        = 0000000D
PR$_SSP        = 00000002
PR$_TBDR       = 00000024
PR$_TBIA       = 00000039
PR$_TBIS       = 0000003A
PR$_TODR       = 0000001B
PR$_TXCS       = 00000022
PR$_TXDB       = 00000023
PR$_UBRESET    = 00000037
PR$_USP        = 00000003
PR$_WCSA       = 0000002C
PR$_WCSD       = 0000002D
SS$_ACCVIO       ********   X    09
SS$_INSFARG      ********   X    09
SS$_NORMAL       ********   X    0C
SYS$K_VERSION    ********   X    08
USER_GET_TODR    00000001 RG     0C
USER_NULL        00000054 RG     0C
USER_SET_PFC     0000001E RG     0C

                                                        +----------------+
                                                        ! Psect synopsis !
                                                        +----------------+
PSECT name                    Allocation          PSECT No.  Attributes
----------                    ----------          ---------  ----------
.  ABS  .                     00000000  (     0.)  00 (   0.)  NOPIC   USR   CON   ABS   LCL NOSHR NOEXE NORD   NOWRT NOVEC BYTE
. BLANK .                     00000000  (     0.)  01 (   1.)  NOPIC   USR   CON   REL   LCL NOSHR   EXE   RD     WRT NOVEC BYTE
$ABS$                         00000184  (   388.)  02 (   2.)  NOPIC   USR   CON   ABS   LCL NOSHR   EXE   RD     WRT NOVEC BYTE
KERNEL_NARG                   00000002  (     2.)  03 (   3.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC BYTE
EXEC_NARG                     00000001  (     1.)  04 (   4.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC BYTE
$$$TRANSFER_VECTOR            00000017  (    23.)  05 (   5.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC PAGE
USER_KERNEL_DISP1             00000004  (     4.)  06 (   6.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC BYTE
USER_EXEC_DISP1               00000002  (     2.)  07 (   7.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC BYTE
USER_SERVICES                 00000020  (    32.)  08 (   8.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT   VEC PAGE
USER_KERNEL_DISP0             0000003B  (    59.)  09 (   9.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC BYTE
USER_KERNEL_DISP2             00000001  (     1.)  0A (  10.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC BYTE
USER_EXEC_DISP0               0000003B  (    59.)  0B (  11.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC BYTE
USER_EXEC_DISP2               0000005C  (    92.)  0C (  12.)    PIC   USR   CON   REL   LCL NOSHR   EXE   RD   NOWRT NOVEC BYTE

                                                     +------------------------+
                                                     ! Performance indicators !
                                                     +------------------------+
Phase                        Page faults    CPU Time        Elapsed Time
-----                        -----------    --------        ------------
Initialization                         8    00:00:00.04     00:00:00.18
Command processing                    13    00:00:00.18     00:00:00.46
Pass 1                               306    00:00:06.64     00:00:09.97
Symbol table sort                      7    00:00:00.25     00:00:00.41
Pass 2                               200    00:00:01.49     00:00:02.00
Symbol table output                   27    00:00:00.12     00:00:00.15
Psect synopsis output                  5    00:00:00.06     00:00:00.06
```

```
USER_SYS_DISP                   - Example of user system service dispatc 10-MAR-1980 15:48:30  VAX-11 Macro V02.42         Page  14
VAX-11 Macro Run Statistics                                              10-MAR-1980 15:48:21  _DBB2:[HUSTVEDT.USS]USSDISP.MAR;23(1)

Cross-reference output            0     00:00:00.00    00:00:00.00
Assembler run totals            567     00:00:08.78    00:00:13.24

The working set limit was 293 pages.
27596 bytes (54 pages) of virtual memory were used to buffer the intermediate code.
There were 10 pages of symbol table space allocated to hold 194 non-local and 4 local symbols.
441 source lines were read in Pass 1, producing 41 object records in Pass 2.
17 pages of virtual memory were used to define 15 macros.

                                             +---------------------------+
                                             ! Macro library statistics !
                                             +---------------------------+
Macro library name                            Macros defined
------------------                            --------------
_DRA5:[SYSLIB]LIB.MLB;1                  14
_DRA5:[SYSLIB]STARLET.MLB;1                  0
TOTALS (all libraries)                                  14

427 GETS were required to define 14 macros.

There were no errors, warnings or information messages.

USSDISP/LIS
```

```
                          USSTEST.LIS
USSTEST                                                       10-MAR-1980 15:12:23  VAX-11 Macro V02.42        Page   0
Table of contents

   (1)      45      Sample invocation of user written system


USSTEST                                                       10-MAR-1980 15:12:23  VAX-11 Macro V02.42        Page   1
V1.0                                                          10-MAR-1980 15:02:56  _DBB2:[HUSTVEDT.USS]USSTEST.MAR;5 (1)

                                    0000     1        .TITLE  USSTEST
                                    0000     2        .IDENT  /V1.0/
                                    0000     3 ;
                                    0000     4 ; Copyright (C) 1980
                                    0000     5 ; Digital Equipment Corporation, Maynard, Massachusetts 01754
                                    0000     6 ;
                                    0000     7 ; This software is furnished under a license for use only on a single
                                    0000     8 ; computer  system  and  may be copied only with the inclusion of the
                                    0000     9 ; above copyright notice. This software, or any other copies thereof,
                                    0000    10 ; may not be provided or otherwise made available to any other person
                                    0000    11 ; except for use on such system and to one who agree to these license
                                    0000    12 ; terms.  Title to  and  ownership of the software shall at all times
                                    0000    13 ; remain in DEC.
                                    0000    14 ;
                                    0000    15 ; The information in the software is subject to change without notice
                                    0000    16 ; and should  not  be construed  as a commitment by Digital Equipment
                                    0000    17 ; Corporation.
                                    0000    18 ;
                                    0000    19 ; DEC assumes  no  responsibility  for the use or  reliability of its
                                    0000    20 ; software on equipment which is not supplied by DEC.
                                    0000    21 ;
                                    0000    22 ;
                                    0000    23 ; Facility: Example of User Written System Services
                                    0000    24 ;++
                                    0000    25 ; Abstract:
                                    0000    26 ;       This module contains an example of a program that invokes a sample
                                    0000    27 ;       user-written system service that is contained in a privileged
                                    0000    28 ;       shareable image.  The module USSDISP contains the sample service
                                    0000    29 ;       and associated dispatching code being invoked by this simple test
                                    0000    30 ;       program.
                                    0000    31 ;--
                                    0000    32 ; Link Command File:
                                    0000    33 ;
                                    0000    34 ;       $ !
                                    0000    35 ;       $ !     Link Command file for USSTEST
                                    0000    36 ;       $ !
                                    0000    37 ;       $ LINK USSTEST/MAP/FULL,SYS$INPUT/OPTIONS
                                    0000    38 ;       !
                                    0000    39 ;       !       Options file for USSTEST
                                    0000    40 ;               USS.EXE/SHARE
                                    0000    41 ;
                                    0000    42 ;--
                          00000000  0000    43 BUF:    .LONG   0                       ; Location to receive TODR contents
```

```
USSTEST                                                   10-MAR-1980 15:12:23  VAX-11 Macro V02.42        Page   2
V1.0                        Sample invocation of user written system 10-MAR-1980 15:02:56  _DBB2:[HUSTVEDT.USS]USSTEST.MAR;5 (1)

                                0004    45         .SBTTL  Sample invocation of user written system service
                                0004    46 ;++
                                0004    47 ; Functional Description:
                                0004    48 ;       This routine shows an invocation of the example user system service that
                                0004    49 ;       will read the contents of the time of day register.
                                0004    50 ;
                                0004    51 ;       As can be seen by this example, the privileged nature of the code used
                                0004    52 ;       to implement the reading of the TODR is not visible to the caller.
                                0004    53 ;       For coding convenience and better maintainability, the code can be
                                0004    54 ;       generated by macros patterned on the standard VMS system service macros.
                                0004    55 ;
                                0004    56 ;--
                                0004    57
                                0004    58
                           0000 0004    59         .ENTRY  USSTEST,^M<>             ; Entry mask and definition
                      F7 AF  9F 0006    60         PUSHAB  BUF                      ; Build argument list - set address for
                                0009    61                                          ;  return value
           00000000'EF   01  FB 0009    62         CALLS   #1,USER_GET_TODR         ; Invoke routine in privileged sh. image
                                0010    63                                          ;  to get value from Time-of-day register
                             04 0010    64         RET                              ;
                                0011    65
                                0011    66         .END    USSTEST                  ;
```

```
USSTEST                                                   10-MAR-1980 15:12:23  VAX-11 Macro V02.42        Page   3
Symbol table                                              10-MAR-1980 15:02:56  _DBB2:[HUSTVEDT.USS]USSTEST.MAR;5 (1)

BUF              00000000 R     01
USER_GET_TODR    ********   X   01
USSTEST          00000004 RG    01

                                          +----------------+
                                          ! Psect synopsis !
                                          +----------------+

PSECT name                     Allocation          PSECT No.  Attributes
----------                     ----------          ---------  ----------
.  ABS  .                      00000000  (    0.)  00 (  0.)  NOPIC    USR    CON    ABS    LCL NOSHR NOEXE NORD   NOWRT NOVEC BYTE
. BLANK .                      00000011  (   17.)  01 (  1.)  NOPIC    USR    CON    REL    LCL NOSHR   EXE   RD     WRT NOVEC BYTE

                                       +------------------------+
                                       ! Performance indicators !
                                       +------------------------+

Phase                       Page faults    CPU Time       Elapsed Time
-----                       -----------    --------       ------------
Initialization                       9     00:00:00.05    00:00:00.16
Command processing                  14     00:00:00.18    00:00:00.98
Pass 1                              33     00:00:00.23    00:00:01.11
Symbol table sort                    0     00:00:00.00    00:00:00.00
Pass 2                              35     00:00:00.14    00:00:00.21
Symbol table output                  0     00:00:00.01    00:00:00.01
Psect synopsis output                2     00:00:00.02    00:00:00.02
Cross-reference output               0     00:00:00.00    00:00:00.00
Assembler run totals                95     00:00:00.63    00:00:02.49
```

```
The working set limit was 200 pages.
673 bytes (2 pages) of virtual memory were used to buffer the intermediate code.
There were 10 pages of symbol table space allocated to hold 3 non-local and 0 local symbols.
66 source lines were read in Pass 1, producing 13 object records in Pass 2.
0 pages of virtual memory were used to define 0 macros.

                                           +--------------------------+
                                           ! Macro library statistics !
                                           +--------------------------+
Macro library name                          Macros defined
------------------                          --------------
_DRA5:[SYSLIB]STARLET.MLB;1                0

0 GETS were required to define 0 macros.

There were no errors, warnings or information messages.

USSTEST/LIS
```

```
!                       USSLNK.COM


$!
$!      Command file to link User System Service example.
$!
$ LINK/PROTECT/NOSYSSHR/SHARE=USS/MAP=USS/FULL SYS$INPUT/OPTIONS
!
!       Options file for the link of User System Service example.
!
        SYS$SYSTEM:SYS.STB/SELECTIVE
!
!       Create a separate cluster for the transfer vector.
!
CLUSTER=TRANSTER_VECTOR,,,SYS$DISK:[]USSDISP
!
GSMATCH=LEQUAL,1,1
```

```
!                       USSTSTLNK.COM
!
$ !
$ !     Link Command file for USSTEST
$ !
$ LINK USSTEST/MAP/FULL,SYS$INPUT/OPTIONS
!
!       Options file for USSTEST
        USS.EXE/SHARE
```

```
                        USS.MAP
USS                                              10-MAR-1980 15:48       LINKER V02.42                 Page     1
                                     +------------------------+
                                     ! Object Module Synopsis !
                                     +------------------------+

Module Name      Ident              Bytes      File                               Creation Date      Creator
-----------      -----              -----      -----                              -------------      -------
USER_SYS_DISP    V1.0                 275 _DBB2:[HUSTVEDT.USS]USSDISP.OBJ;18   10-MAR-1980 15:48  VAX-11 Macro V02.42
SYS              .STB;1                 0 _DRA5:[SYSEXE]SYS.STB;1               5-MAR-1980 20:17  LINK-32 V02.42
SYSVECTOR        0221                   0 _DRA5:[SYSLIB]STARLET.OLB;1           5-MAR-1980 00:11  VAX-11 Macro V02.42

_DBB2:[HUSTVEDT.USS]USS.EXE;19                    10-MAR-1980 15:48       LINKER V02.42                 Page     2
                                     +------------------------+
                                     ! Image Section Synopsis !
                                     +------------------------+

   Cluster        Type Pages   Base Addr Disk VBN PFC Protection and Paging      Global Sec. Name   Match     Majorid   Minorid
   -------        ---- -----   --------- -------- --- ---------------------      ----------------   -----     -------   -------

TRANSTER_VECTOR     4     1     00000200        2   0 READ ONLY
                    4     1     00000400        3   0 READ ONLY

_DBB2:[HUSTVEDT.USS]USS.EXE;19                    10-MAR-1980 15:48       LINKER V02.42                 Page     3
                                    +--------------------------+
                                    ! Program Section Synopsis !
                                    +--------------------------+

Psect Name       Module Name         Base     End            Length           Align               Attributes
----------       -----------         ----     ---            ------           -----               ----------

. BLANK .                          00000000 00000000 00000000 (          0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT,NOVEC
                 SYSVECTOR         00000000 00000000 00000000 (          0.) BYTE 0

$$$TRANSFER_VECTOR                 00000200 00000216 00000017 (         23.) PAGE 9   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     00000200 00000216 00000017 (         23.) PAGE 9

. BLANK .                          00000200 00000200 00000000 (          0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT,NOVEC
                 USER_SYS_DISP     00000200 00000200 00000000 (          0.) BYTE 0

EXEC_NARG                          00000217 00000217 00000001 (          1.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     00000217 00000217 00000001 (          1.) BYTE 0

KERNEL_NARG                        00000218 00000219 00000002 (          2.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     00000218 00000219 00000002 (          2.) BYTE 0

USER_EXEC_DISP0                    0000021A 00000254 0000003B (         59.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     0000021A 00000254 0000003B (         59.) BYTE 0

USER_EXEC_DISP1                    00000255 00000256 00000002 (          2.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     00000255 00000256 00000002 (          2.) BYTE 0

USER_EXEC_DISP2                    00000257 000002B2 0000005C (         92.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     00000257 000002B2 0000005C (         92.) BYTE 0

USER_KERNEL_DISP0                  000002B3 000002ED 0000003B (         59.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     000002B3 000002ED 0000003B (         59.) BYTE 0
```

```
USER_KERNEL_DISP1                  000002EE 000002F1 00000004 (         4.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     000002EE 000002F1 00000004 (         4.) BYTE 0

USER_KERNEL_DISP2                  000002F2 000002F2 00000001 (         1.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 USER_SYS_DISP     000002F2 000002F2 00000001 (         1.) BYTE 0

USER_SERVICES                      00000400 0000041F 00000020 (        32.) PAGE 9   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,  VEC
                 USER_SYS_DISP     00000400 0000041F 00000020 (        32.) PAGE 9

_DBB2:[HUSTVEDT.USS]USS.EXE;19                           10-MAR-1980 15:48        LINKER V02.42                  Page    4
                                                 +------------------+
                                                 ! Symbols By Name  !
                                                 +------------------+

Symbol           Value                Symbol           Value                Symbol           Value                Symbol           Value
------           -----                ------           -----                ------           -----                ------           -----
CTL$GL_PHD       7FFEFE88
EXEC_DISPATCH    00000227-R
KERNEL_DISPATCH  000002C0-R
SS$_ACCVIO       0000000C
SS$_INSFARG      00000114
SS$_NORMAL       00000001
SYS$K_VERSION    35503058
USER_GET_TODR    00000258-RU
USER_NULL        000002AB-RU
USER_SET_PFC     00000275-RU
_DBB2:[HUSTVEDT.USS]USS.EXE;19                           10-MAR-1980 15:48        LINKER V02.42                  Page    5
                                                 +-------------------+
                                                 ! Symbols By Value  !
                                                 +-------------------+

Value                                   Symbols...
-----                                   ----------
00000001        SS$_NORMAL
0000000C        SS$_ACCVIO
00000114        SS$_INSFARG
00000227      R-EXEC_DISPATCH
00000258      R-USER_GET_TODR
00000275      R-USER_SET_PFC
000002AB      R-USER_NULL
000002C0      R-KERNEL_DISPATCH
35503058        SYS$K_VERSION
7FFEFE88        CTL$GL_PHD


          Key for special characters above:
                +------------------+
                ! *  - Undefined   !
                ! U  - Universal   !
                ! R  - Relocatable !
                ! WK - Weak        !
                +------------------+
_DBB2:[HUSTVEDT.USS]USS.EXE;19                             10-MAR-1980 15:48        LINKER V02.42              Page    6
                                                                 +-----------------+
                                                                 ! Image Synopsis !
                                                                 +-----------------+

Virtual memory allocated:                 00000200 000005FF 00000400 (1024. bytes, 2. pages)
Stack size:                                      0. pages
Image header virtual block limits:               1.         1. (     1. block)
Image binary virtual block limits:               2.         3. (     2. blocks)
Image name and identification:            USS .STB;
Number of files:                                 3.
Number of modules:                               3.
Number of program sections:                     18.
Number of global symbols:                        9.
Number of image sections:                        4.
Image type:                                 PIC, SHAREABLE. Global section match = "LESS/EQUAL", G.S. Ident, Major=1, Minor=1
Map format:                               FULL in file "_DBB2:[HUSTVEDT.USS]USS.MAP;19"
Estimated map length:                     43. blocks
                                                              +---------------------+
                                                              ! Link Run Statistics !
                                                              +---------------------+

Performance Indicators                                 Page Faults   CPU Time        Elapsed Time
----------------------                                 -----------   --------        ------------
    Command processing:                                          9   00:00:00.08     00:00:00.12
    Pass 1:                                                     31   00:00:01.00     00:00:01.79
    Allocation/Relocation:                                       7   00:00:00.05     00:00:00.19
    Pass 2:                                                      3   00:00:00.22     00:00:00.72
    Map data after object module synopsis:                      17   00:00:00.25     00:00:00.70
    Symbol table output:                                         0   00:00:00.04     00:00:00.33
Total run values:                                               67   00:00:01.64     00:00:03.85

Using a working set limited to 200 pages and 14 pages of data storage (excluding image)

Total number object records read (both passes):   272
    of which 51 were in libraries and 2 were DEBUG data records containing 414 bytes

Number of modules extracted explicitly            = 0
    with 1 extracted to resolve undefined symbols

0 library searches were for symbols not in the library searched

A total of 4 global symbol table records was written

/PROTECT/NOSYSSHR/SHARE=USS/MAP=USS/FULL SYS$INPUT/OPTIONS

Ready
```

CHAPTER 7

# PROGRAM EXAMPLES

This chapter presents applications that use many of the features discussed in this manual. Each application is explained, and the program listings are given. The programs are in VAX-11 FORTRAN, although some routines are in VAX-11 MACRO.

The following applications are included in this chapter:

- An analog-to-digital (A/D) data acquisition and manipulation system

- An airline reservations system

## 7.1 DATA ACQUISITION AND MANIPULATION

This system, called LABIO, allows multiple users to receive and manipulate analog-to-digital (A/D) data in real time. In this example, a 16-channel A/D converter, such as the AD11-K, is shared by 1 to 16 independent users. This example demonstrates the real-time use of many VAX/VMS system services and features (described in Sections 7.1.2 and 7.1.3). However, because each real-time application is unique, this example does not show the only, or necessarily the most efficient, use of these features. It is meant only as a guideline for possible implementations.

### 7.1.1 Application Overview

In the LABIO system the 16-channel A/D converter is to be used independently by up to 16 users; that is, each user must be able to specify collection parameters and collect data from one or more A/D channels without conflicting with other users. This independence is achieved by placing a single "privileged" process (LABIO_DATA_ACQ) in control of the AD11-K.

The LABIO_DATA_ACQ process collects data from the AD11-K and stores the data in buffers in a shared data array. The process runs at a real-time priority and uses the VAX/VMS connect-to-interrupt capability to process interrupts from a dedicated KW11-K real-time clock. On every clock overflow, data from the AD11-K is taken and stored in the shared data array. The process uses control information stored in the shared data array to determine how much data is to be collected for each A/D channel. To protect users from other users (and from themselves), the shared data array is read-only for the users.

To store control information in the control block, each user communicates with a second "privileged" process, LABIO_CONNECT. The LABIO_CONNECT process receives, validates, and acknowledges each user request, and modifies the data base accordingly. Simultaneous requests from different users are serialized through the use of mailboxes. The mailbox that receives user requests has the logical name LABIO_CONNECT. Users can issue four types of request:

- CONNECT
- ALLOCATE
- DISCONNECT
- DEALLOCATE

The first user request must be CONNECT. This request makes the user known to the LABIO system. The user also passes the logical name of a mailbox, which the LABIO_CONNECT process will use to ackowledge the user's requests.

After a CONNECT request is completed, the user can issue ALLOCATE and DEALLOCATE requests. The ALLOCATE request is used to gain ownership of a specific A/D channel; once a channel is allocated by a user, no other users can allocate it until the owner specifies it in a DEALLOCATE request. Four parameters are associated with the ALLOCATE request:

- Channel number
- Sample rate
- Buffer size
- Buffer count (number of buffers to be acquired)

A user can allocate any number of A/D channels. The ALLOCATE request can also be used to change collection parameters for a channel a user already owns.

When finished with a channel, a user issues a DEALLOCATE request for the channel; and when finished altogether, a user issues a DISCONNECT request. The DISCONNECT request removes a user from the LABIO system and implicitly deallocates any channels still allocated to the user.

Once connected to the LABIO system and allocated channels, a user communicates with the data acquisition process (LABIO_DATA_ACQ) using event flags. Each channel has three flags associated with it:

- ACTIVITY flag
- NOTIFY flag
- STATUS flag

The ACTIVITY flag determines whether data collection is enabled (flag set by user) or disabled (flag cleared). The user process tells the LABIO_DATA_ACQ process to check the ACTIVITY flag by setting the NOTIFY flag; that is, when the NOTIFY flag is set, the LABIO_DATA_ACQ process checks the state of the corresponding ACTIVITY flag and enables or disables the channel. When a data buffer is ready for user processing, the LABIO_DATA_ACQ process sets the STATUS flag for the channel. When the user process detects that the STATUS flag is set, it clears the flag and processes the data buffer.

There is one utility program associated with the LABIO system: LABIO_STATUS, which displays the status of each of the A/D channels on a VT52-compatible video terminal.

### 7.1.2 LABIO System Details

The LABIO system uses a number of VAX/VMS features described in this manual. The following sections describe the major features illustrated in this system.

7.1.2.1 **Shared Data Base** - The processes share data by using global sections. The LABIO_DATA_ACQ process creates the global section using the Create and Map Section ($CRMPSC) system service. A VAX-11 MACRO routine (GBL_SECTION_UFO) is used to open the data file to be associated with the global section. This global section is read/write for processes with the same UIC (that is, LABIO_DATA_ACQ and LABIO_CONNECT), but read-only for other processes in the group (that is, the processes running the user programs). The global section is not accessible by any processes outside the group. Other processes map the global section using the Map Global Section ($MGBLSC) system service, specifying the global section name LABIO_COMMON.

Because global sections are mapped by pages, it is important to ensure that the data arrays are page aligned. To ensure this alignment, the VAX-11 FORTRAN named-common and block-data features are used with the VAX-11 Linker cluster option.

The shared data region contains three arrays:

- AD_BLOCK, containing 16 control blocks, one for each A/D channel
- CONNECT_BLOCK, containing 16 control blocks, one for each process that can be connected to the system (each process is identified by its process identification)
- DATA_BUFFER, the array into which the A/D data is stored

7.1.2.2 **Common Event Flag Clusters** - Two common event flag clusters are used in the LABIO system:

- LABIO_EF_NOTIFY, containing 16 NOTIFY flags
- LABIO_EF_STATUS, containing 16 ACTIVITY flags and 16 STATUS flags

The LABIO_DATA_ACQ process waits for the logical OR of the 16 NOTIFY flags; that is, the process is activated whenever any of the flags is set. Each user process normally waits for the logical OR of the STATUS flags for the channels it has allocated. Each user process must set and clear the ACTIVITY flags as appropriate, and must set the corresponding NOTIFY flag if it wants the LABIO_DATA_ACQ process to check the ACTIVITY flag. The LABIO_DATA_ACQ process sets the STATUS flag when a buffer is ready and stores the buffer index in AD_BLOCK. The user process is then responsible for clearing the STATUS flag.

7.1.2.3 **Mailboxes** - The LABIO_CONNECT process creates a mailbox with the logical name LABIO_CONNECT. All user processes write their requests to this mailbox. Each user process must also create a mailbox, and must specify the mailbox's logical name in the CONNECT request. If the LABIO_CONNECT process accepts the CONNECT request, it opens the user's mailbox and acknowledges the request by returning the user request line preceded by a 2-character code:

- Zero to indicate a positive acknowledgment
- Nonzero to indicate a negative acknowledgment (the specific code corresponds to the field containing the error)

7.1.2.4 **Connecting to an Interrupt Vector** - The actual analog-to-digital I/O is performed by an interrupt service routine specified in the connect-to-interrupt $QIO call. The process connects to the interrupt vector for the KW11-K real-time clock, which generates an interrupt every millisecond. On each interrupt, the interrupt service routine does the following for each active AD11-K channel (all control information is stored in AD_BLOCK):

1. Decrements the timer for the current channel
2. If the timer overflows, takes an A/D reading and stores the result in DATA_BUFFER
3. If the data buffer is full, switches to the next buffer
4. If the last buffer has been acquired, deactivates the channel

If any buffer was filled, an AST is requested and bits 0 to 15 of the AST parameter word are set to indicate those channels that had a buffer filled. The AST service routine SET_EF_AST sets the STATUS event flags corresponding to the channels that had buffers filled.

### 7.1.3 Typical LABIO User Program Logic

A typical program running in a user process in the LABIO system would contain the following logical steps:

1. Map the global section LABIO_COMMON
2. Associate with the common event flag clusters LABIO_EF_NOTIFY and LABIO_EF_STATUS
3. Open the mailbox LABIO_CONNECT
4. Create a mailbox to receive acknowledgments from the LABIO_CONNECT process
5. Issue a CONNECT request and wait for an acknowledgment
6. Allocate channels using ALLOCATE requests and wait for acknowledgments
7. Start data acquisition by setting the ACTIVITY and NOTIFY event flags
8. Wait for buffer(s) to be filled by waiting for STATUS event flags to be set

9. Process the contents of the buffers

10. Repeat steps 8 and 9 until finished

### 7.1.4 Program Listings

This section lists the files needed to create and use the laboratory data acquisition application. Three programs that make up the system and three sample programs that use the system are presented first, followed by modules used by all or some of the programs. The remaining files are used to activate the system and to compile and link the program.

The files are presented in the following order:

1. Three programs that make up the system. The modules in each program are as follows (LABIOCOM.FOR, listed later, is common to all three programs):

    a. LABIOACQ.FOR, GBLSECUFO.MAR, LABIOCIN.MAR

    b. LABIOCON.FOR

    c. LABIOSTAT.FOR

2. Three sample programs to use the system. The modules in each program are as follows (LABIOCOM.FOR, listed later, is common to all three programs):

    a. LABIOPEAK.FOR, PEAK.FOR

    b. LABIOSAMP.FOR

    c. TESTLABIO.FOR

3. Modules used by all or some programs

    a. LABIOCOM.FOR (common routines)

    b. LABMBXDEF.FOR (mailbox format)

    c. LABCHNDEF.FOR (common data structures)

    d. LABIOSEC.FOR (common data definitions)

4. Command procedures to activate the system

    a. CONNECT.COM

    b. LABIOSTRT.COM

5. Files to compile and link the programs

    a. LABIOCOMP.COM

    b. LABIOLINK.COM

    c. LABIO.OPT

    d. LABIOCIN.OPT

```
!File:  LABIOACQ.FOR
        Program LABIO_DATA_ACQ

! This is the program that acquires data for the LABIO system
! It uses the connect-to-interrupt feature of VMS to acquire
! via a user written I/O routine. The actual I/O routine is
! written in MACRO. The main program monitors the event flags
! and enables and disables data acquisition for each channel.
! It also notifies users via event flags when a buffer is full.


! Define the LABIO data base

        Include 'LABCHNDEF.FOR'

! Local Variables
        Logical*4 SECTION_FLAGS, SECTION_PROT

! System Services
        Logical*4 SYS$ASCEFC,SYS$MGBLSC,SYS$ASSIGN,SYS$QIO
        Logical*4 SYS$CLREF

! External constants
        External SEC$M_GBL,SEC$M_WRT,SS$_CREATED,SS$_WASSET
        External SET_EF_AST

! Misc.
        Logical*4 AD_CIN_UP,SUCCESS

!
! Create the Global Section for the data buffer
! This data buffer will be READ/WRITE for the owner, READ only  for the GRO
!
! First see if the global section already exists, if it
! does just map to it, and set the restart flag.
!
! If not, Open the Data File. This can not be opened
! via FORTRAN since we need the VMS channel number.
!
!
        SECTION(1) = %Loc( LABIO_BUFFER_S)       !Start address of section
        SECTION(2) = %Loc( LABIO_BUFFER_E) - 1   !End address
! Page count for the section
        SECTION_SIZE = ( SECTION(2) - SECTION(1) )/512 + 1

! FLAGS for Section are GLOBAL,SHARED,NON_ZEROED,READ/WRITE,TEMP
!
          SECTION_FLAGS = %Loc( SEC$M_GBL ) + %Loc( SEC$M_WRT )

! Try just mapping to the global section
        SUCCESS = SYS$MGBLSC( SECTION,,,%Val(SECTION_FLAGS),'LABIOCOMMON',,
        If( SUCCESS ) Then
          RESTART = .TRUE.       !Succes, this is a restart
        Else
          SUCCESS = GBL_SECTION_UFO( SECTION_SIZE, 'LABIO_SEC_FILE',
        1                          SECTION_CHANNEL )
          If( .not. SUCCESS )
        1   Call FATAL_ERROR(SUCCESS,'Opening Global Section File')
```

```
! PROTECTION is OWNER = READ/WRITE, GROUP = READ, SYSTEM/WORLD = none
!
          SECTION_PROT = 'F E 0 F'X !Protection for section

! Create and Map the Section
!
          SUCCESS = SYS$CRMPSC( SECTION,,,%Val(SECTION_FLAGS),'LABIOCOMMON',
        1           ,,%Val(SECTION_CHANNEL),%Val(SECTION_SIZE),,
        1           %Val(SECTION_PROT),%Val(SECTION_SIZE))
          If( .not. SUCCESS )
        1 Call FATAL_ERROR(SUCCESS,'Creating Global Section')
          RESTART = .FALSE.                      !We are not restarting
        End If
!
! If this is not a restart, clear the data structures
!
        If( .not. RESTART ) Then
         Do 32 I = 1, MAX_AD_CHANNEL             !Clear AD_BLOCK
          Do 30 J = 1, 16
30         AD_BLOCK(J,I) = 0
          Do 31 K = 1, BUFFER_COUNT              !Clear Data buffers
           Do 31 J = 1, MAX_BUF_SIZE
31          DATA_BUFFER(J,K,I) = 0
32       Continue
         Do 33 I = 1, MAX_PID
          Do 33 J = 1,2
33         CONNECT_BLOCK(I,J) = 0                !Clear Process connect block
        End IF
!
! Create event flag cluster EF_NOTIFY and associate with event flags 64-95
! These are used to notify the Data Acquisition process.

        SUCCESS = SYS$ASCEFC( %VAL(EF_NOTIFY_1),EF_NOTIFY_CLSTR,,)
        If ( .not. SUCCESS)
        1       Call FATAL_ERROR( SUCCESS, 'CREATING EVENT FLAG CLUSTER')
!
! Create event flag cluster EF_STATUS and associate with event flags 96-127
! These are used to notify and report the status of the user buffers
!
        SUCCESS = SYS$ASCEFC( %VAL(EF_STATUS_1),EF_STATUS_CLSTR,,)
        If ( .not. SUCCESS)
        1       Call FATAL_ERROR( SUCCESS, 'CREATING EVENT FLAG CLUSTER')

!
! Make sure that we can't be swapped
!
        Call SYS$SETSWM(%Val(1))

!
! Set-up the Connect-to-Interrupt
! First assign a VMS channel for the device
! Then call the connect-to-interrupt setup routine.
!
        SUCCESS = SYS$ASSIGN( 'LABIO_AD',CIN_CHANNEL,, )
        If ( .not. SUCCESS )
        1 Call FATAL_ERROR( SUCCESS, 'assigning A/D device' )

        SUCCESS = AD_CIN_SETUP( CIN_CHANNEL,SET_EF_AST )
        If( .not. SUCCESS )
        1 Call FATAL_ERROR( SUCCESS, 'connecting-to-interrupt')
!
```

```
! End Of Initialization, Notify other processes by setting EF_DATA_ACQ
!
        Call SYS$SETEF( %Val( EF_DATA_ACQ) )
!
! Wait for an event flag in the EF_NOTIFY cluster
! Then read the EF_NOTIFY CLUSTER and EF_STATUS_CLUSTER

10      Call SYS$WFLOR( %Val(EF_NOTIFY_1) , %Val('FFFF'X) )
!
! Look for the flag(s) set in EF_NOTIFY
! If the corresponding activity flag is set, activate the channel,
! otherwise deactivate it. Also check the buffer status flag, if clear
! clear the buffer index.
!
        Do 20 I = 1,16
        If( SYS$CLREF( %Val(EF_NOTIFY_OFF + I)) .eq. %Loc(SS$_WASSET)) Then
          If( AD_BLOCK(1,I) .ne. 0 ) Then
            If( SYS$READEF( %Val(EF_ACTIVITY_OFF + I),EF_STATE )
        1                         .eq. %Loc(SS$_WASSET ) ) Then
                AD_BLOCK(1,I) = ACTIVE
             Else
                AD_BLOCK(1,I) = INACTIVE
             End if
            If( SYS$READEF( %Val(EF_STATUS_OFF + I),EF_STATE )
        1                         .eq. %Loc(SS$_WASCLR))  AD_BLOCK(7,I) = 0
          End If
        End If
20      Continue
        Go To 10

        End
        Subroutine SET_EF_AST( EVENT_FLAGS )

! This is a AST routine which is invoked by the
! Interrupt service routine. This routine sets
! the event flags indicated by the ISR.

        Include 'LABCHNDEF.FOR'
        Integer EVENT_FLAGS

!
! The Event flags are set in cluster EF_STATUS_CLSTR
!

        Do 10 I = 1,16
        If( (EVENT_FLAGS .and. BIT(I)) .ne. 0 )
        1 Call SYS$SETEF( %Val(EF_STATUS_OFF + I) )
10      Continue
        Return

        End
![End of File]
```

```
.TITLE GBLSECUFO        Global Section UFO (User FIle Open)
;This routine opens a file to be used as a global section
;An RMS OPEN is performed with the file options (FOP) of
;User File Open (UFO). The calling routine specifies the
;file name and number of blocks; this routine returns the
;channel number on which the file was opened.
;If the specified file does not exist, the file is created
;
;The calling sequence is
;
;       Call GBL_SECTION_UFO( blkcnt,file-name,chan )
;
; Where
;               blkcnt => Number of blocks in the file
;               file-name => filename descriptor block
;               chan => channel opened
;
;
;
;Example:
;       Integer*4 CHANNEL
;          :
;          :
;       Call GBL_SECTION_UFO(10,'LABIO_DATA.DAT',CHANNEL )
.SBTTL  GBL_SEC_UFO

; RMS FAB for a $CREATE

GBLFAB: $FAB    FAC=PUT,-
                FOP=<UFO,CIF,CBT>

        NUM_ARG  = 3                    ;Number of arguments

        .ENTRY  GBL_SECTION_UFO,0

        MOVL    #SS$_INSFARG,R0         ;Assume bad arg count
        CMPB    (AP),#NUM_ARG           ;Check arg count
        BLSS    EXIT                    ;Too few

        MOVL    8(AP),R1                ;Get file name address string descriptor
        MOVB    (R1),GBLFAB+FAB$B_FNS   ;Store string length in FAB
        MOVL    4(R1),GBLFAB+FAB$L_FNA  ;And file name

        MOVL    @4(AP),GBLFAB+FAB$L_ALQ ;Number of blocks to allocate


        $CREATE FAB=GBLFAB              ;Open data file, Create it if
                                        ;if it does not exist
        MOVL    GBLFAB+FAB$L_STV,@12(AP);Store channel number

EXIT:   RET                             ;Return with error code in R)

        .END
```

```
;       KW_HIST = 1
        .TITLE  LABIO_CIN - LABIO Connect-to-Interrupt Module
        .IDENT  /V01/

;++
;
; FACILITY:
;
;       LABIO demonstation system
;
; ABSTRACT:
;
;       This module contains the I/O code for handling
;       an AD11-K, It is an example of a connect-to interrupt
;       routine, This module contains code to perform the following
;
;               The start I/O routine
;               The interrupt service routine
;               The cancel I/O routine
;
; AUTHOR:
;
;       P, Programmer   15-Nov-79
;
;--
        .SBTTL  DATA STRUCTURES

        .PSECT  LABIO_SECTION   PIC,OVR,REL,GBL,SHR,NOEXE,RD,WRT,LONG


; The following data structures are also defined by a
; FORTRAN INCLUDE file, These definitions must agree,


; AD_BLOCK       A/D Control Block

MAX_AD_CHANNEL = 16             ;Number of A/D channels
AD_BLOCK_SLOTS  = 16            ;number of entries in one block
AD_BLOCK_SIZE   = MAX_AD_CHANNEL*AD_BLOCK_SLOTS

;AD_BLOCK offsets (long words)

AD_STATUS       = 0             ;STATUS (Unknown, inactive, or active )
        ACTIVE_L= 2             ; ACTIVE
        INACTIVE_L = 1          ; INACTIVE
PID             = 4             ; PID of connected process
TICS_SAMPLE     = 8             ; Rate in tics/sample
BUFFER_SIZE     = 12            ; User specified buffer size
BUFFER_COUNT    = 16            ; User specified buffer count
BUFFER_ACQ      = 20            ; Number of buffers acquired
VALID_BUF_IND   = 24            ; Index of current valid data buffer
VALID_BUF_COUNT = 28            ; Number of data points in last buffer
CUR_BUF_IND     = 32            ; Index to current acq. buffer
CUR_BUF_COUNT   = 36            ; Number of data points in last buffer
TICS_REMAINING  = 40            ; Tics remaining to next sample
CUR_ACQ_OFF     = 44            ; Offset to acq point
AD_BLOCK_END    = 64            ; Offset to end of a block

AD_BLOCK:       .BLKL   AD_BLOCK_SIZE

; DATA_BUFFER   Data buffers for LABIO

MAX_BUF_COUNT   = 2             ;Number buffers/channel
MAX_BUF_SIZE   = 512            ;Maximum buffer size (WORDS)
```

```
BUFFER_END       = MAX_BUF_COUNT*MAX_BUF_SIZE*2 ; Size of one set of buffers
DATA_BUF_SIZE    = MAX_AD_CHANNEL*MAX_BUF_SIZE*MAX_BUF_COUNT
DATA_BUFFER:     .BLKW   DATA_BUF_SIZE

DATA_BUFFER_OFF = DATA_BUFFER-AD_BLOCK   ;Offset to data buffer from
                                         ;beginning of data structure

; CONNECT_BLOCK         Process Connect control block

MAX_PID = 16            ;Max number of processes connected

CONNECT_SIZE = MAX_PID*2

CONNECT_BLOCK:  .BLKL   CONNECT_SIZE


        .SBTTL  I/O DEVICES

;This section defines the constants asocciated with the KW11-K clock
;and the AD11-K A/D converter

;KW11-K Clock
;CSR bit assignments
KW11$M_GO = ^O1                 ;GO bit
KW11$M_RATE =  ^O2              ;Rate = bits 2-4
KW11$M_INTENB = ^O100           ;Interrupt enable
KW11$M_READY =  ^O200           ;Ready bit
KW11$M_REPINT = ^O400           ;repeateo interuupts

KW11_CSR_CONS = KW11$M_REPINT!KW11$M_INTENB!<1*KW11$M_RATE>
                                ;Repeated interrupts,interrupt enable
                                ;Rate = 1 MHz
KW11_PRESET = 1000.             ;Preset => Interrupt rate of 1 KHz

KW11_A_BUFFER = ^O2             ;Offset to clock A preset buffer
KW11_A_COUNTER = ^O24           ;Offset to clock A counter

;AD11-K A/D converter

AD11_OFFSET      = -4           ; Offset to the AD11 from thr KW11 clock CSR.
AD11_BUF         = 2            ; AD11 buffer offset from AD11 CSR

AD11_GO          = 1            ; Go bit
AD11_MUX_INCR    = ^O400        ; Mux incr bit
AD11_CSR_CONS    = AD11_GO      ; Initial CSR value

;Limit for stopping ISR loop
AD11_LOOP_LIMIT = AD11_MUX_INCR*<MAX_AD_CHANNEL-1>!AD11_CSR_CONS

        $IDBDEF                 ; Definition for I/O drivers
        $UCBDEF                 ; Data structurs
        $IODEF                  ; I/O function codes
        $CINDEF                 ; Connect-to-interrupt
        $CRBDEF                 ; CRB stuff
        $VECDEF                 ; more
        .SBTTL  LABIO_CIN_START, Start I/O routine

;++
; LABIO_CIN_START - Starts the KW11-K
;
; Functional description:
```

```
;
;       This routine starts the KW11-K
;               Rate = 1 Khz
;               Repeated interrupt
;
; Inputs:
;
;       0(R2)   - arg count of 4
;       4(R2)   - Address of the process buffer
;       8(R2)   - Address of the IRP (I/O request packet)
;       12(R2)  - Address of the device's CSR
;       16(R2)  - Address of the UCB (Unit control block)
;
; Outputs:
;       none
;
;       The routine must preserve all registers except R0-R2 and R4.
;
;--
        .PSECT LABIO_CIN

LABIO_CIN_START::
        MOVL    12(R2),R0                       ; Get address of the KW11 CSR
        CLRW    (R0)                            ; Clear the Clock

        MNEGW   #KW11_PRESET,-                  ; Preset count buffer
                KW11_A_BUFFER(R0)
        MOVW    #KW11_CSR_CONS+KW11$M_GO,(R0) ; Set the bits for
                                                ;   Repeated interrupt
                                                ;   Interrupt Enable
                                                ;   GO!

        MOVW    #SS$_NORMAL,R0                  ; Load a success code into R0.
        RSB                                     ; Return
        .SBTTL  LABIO_CIN_INTERRUPT, Interrupt service routine

;++
; LABIO_CIN_INTERRUPT
; Functional description:
;
;
; Inputs:
;
;       0(R2)   - arg count of 5
;       4(R2)   - Address of the process buffer
;       8(R2)   - Address of the AST parameter
;       12(R2)  - Address of the device's CSR
;       16(R2)  - Address of the IDB (interrupt dispatch block)
;       20(R2)  - Address of the UCB (Unit control block)
;
;
; Outputs:
;       Sets those bits in the AST parameter for those
;       channels who had a buffer filled
;
;       The routine must preserve all registers except R0-R4
;
;--
CIN_BUF_ADD = 4                                 ;Address of CIN buffer
AST_PARM = 8                                    ;Offset to AST permeter address
CIN_CSR_ADD = 12                                ;Address of CSR
```

```
;
AD_LOOP_DATA:
1$:     TSTB    (R4)                    ;Wait for A/D conversion
        BGEQ    1$                      ;
        .IF NDF KW_HIST                 ;Time histogram don't store actual data
        MOVW    AD11_BUF(R4),(R1)[R0]   ;store data point in buffer.
        .ENDC

;All done with this channel, setup for the next
;
AD_LOOP_NEXT:
        ADDL    #AD_BLOCK_END,R5        ;Next channel block
        ADDL    #BUFFER_END,R1          ;Next buffer
        ADDW    #AD11_MUX_INCR,R6       ;Incr A/D MUX
        AOBLSS  S^#MAX_AD_CHANNEL,R3,-  ;Next channel
                AD_LOOP                 ;Br if not done

;Exit routine - If any buffer overflowed, queue an AST
        MOVL    @AST_PARM(R2),R0        ;If any bit in the AST parameter
        BEQL    1$                      ;is set we must queue an AST
        MOVL    #1,R0                   ; 1 means queue the AST, 0 means don't
1$:     POPR    #^M<R5,R6>              ; Restore R5,R6
        RSB                             ;

        .SBTTL  LABIO_CIN_CANCEL, Cancel I/O routine

;++
; LABIO_CIN_CANCEL, Cancels an I/O operation in progress
;
; Functional description:
;
;       This routine turns off the KW11-K
;
; Inputs:
;       R5      -  Addr of the UCB
;
; Outputs:
;
;       The routine must preserve all registers except R0-R3.
;
;
;--

LABIO_CIN_CNCL::
        MOVL    UCB$L_CRB(R5),R0                ; Get Address of the CRB
        MOVL    CRB$L_INTD+VEC$L_IDB(R0),R0     ;Address of the IDB
        MOVL    IDB$L_CSR(R0),R0                ; Get addr of KW11
        CLRW    (R0)                            ; Turn of the KW11
        MOVW    #SS$_NORMAL,R0                  ; And return
        RSB
        .SBTTL  LABIO_CIN_END, End of Module

;++
; Label that marks the end of the module
;--

LABIO_CIN_END:                          ; Last location in module
.SBTTL  AD_CIN_SETUP    Set-up routine for LABIO connect-to-interrupt

;+
; This routine issues the QIO to connect to the AD11/KW11 interrupts.
```

```
LABIO_CIN_INT::
        PUSHR   #^M<R5,R6>              ;Service device interrupt, save R5,R6
        MOVL    CIN_CSR_ADD(R2),R4      ;Address of the KW11 CSR
        MOVL    CIN_BUF_ADD(R2),R5      ;Address of AD_BLOCK, control block
                                        ;for each A/D Channel
        MOVAL   DATA_BUFFER_OFF(R5),R1  ;Data Buffers
        MOVAL   AD11_OFFSET(R4),R4      ;Address of the AD11 CSR

        MOVW    #AD11_CSR_CONS,R6       ;AD11 CSR bits, GO bit on
        CLRL    @AST_PARM(R2)           ;Zero the AST parameter
        CLRL    R3
AD_LOOP:
        CMPL    (R5),S^#ACTIVE_L        ;Is this channel active?
        BLSS    AD_LOOP_NEXT            ;No, try next channel

        SOBGTR  TICS_REMAINING(R5),AD_LOOP_NEXT
                                        ;Decr the timer for this channel
                                        ;Br if no conversion required

        MOVW    R6,(R4)                 ;Start conversion, while that's going o
        .IF DF KW_HIST                  ;Time histogram, stored in data buffer
        MOVZWL  KW11_A_COUNTER-AD11_OFFSET(R4),R0 ;Get current clock contents
        ADDW    #KW11_PRESET,R0         ;Calc time from intgerrupt
        INCW    (R1)[R0]                ;Add one to that time bin
        .ENDC
; While the A/D is converting, the tic counter for this channel,
; get the offset to the data pointer, and update it. Take appropriate
; action if we have buffer overflow.

        MOVL    TICS_SAMPLE(R5),-       ;Reset timer for this channel
                TICS_REMAINING(R5)
        MOVL    CUR_ACQ_OFF(R5),R0      ;Get index to next data point
        INCL    CUR_ACQ_OFF(R5)         ;Advance it
        AOBLSS  BUFFER_SIZE(R5),-       ;Update current data count
                CUR_BUF_COUNT(R5),-     ;Br if no buffer overflow
                AD_LOOP_DATA
;Buffer overflowed, reset data pointer, reset buffer pointer
;increment acquired buffer count, termimate channel I/O if done

        MOVL    CUR_BUF_IND(R5),-       ;Valid data buf available for user
                VALID_BUF_IND(R5)
        MOVL    CUR_BUF_COUNT(R5),-     ;Number of points in buffer
                VALID_BUF_COUNT(R5)
        MULL3   CUR_BUF_IND(R5),-       ;Offset to next data point
                #MAX_BUF_SIZE,-
                CUR_ACQ_OFF(R5)
        CLRL    CUR_BUF_COUNT(R5)       ;Reset data count
        AOBLEQ  #MAX_BUF_COUNT,-        ;Next buffer index
                CUR_BUF_IND(R5),1$
        MOVL    #1,CUR_BUF_IND(R5)      ;Wrap-around, reset buffer index
        CLRL    CUR_ACQ_OFF(R5)         ;And buffer offset
1$:     INSV    #1,R3,#1,@AST_PARM(R2)  ;Set bit in AST parameter word
        AOBLSS  BUFFER_COUNT(R5),-      ;Incr buffer count
                BUFFER_ACQ(R5),2$       ;Done with all buffers?
        TSTL    BUFFER_COUNT(R5)        ;If original count was zero
        BEQL    2$                      ;Don't stop
        MOVL    #INACTIVE_L,(R5)        ;Deactivate channel
2$:

; Now, get the data point and store it in the buffer.
; It takes care of the internals associated with the connect-to-interrupt
; QIO. Input parameters  the VMS channel and the AST service routine address.
; The connect-to-interrupt QIO condition code is returned.

        .PSECT  AD_CIN_SETUP

AD_CIN_SETUP::
        .WORD   0
        MOVL    8(AP),USER_AST          ;Get the user AST routine addr
AD_CIN_QIO:
        $QIO_S  CHAN=@4(AP),-           ;Channel
                FUNC=#IO$_CONINTWRITE,- ;Allow writing to the data buffer
                IOSB=AD_CIN_IOSB,-      ;I/O status Block
                P1=AD_CIN_BUF_DESC,-    ;Buffer descriptor
                P2=#AD_CIN_ENTRY,-      ;Entry list
                P3=#AD_CIN_MASK,-       ;Status bits,etc
                P4=#AD_CIN_AST,-        ;AST service routine
                P6=#10                  ;preallocate some AST control blocks
        RET                             ;Return to caller

AD_CIN_BUF_DESC:                        ;Buffer descriptor for CIN
        .LONG   LABIO_CIN_END-AD_BLOCK  ;Size of buffer and CIN handler
        .LONG   AD_BLOCK                ;Address of buffer

AD_CIN_ENTRY:
        .LONG   0                       ;No init code
        .LONG   LABIO_CIN_START-AD_BLOCK;Start code
        .LONG   LABIO_CIN_INT-AD_BLOCK  ;Interrupt service routine
        .LONG   LABIO_CIN_CNCL-AD_BLOCK ;I/O cancel routine

AD_CIN_IOSB:
        .LONG   0,0                     ; I/O Status Block

; Control mask

AD_CIN_MASK = CIN$M_REPEAT!CIN$M_START!CIN$M_ISR!CIN$M_CANCEL

;
; AD_CIN_AST

; This AST routine calls the user AST routine. The user routine
; can not be called directly because the AST parameter itself
; not its address is returned via the connect-to-interrupt routine.
; This routine simply calls the  user routine with the ADDRESS of
; the AST parameter.

AD_CIN_AST::
        .WORD   0
        PUSHAL  4(AP)                   ;Get the AST parameter addr
        CALLS   #1,@USER_AST            ;Call the USER routine
        RET

USER_AST:
        .LONG                           ;Addr of the user AST routine

        .END
```

```
!File:  LABIOCON.FOR

        Program LABIO_CONNECT

! Define Labio data structures
        Include   'LABCHNDEF.FOR'

! Mailbox Definitions

        Include 'LABMBXDEF.FOR'          !Defines Mailbox Data Structures

! System Service Definitions

        Logical*4 SYS$CREMBX,SYS$ASSIGN
        Logical*4 SUCCESS
        External SS$_ENDOFFILE

! Subroutine Definitions

        Integer CONNECT,DISCONNECT,ABORT,ALLOCATE,DEALLOCATE
        Integer READ_MAILBOX,WRITE_MAILBOX,LABIO_LOG,ACKNOWLEDGE
        Integer CHECK_PID,RETURN_CODE


! Command Data Structures

        Parameter     MAX_COMMAND = 5
        Character*15 COMMAND,COMMAND_TABLE(MAX_COMMAND)
        Data COMMAND_TABLE     /'CONNECT',
        1                       'DISCONNECT',
        1                       'ABORT',
        1                       'ALLOCATE',
        1                       'DEALLOCATE'/


!
! Map to the Global Data Section 'LABIO_COMMON'
! And Define the Commom Event Flag CLusters
! Request write access to the data base.
!
        Call LABIO_INIT ( 1 )

!
! See if mailbox LABIO_CONNECT exists by attempting to assign it, if
! it does not exist, create it. This mailbox is used to communicate with
! other LABIO processes. Restrict  it to processes within this group.
!
        SUCCESS = SYS$ASSIGN('LABIO_CONNECT',MBX_CHANNEL,,)
        If (.not.  SUCCESS ) Then
          SUCCESS = SYS$CREMBX(,MBX_CHANNEL,,,%Val('FD00'x),,'LABIO_CONNECT
          If (.not. SUCCESS)
        1   Call FATAL_ERROR( SUCCESS, 'Creating mailbox')
        End If

!
! Tell other processes that we're ready to go.
!
        Call SYS$SETEF( %val( EF_CONNECT ) )

! Get a command from a requesting processes
```

```
!
10      Call READ_MAILBOX           !Get a message
        Call CONNECT_CHECK          !Check the database to clear
                                    !any deleted processes.
!
! If I/O status is EOF then process has terminated, ABORT it.
!
        If ( MBX_IO_STATUS .eq. %Loc(SS$_ENDOFFILE) ) Go To 23
!
! Decode characters as a command
!
        If ( MBX_MESSAGE_L .eq. 0 ) Go To 10
        Decode (MBX_MESSAGE_L,100,MBX_MESSAGE,ERR=10) COMMAND
!
! Search Command Table for Command
!

        Do 11 COMMAND_INDEX = 1,MAX_COMMAND
        If( COMMAND .eq. COMMAND_TABLE(COMMAND_INDEX) ) Go To 12
11      Continue

        Go To 13        !Illegal command
!
! Dispatch to correct routine
!

12      Go To (21,22,23,24,25) COMMAND_INDEX

!
! If we get here, it's an unknown command

13      Call LABIO_LOG(-1)

!
! CONNECT command
!

21      RETURN_CODE =  CONNECT (MBX_PID)
        Call ACKNOWLEDGE( RETURN_CODE )        !Acknowledge the request
        Call LABIO_LOG ( RETURN_CODE )         !Log the acknowledgement
!
! Disconnect if was bad connect
!
        If (RETURN_CODE .ne. 0 ) Call DISCONNECT(-1)
        Go To 10

!
! DISCONNECT Command
!

22      RETURN_CODE =  DISCONNECT  (MBX_PID)
        Call LABIO_LOG ( RETURN_CODE )          !Log the acknowledgement
        Go To 10

!
! ABORT command
!

23      RETURN_CODE =  ABORT  (MBX_PID)
        Go To 40
```

```
!
! ALLOCATE command
!

24      RETURN_CODE =  ALLOCATE (MBX_PID)
        Go To 40

!
! DEALLOCATE command
!

25      RETURN_CODE = DEALLOCATE (MBX_PID)
        Go To 40

!
! Return status in first character position
!

40      Call ACKNOWLEDGE( RETURN_CODE )         !Acknowledge the request
        Call LABIO_LOG( RETURN_CODE )           !Log the acknowledgement
        Go To 10

!
! Formats
!

100     Format (A)

        End
        Subroutine CONNECT_CHECK

! This routine checks to make sure all processes
! connected (in CONNECT_BLOCK) actually exist.
! If a process has been deleted, this routine
! removes it from the database by calling ABORT

        Include 'LABCHNDEF.FOR'

        Logical*4 SYS$GETJPI

        Do 10  I = 1, MAX_PID
        PID = CONNECT_BLOCK(I,1)
        If ( PID .ne. 0 ) Then
          If( .not. SYS$GETJPI(%Val(2),PID,,0,,,) ) Call ABORT( PID )
        End If
10      Continue

        Return

        End
        Logical*4 Function READ_MAILBOX
!
! This routine reads the LABIO_CONNECT mailbox
! Returns when a message is ready
!

        External IO$_READVBLK
        Include 'LABMBXDEF.FOR'
        Logical*4 SYS$QIOW,SUCCESS

!
```

```
! Read for a message from another process
!
        MBX_READ=%LOC(IO$_READVBLK)
        MBX_MESSAGE(1) = ' '

        READ_MAILBOX  = SYS$QIOW(,%Val(MBX_CHANNEL),%Val(MBX_READ),
        1         MBX_IO_STATUS,,,MBX_MESSAGE,
        1         %Val(MAX_MESSAGE),,,,)
        Return

        End
        Logical*4 Function WRITE_MAILBOX(MBX_CHAN,MESSAGE,MESSAGE_LENGTH)

! This routine writes a message to a mailbox
! Input are the MBX channel, the message, and message length
!

        External IO$_WRITEVBLK,IO$M_NOW
        Logical SYS$QIO
!
! Write response buffer of MBX
!

        MBX_WRITE =%Loc(IO$_WRITEVBLK)+%Loc(IO$M_NOW)

        WRITE_MAILBOX = SYS$QIO(,%Val(MBX_CHAN),%Val(MBX_WRITE),,,,
        1               MESSAGE,%Val(MESSAGE_LENGTH),,,,)

99      Return

        End
        Logical*4 Function OPEN_MAILBOX(MAILBOX_CHAN,MAILBOX_NAME)

! This routine opens mailbox indicated by MAILBOX_NAME. It returns
! the VMS channel number assigned to it. The mailbox name can be
! padded on the right with blanks.

        Character*(*) MAILBOX_NAME
        Integer   MAILBOX_CHAN
        Logical*4 SYS$ASSIGN,SUCCESS

!
!       Determine length of mailbox name string
!

        MAILBOX_NAME_L=Index(MAILBOX_NAME,' ')-1
        If (MAILBOX_NAME_L .lt. 0 ) MAILBOX_NAME_L=Len(MAILBOX_NAME)

!
!       Assign a channel to mailbox
!       Return status to caller
!
        OPEN_MAILBOX =SYS$ASSIGN(MAILBOX_NAME(:MAILBOX_NAME_L),MAILBOX_CHAN,,)

        Return

        End
        Subroutine ACKNOWLEDGE (ACK_CODE)
!
! This routine acknowledges a request of process, by return the
! command string the process sent us. The string is preceded
! an acknowledge code (ACK_CODE).  The acknowledgement is sent
! via the mailbox the the sending processes had created.
! If that process has not connected to us, we do nothing.


        Include 'LABCHNDEF.FOR'

        Logical*4 WRITE_MAILBOX

        Include 'LABMBXDEF.FOR'
        Integer CONNECT_INDEX,CHECK_PID,ACK_CODE
!
! If process is not in CONNECT_BLOCK, do not respond.
!
        CONNECT_INDEX = CHECK_PID(MBX_PID)

        If (CONNECT_INDEX .ne. 0 ) Then
          Encode( MBX_RESPONSE_L,100,MBX_RESPONSE) ACK_CODE
          MAILBOX = CONNECT_BLOCK(CONNECT_INDEX,2)
          Call WRITE_MAILBOX( MAILBOX, MBX_RESPONSE,
        1                      MBX_MESSAGE_L + MBX_RESPONSE_L )
        End If

        Return

100     Format ( I2 )

        End

        Subroutine LABIO_LOG( CODE )
!
! This routine logs a message that has been processed. The message
! is written to the log file, along with the time, process ID, IO status
! word and the message length. This routine opens the log file
! if it hasn't been opened.

        Include 'LABMBXDEF.FOR'

        Character*24 TIME
        Logical LOG_OPEN
        Integer CODE

        Data LOG_OPEN/.false./

        Call SYS$ASCTIM(,TIME,,)         !Get the date and time


!
! Open Log file if this is the first time thru
!
        If ( .not. LOG_OPEN ) Then
          Open (Unit = 1, Name='LABIO_LOG', Type='Unknown', Access = 'Appenc
          LOG_OPEN = .True.
          Write(1,100) TIME,' Labio Log Opened'
        End If

10      Write(1,200) TIME,MBX_PID,MBX_IO_STATUS,MBX_MESSAGE_L,
        1            CODE,(MBX_MESSAGE(I),I=1,MBX_MESSAGE_L)

        Return

100     Format( 2A )
```

```
200     Format( A,Z10,Z10,I10/I3,128A1 )

        End

        Integer Function CONNECT(REQ_PID)

        Include 'LABCHNDEF.FOR'

        Include 'LABMBXDEF.FOR'
        Character*63 MAILBOX_NAME

        Integer*4 REQ_PID,CHECK_PID
        Logical*4 OPEN_MAILBOX

        CONNECT = 1
!
! Find an empty CONNECT_BLOCK slot
!

        Do 10 I = 1, MAX_PID
        If ( CONNECT_BLOCK(I,1) .eq. 0 ) Go To 20
10      Continue

! We should never get here, since the last slot of
! the CONNECT_BLOCK is a spare for sending message
! disallowing a connect!

        Go To 99
!
! Open user specified MAILBOX
!

20      Decode (MBX_MESSAGE_L,100,MBX_MESSAGE) MAILBOX_NAME
        If( .not. OPEN_MAILBOX( MAILBOX_CHAN,MAILBOX_NAME) ) Go To 99

!
! Allocate the connect block. If it is not a duplicate
! PID, store the PID and mailbox channel in CONNECT_BLOCK
! If it is a duplicate, store the PID as -1.

        If( CHECK_PID(REQ_PID) .eq. 0 ) Then
          CONNECT_BLOCK(I,1) = REQ_PID
          CONNECT = 0
          Else
          CONNECT_BLOCK(I,1) = -1        !Duplicate PID! We will Disconnect
                                         !After Acknowledging request
        End If

        CONNECT_BLOCK(I,2) = MAILBOX_CHAN

        If ( I .ge. MAX_PID ) CONNECT = 1 !No room for process!

99      Return

100     Format(15X,A)

        End
        Integer Function DISCONNECT(REQ_PID)

! This routine disconnects a process from the LABIO system.
! If it is a valid process, all channels still allocated are
! deallocated, the request is acknowledged, the channel assigned
! to the mailbox is deassigned, and the CONNECT_BLOCK entry is removed.

        Include 'LABCHNDEF.FOR'
        Integer*4 REQ_PID,CHECK_PID

        DISCONNECT = 1
!
! Find index into connect block
!
        CONNECT_INDEX = CHECK_PID(REQ_PID)
        If (CONNECT_INDEX .eq. 0 ) Go To 99 !Not connected
!
! Deallocate all A/D channels
!
        Call DEALLOCATE_ALL(REQ_PID)
!
! Acknowledge DISCONNECT request
!
        Call ACKNOWLEDGE(0)
!
! Close the mailbox, and zero CONNECT_BLOCK
!
        Call SYS$DASSGN( %Val(CONNECT_BLOCK(CONNECT_INDEX,2)) )
        CONNECT_BLOCK(CONNECT_INDEX,1) = 0
        CONNECT_BLOCK(CONNECT_INDEX,2) = 0
        DISCONNECT =0

99      Return

        End



        Integer Function ABORT(REQ_PID)

        Call DISCONNECT( REQ_PID )

        Return

        End
        Integer Function ALLOCATE(REQ_PID)

! This routines allocates an A/D channel to a specific process.
! The process request a channels by number (1-16), specifing
! the asample rate in tics/sample, the buffer size in words, and
! the number of buffers to acquire ( 0 = infinity ). The user can
! default the rate to 1 tic/sample. Default the buffer size to
! the maximum, and the buffer count to 0. If the user reallocates
! the channel, the defaults are the previous values allocated.
! The channel must been INACTIVE if it is reallocated.


        Include 'LABCHNDEF.FOR'
        Include 'LABMBXDEF.FOR'

        Integer*4 REQ_PID        !PID of requesting process
        Integer*4 PARM(4)        !4 input parameters
        Integer*2 CONNECT_INDEX,CHECK_PID
        Integer*4 REQ_AD_CHAN,REQ_TICS,REQ_BUF_SIZE,REQ_BUF_COUNT
        Logical   CHECK_PARM
!
! Get index into CONNECT_BLOCK for REQ_PID
! If index is not > 0 , ignore request
!
        ALLOCATE = 1              !Checking first field

        CONNECT_INDEX = CHECK_PID(REQ_PID)
        If ( CONNECT_INDEX .le. 0 ) Go To 99 !Req, Proc not connected!
!
! Decode message into four fields
!
        Decode ( MBX_MESSAGE_L,100,MBX_MESSAGE) PARM

        REQ_AD_CHAN = PARM(1)     !Requested A/D channel is first parm
        REQ_TICS    = PARM(2)     !Tics/sample is 2nd
        REQ_BUF_SIZE= PARM(3)     !Buffer size is 3rd
        REQ_BUF_COUNT=PARM(4)     .Number of buffers is 4th

        ALLOCATE = 2              !Check next parameter (channel number)

! Valid channel numbers are 1-16

        If (REQ_AD_CHAN .lt. 1 .or. REQ_AD_CHAN .gt. 16) Go To 99

! Requested channel must not allocated, or
! allocated to the requesting process

        If ( AD_BLOCK(2,REQ_AD_CHAN) .ne. 0 .and.
        1    AD_BLOCK(2,REQ_AD_CHAN) .ne. REQ_PID ) Go To 99

! The channel must not be active
        If (AD_BLOCK(1,REQ_AD_CHAN) .gt. INACTIVE ) Go To 99

        ALLOCATE = 3              !Checking next parm (Tics/sample)

! Tics/sample must be between 1 and 2^31-1

        If( .not. CHECK_PARM(REQ_TICS,AD_BLOCK(3,REQ_AD_CHAN),
        1                    1,'7FFFFFFF'X,1) ) Go To 99

        ALLOCATE = 4    !Checking  parmeter (Buffer size)

!
! Buffer size between 1 and MAX_BUF_SIZE
!
        If( .not. CHECK_PARM(REQ_BUF_SIZE,AD_BLOCK(4,REQ_AD_CHAN),
        1                    1,MAX_BUF_SIZE,MAX_BUF_SIZE) ) Go To 99

        ALLOCATE = 5    ! Checking next parameter (number of buffers)

! Number of buffers to acquire must be between 1 and 2^31-1, or
! zero to indicate no limit

        If ( .not. CHECK_PARM(REQ_BUF_COUNT,AD_BLOCK(5,REQ_AD_CHAN),1,
        1                     '7FFFFFFF'x,0) ) Go To 99

        ALLOCATE = 0    !Everything is acceptable
!
! Enter info into AD_BLOCK
!
```

```
        AD_BLOCK(1,REQ_AD_CHAN) = 0             !Lock the data base
!
! Clear associated event flags
!
        Call SYS$CLREF(%Val( EF_NOTIFY_OFF + REQ_AD_CHAN ) )
        Call SYS$CLREF(%Val( EF_ACTIVITY_OFF + REQ_AD_CHAN) )
        Call SYS$CLREF(%Val( EF_STATUS_OFF + REQ_AD_CHAN ) )


        AD_BLOCK(2,REQ_AD_CHAN) = REQ_PID       !Requesting PID
        AD_BLOCK(3,REQ_AD_CHAN) = REQ_TICS      !Tics/sample
        AD_BLOCK(4,REQ_AD_CHAN) = REQ_BUF_SIZE  !Requested buffer size
        AD_BLOCK(5,REQ_AD_CHAN) = REQ_BUF_COUNT !Number of buffers to acqui
        AD_BLOCK(6,REQ_AD_CHAN) = 0             !No buffers acquired
        AD_BLOCK(7,REQ_AD_CHAN) = 0             !No data buffer available
        AD_BLOCK(8,REQ_AD_CHAN) = 0             !Number elements in last bu
        AD_BLOCK(9,REQ_AD_CHAN) = 1             !Current buffer index
        AD_BLOCK(10,REQ_AD_CHAN) = 0            !Current buffer count
        AD_BLOCK(11,REQ_AD_CHAN) = 1            !Tics remaining
        AD_BLOCK(12,REQ_AD_CHAN) = 0            !Offset to next data point
        AD_BLOCK(1,REQ_AD_CHAN) = INACTIVE      !Channel is inactive
        Return
!
!       Error return
!

99      Return          !Return to caller

100     Format(15X,4I)

        End
        Integer Function DEALLOCATE(REQ_PID)

! This routine deallocates a channel previously allocated by
! a process. The channel must be INACTIVE when deallocated.

        Include 'LABCHNDEF.FOR'
        Include 'LABMBXDEF.FOR'

        Integer*4 REQ_PID       !PID of requesting process
        Integer*2 CONNECT_INDEX,CHECK_PID
        Integer*4 REQ_AD_CHAN

! Get index into CONNECT_BLOCK for REQ_PID
! If index is not > 0 , ignore request

        DEALLOCATE = 1          !Checking first field

        CONNECT_INDEX = CHECK_PID(PID)
        If ( CONNECT_INDEX .le. 0 ) Go To 99

        DEALLOCATE = 2

        Decode (MBX_MESSAGE_L,100,MBX_MESSAGE) REQ_AD_CHAN

! Valid channel numbers are 1-16

        If (REQ_AD_CHAN .lt. 1 .or. REQ_AD_CHAN .gt. 16) Go To 99

! Does requesting process own the channel?
        DEALLOCATE = 21
        If (AD_BLOCK(2,REQ_AD_CHAN) .ne. REQ_PID ) Go To 99

! Is the channel inactive, clear the channel parameters
        DEALLOCATE = 22

        If ( AD_BLOCK(1,REQ_AD_CHAN) .ne. INACTIVE ) Go to 99

        Call AD_CANCEL(REQ_AD_CHAN)

        DEALLOCATE = 0          !Everything OK

        Return

!
! ERROR return
!

99      Return

!
! This entry point is used to deallocate all channels
! allocated to a specific process.

        Entry DEALLOCATE_ALL(REQ_PID)

        DEALLOCATE = 1

! Valid PID?

        CONNECT_INDEX = CHECK_PID(PID)
        If ( CONNECT_INDEX .ne. 0 ) Then
! Look for all A/D channels allocated to process
! and cancel all I/O unconditionally.
          Do 10 AD_CHAN = 1 , MAX_AD_CHANNEL
          If ( AD_BLOCK(2,AD_CHAN) .eq. REQ_PID ) Call AD_CANCEL(AD_CHAN)
10        Continue
          DEALLOCATE_ALL = 0
        End If

        Return
100     Format(15X,I15)
        End

        Integer*4 Function AD_CANCEL( CHANNEL )

! Clears the parameter table associated with A/D channel

        Include 'LABCHNDEF.FOR'
        Integer CHANNEL

        AD_CANCEL = 1           !Assume error
!
! Legal channel numbers are 1-16
!
        If ( CHANNEL .ge. 1 .and. CHANNEL .le. 16 ) Then
!
! Zero the AD_BLOCK for this channel
!
          Do 10  J = 1 , 16             !Clear everthing
10        AD_BLOCK(J, CHANNEL ) = 0
          AD_CANCEL = 0                 !Everything ok
        End IF
```

```
!
! Clear associated event flags
!
        Call SYS$CLREF(%Val( EF_NOTIFY_OFF + CHANNEL ) )
        Call SYS$CLREF(%Val( EF_ACTIVITY_OFF + CHANNEL ) )
        Call SYS$CLREF(%Val( EF_STATUS_OFF + CHANNEL ) )

99      Return

        End
        Logical Function CHECK_PARM(IVAL,OVAL,MIN,MAX,DEFAULT)

! This routine validates and defaults an input parameter (IVAL)
! If IVAL is not 0, it compares it to MIN and MAX, returning TRUE or FALSE.
! If IVAL is 0, and OVAL is not zero, IVAL = OVAL
! If IVAL is 0, and OVAL is zero, IVAL = DEFAULT

        Integer*4 IVAL,OVAL,MIN,MAX,DEFAULT

        CHECK_PARM = .false.     !assume the worst

        If (IVAL .ne. 0 ) Then
          If( IVAL .lt. MIN .or. IVAL .gt. MAX) Go To 99
        Else
          If (OVAL .ne. 0 ) Then
            IVAL = OVAL
          Else
            IVAL = DEFAULT
          End If
        End IF

        CHECK_PARM = .true.

99      Return

        END
        Integer Function CHECK_PID(PID)

! This routine checks to see if a PID is in CONNECT_BLOCK
! If it is, the INDEX into CONNECT_BLOCK is returned. If
! it isn't, 0 is returned

        Include 'LABCHNDEF.FOR'
        Integer*4 PID

! Assume PID is not in database
        CHECK_PID = 0

! If PID is found, return index.

        Do 10 I = 1 , MAX_PID
        If( CONNECT_BLOCK(I,1) .eq. PID ) CHECK_PID = I
10      Continue

        Return
        End
```

```
!File:  LABIOSTAT.FOR
        Program LABIO_STATUS
! This is a utility routine for the LABIO  system. It displays
! the status of all 16 channels of the A/D. It assumes that
! the terminal is a VT52 or an equivalent, e.g VT100 in VT52 mode.
! The display is update once every 1-9 seconds. Default is
! one second. There are 5 commands associated with the program
!
!       C - display status of 16 channels
!       P - display status by process PID
!       H - display help frame (timeouts after 1 min.)
!       E - Exit to VMS DCL
!       Digit(1-9) Change cycle time.
!
! The key pad can also be used to enter commands. The special function
! Keys on the VT52 or VT100 correspond to the first 4 commands (3 on VT52).
!
! Typing ANY key will cause a display refresh.


        Include 'LABCHNDEF.FOR'

        Character*10 STATUS(4)
        Character*8 XTIME
        Character*9 XDATE
        Parameter COMMAND_MAX = 4
        Character*1 COMMAND,COMMAND_TABLE(COMMAND_MAX,2),ESCAPE,TERMINATOR
        Character*63 COMMAND_DEV

        External SS$_NOTRAN,SS$_NORMAL,SS$_PARTESCAPE
        External IO$M_CVTLOW,IO$M_NOECHO,IO$M_TIMED,IO$_READVBLK,IO$M_PURGE

        Logical SUCCESS,SYS$QIOW,SYS$ASSIGN
        Integer CHANNEL,DISPLAY_FLAG,OLD_DISPLAY,COMMAND_CHAN
        Integer DEF_TIME_OUT,TIME_OUT
        Byte    ERASE_SCREEN(2),HOME(2),ERASE_LINE(2),VT52_MODE(7)
        Integer*2 IO_STATUS(4),CHAR_COUNT
        Equivalence (ESCAPE,HOME),(CHAR_COUNT,IO_STATUS(2))
!
! VT52 control ESCAPE Sequences
!
        Data HOME,ERASE_SCREEN,ERASE_LINE
        1   /'33'O,'H','33'O,'J','33'O,'K'/
!
! VT100 control ESCAPE sequences
! This ESC seq places a VT100 in VT52 mode
!
        Data VT52_MODE/'33'O,'[','?','2','1','33'O,']'/

        Data STATUS/'Unknown ','Inactive',' Active ',' '/
        Data COMMAND_TABLE/'C','P','E','H','P','Q','S','R'/
        Data DISPLAY_FLAG,ERASE_FLAG /1,.TRUE./
        Data DEF_TIME_OUT /1/
!
! Map to the GLOBAL DATA section created by the I/O program
!
        Call LABIO_INIT(0)
!
```

```
! Place VT100's in VT52 mode
!
        Type 500, VT52_MODE
!
! Initialize Command input channel
! We will read the command via a QIOW with a 1 sec timeout
! Commands are single character, to simplify matters we will
! read with no echo and convert lower to upper case.
!

        Call SYS$ASSIGN( 'TT',COMMAND_CHAN,,,)
        QIO_READ = %Loc(IO$M_NOECHO) + %Loc(IO$M_CVTLOW) + %Loc(IO$M_TIMED)
        1 + %Loc(IO$_READVBLK)
        TT_PURGE = %Loc(IO$M_PURGE)
        Go To 25                    ! Display Something
!
! Get a command from the user, but only wait a short time (TIME_OUT)
! so we can update the screen. The input buffer is purged if a command
! was decode on the last read. (Prevents unnecessary erase loops)
!
20      DISPLAY_FLAG = OLD_DISPLAY !Default is last display
        TIME_OUT = DEF_TIME_OUT    !Default time out

21      TABLE_INDEX = 1             !Assume no escape sequence

22      Call SYS$QIOW(,%Val(COMMAND_CHAN),%Val(QIO_READ+PURGE),
        1 IO_STATUS,,,%Ref(COMMAND),%Val(1),%Val(TIME_OUT),,,,)
        PURGE = 0

! If escape seq., set command table pointer to second table and
! get character following escape.
        TERMINATOR = Char( IO_STATUS(3) )
        If( TERMINATOR .ne. ESCAPE ) Go To 23
        TABLE_INDEX = 2
        Go To 22         !Get char following escape

23      If( CHAR_COUNT .ne. 0) Then      ! Char count not 0
! Check for char 1-9
          If( COMMAND .ge. '0' .and. COMMAND .le. '9' ) Then
            DEF_TIME_OUT = Ichar ( COMMAND )  - Ichar( '0' )
! Not 1-9 try a command.
          Else
            ERASE_FLAG = .true.          ! Screen erase
            Do 24 I = 1,COMMAND_MAX
            If( COMMAND .eq. COMMAND_TABLE(I,TABLE_INDEX)) DISPLAY_FLAG = I
24          Continue
          End If
          PURGE = TT_PURGE               !Purge the input buffer next time
        End If
!
! Get date and time, then dispatch to display routine
!
25      Call DATE (XDATE)
        Call TIME (XTIME)

        Go to (50,60,99,40) DISPLAY_FLAG
!
! Refresh the screen (Erase and Redisplay)
!
30      DISPLAY_FLAG = OLD_DISPLAY       !Redisplay last display
        ERASE_FLAG = .true.
        Go To 25
!
! Display the HELP frame, set the temporary time-out to 1 minute
!
40      Type 600, HOME,ERASE_SCREEN      !Display the help frame
        TIME_OUT = 60                    !Give the user 1 minute to read it
        DISPLAY_FLAG = OLD_DISPLAY       !When it times out, default old
        ERASE_FLAG = .true.
        Go To 21
!
! Generate the Status Line for each A/D channel
!
50      If ( ERASE_FLAG ) Type 300, HOME,ERASE_SCREEN
        Type 100, HOME,XTIME,XDATE
        CHANNEL_COUNT = 0
        Do 51 CHANNEL = 1,MAX_AD_CHANNEL
        If( AD_BLOCK(2,CHANNEL) .ne. 0 ) Then     !If allocated, display info
          Type 200,CHANNEL, STATUS(AD_BLOCK(1,CHANNEL)+1),
        1 (AD_BLOCK(J,CHANNEL), J = 2,6 )
          CHANNEL_COUNT = CHANNEL_COUNT + 1
        Else                                      !If not allocated, say so
          Type 900, CHANNEL,'<Unused>',ERASE_LINE
        End If
51      Continue
        PID_COUNT = 0
        Do 52 PID_INDEX = 1, MAX_PID
        PID = CONNECT_BLOCK(PID_INDEX,1)
        If ( PID .ne. 0 ) PID_COUNT = PID_COUNT + 1
52      Continue

        Type 400,ERASE_LINE, PID_COUNT,CHANNEL_COUNT
        OLD_DISPLAY = DISPLAY_FLAG
        ERASE_FLAG = .false.
        Go to 20
!
! Status display via process (PID)
!
60      If ( ERASE_FLAG ) Type 300, HOME,ERASE_SCREEN
        Type 100, HOME,XTIME,XDATE
        PID_COUNT = 0                    ! Number of connected processess
        CHANNEL_COUNT = 0                ! Number of allocated channels
        Do 61 PID_INDEX = 1, MAX_PID
        PID = CONNECT_BLOCK(PID_INDEX,1)
        If ( PID .ne. 0 ) Then
        PID_COUNT = PID_COUNT + 1
        OLD_COUNT = CHANNEL_COUNT
          Do 62 CHANNEL = 1, MAX_AD_CHANNEL
          If( AD_BLOCK( 2,CHANNEL) .eq. PID ) Then  !If right PID, display info
            Type 200, CHANNEL, STATUS(AD_BLOCK(1,CHANNEL)+1),
        1   (AD_BLOCK(J,CHANNEL), J = 2,6 )
            CHANNEL_COUNT = CHANNEL_COUNT + 1
          End IF
62        Continue
        If (OLD_COUNT .eq. CHANNEL_COUNT ) Type 800, '<None>',PID,ERASE_LINE
        End IF
61      Continue
        Type 400,ERASE_LINE,PID_COUNT,CHANNEL_COUNT,ERASE_SCREEN
        OLD_DISPLAY = DISPLAY_FLAG
        ERASE_FLAG = .false.
        Go to 20
```

```
!
! Exit
!

99      Call Exit

!
! Format Statments
!
100     Format(1X,2A1,'           Lab IO Status as of ',A,' ',A//
        1' Channel   Status       PID     Tics/Sample  Buffer Size
        1      Buffers  '/)

200     Format(I5,5x,A8,Z10,4I12)

300     Format(' ',4A1)

400     Format(' '2A1/' Totals: ',i2,' Processes connected   ',I2,' Channel
        1 allocated'/'                 <Type an H for help>'2A1$)

500     Format(' '7A1)

600     Format(' '4A1/
        1' The following commands are available:'//
        1'     VT100   VT52      any'/
        1'     ------  ----      ---'/
        1'      PF1    red        C    Channel Display'/
        1'      PF2    blue       P    Process Display'/
        1'      PF3    grey       H    Help Display'/
        1'      PF4     n/a       E    Exit'//
        1' To change display time, type a digit 0-9 for the desired time'//
700     Format(A)

800     Format(' ',A6,11X,Z10,2A1)

900     Format(I5,5x,A8,2A1)

        End
![End of File]
```

```
!File:  LABIOPEAK.FOR

        Program LABIO_PEAK
! This routine continuously samples channel #1 search for peaks.
! The sample rate is 1/TIC. It reports the PEAK height and position
! to logical channel 'LABIO_PEAK_DATA'

        Include 'LABCHNDEF.FOR'

        Parameter MBX_NAME = 'LABIO_PEAK'
        Character*130 RETURN
        Character*15 COMMAND
        Character*24 DATE_TIME
        Logical*4 SUCCESS,SYS$CREMBX

        Parameter AD_CHANNEL = 1                ! Channel Number
        Parameter AD_RATE = 1                   ! Rate
        Parameter AD_BUF_SIZE = 512             ! Buffer Size

        Parameter MAX_PEAKS = 10
        Integer*4 ITABLE(10),INLAST,INPTR,OUTPUT(2,MAX_PEAKS),IDIMO,NPEAKS
        Integer*2 INPUT(AD_BUF_SIZE*2)

        Data ITABLE/10*0/
        Data INLAST,INPTR,IDIMO,NPEAKS/0,0,MAX_PEAKS,0/
!
! Map To the Global Data Base and the event flags
!
        Call LABIO_INIT(0)
!
! Open Mailbox to LABIO_CONNECT
!
        Open ( Unit = 1, Name = 'LABIO_CONNECT' , Type = 'OLD' )
!
! Create Mailbox for response from LABIO_CONNECT
!
        SUCCESS = SYS$CREMBX(,MBX_CHANNEL,,,%Val('FD00'x),,MBX_NAME)
        If (.not.  SUCCESS )  Call FATAL_ERROR( SUCCESS, 'CREATING MAILBOX')
!
! Open via FORTRAN
!
        Open ( Unit = 2, Name = MBX_NAME, Type = 'OLD' )
!
! Deassign the channel assigned when we created it
!
        Call SYS$DASSGN( %Val(MBX_CHANNEL) )
!
! Open A Data File
!
        Open( Unit = 3, Name = 'LABIO_PEAK_DATA' ,Type = 'NEW' )
!
! Connect to the LABIO system
!
        COMMAND = 'CONNECT'
        Write(1,100) COMMAND,MBX_NAME
!
! Wait for Response from LABIO system
!
        Read(2,200) RETURN_CODE,RETURN
        If( RETURN_CODE .ne. 0 ) Go To 99       !Failed to connect!
!
! Allocate Channel AD_CHANNEL
!          Rate  = AD_RATE
!          Buffer size = AD_BUF_SIZE

        COMMAND = 'ALLOCATE'
        Write(1,400) COMMAND,AD_CHANNEL,AD_RATE,AD_BUF_SIZE,0
        Read(2,200) RETURN_CODE,RETURN
        If( RETURN_CODE .ne. 0 ) Go To 99       !Failed to allocate!


! Enable data acqusition by setting event flag ACTIVITY and NOTIFY
!
        Call SYS$SETEF(%Val(EF_ACTIVITY_OFF+AD_CHANNEL))
        Call SYS$SETEF(%Val(EF_NOTIFY_OFF+AD_CHANNEL))

! Now, wait for buffer to be filled, event flag STATUS will be set
! when data are ready

5       Call SYS$WAITFR( %Val(EF_STATUS_OFF+AD_CHANNEL) )

! Buffer is filled, get the buffer index
!
        INDEX = AD_BLOCK(7,AD_CHANNEL)

! Move data from data buffer to peak processing buffer

        Do 10 I = 1, AD_BUF_SIZE
10      INPUT(I+INLAST) = DATA_BUFFER(I,INDEX,AD_CHANNEL)
        INLAST = INLAST + AD_BUF_SIZE

! Clear the STATUS event flag and notify the I/O process
!
        Call SYS$CLREF( %Val(EF_STATUS_OFF+AD_CHANNEL) )
!(DEBUG) only
!       Write (3,600) (DATA_BUFFER(I,INDEX,AD_CHANNEL),I=1,AD_BUF_SIZE)
!
! Call the peak processing routine
!
15      Call PEAK(ITABLE,INPUT,INLAST,INPTR,OUTPUT,MAX_PEAKS,NPEAKS)

! Report the peak info

        PEAK_SWITCH = NPEAKS            !Remember the peak switch

        If( NPEAKS .ne. 0 ) Then        !We have some peaks
          If( NPEAKS .lt. 0 ) NPEAKS = MAX_PEAKS !WE have the max
          Do 20 I = 1, NPEAKS
           TOTAL_PEAKS = TOTAL_PEAKS + 1 !One more
20         Write(3,500) TOTAL_PEAKS,(OUTPUT(J,I), J = 1,2)
        End If

        NPEAKS = 0                      !Reset the pointer
        If( PEAK_SWITCH .lt. 0 ) Go To 15 !More peaks to find

! Move any unprocessed data to the beginning of the input array

        If ( (INPTR .gt. 0) .and. (INPTR .lt. INLAST) ) Then
          Do 30 I = 1, INLAST-INPTR
30        INPUT(I) = INPUT( INPTR+I )   !Move the data
          INLAST = I                    !Last element stored
        Else
          INLAST = 0
        End If

        INPTR = 0                       !Last element processed


! Go wait for more data
        Go To 5

! All done, Call the exit routine

99      Call EXIT(1)            !Exit

100     Format(' ',A,A)
200     Format(I2,A)
400     Format(' ',A,4I)
```

```
500     Format(3I10)
600     Format(I5)
        End
![End of File]




!File PEAK.FOR

        Subroutine PEAK(ITABLE,INPUT,INLAST,INPTR,OUTPUT,IDIMO,NPEAKS)
!A trivial peak-picking routine. The calling sequence is patterned
!after the LSPLIB routine PEAK.

        Integer*4 ITABLE(10),OUTPUT(2,IDIMO),INLAST,INPTR,IDIM0,NPEAK
        Integer*2 INPUT(1)
        Parameter NOISE = 5     !Noise value = 5 A/D units

!Initialize some parameters, if necesary
        If( NPEAKS .lt. 0 ) NPEAKS = 0
        If( INPTR  .lt. 0 ) INPTR  = 0

!First time thru?
        If( INPTR  .lt. INLAST .and. ITABLE(1) .eq. 0 ) Then
          INPTR = INPTR + 1
          ITABLE(1) = 1                 !Assume we're rising
          ITABLE(2) = 1                 !first point
          ITABLE(3) = INPUT(INPTR)
        End If
!Any data to process?
        If(INPTR .lt. INLAST ) Then
          Do 10 I = INPTR+1, INLAST
           If( ITABLE(1) .gt. 0 ) Then  !We're rising, look for a fall
            If( INPUT(I) .lt. ITABLE(3)-NOISE ) Then !We found a peak
                If( NPEAKS .lt. IDIMO ) Then !Any room to store it?
                  NPEAKS = NPEAKS + 1
                  OUTPUT(1,NPEAKS) = ITABLE(3)
                  OUTPUT(2,NPEAKS) = ITABLE(2)
                  ITABLE(1) = -1
                Else                            !No, tell user
                  INPTR = I - 1
                  NPEAKS = -IDIMO
                  Return
                End If
            End If
           Else                         !We're falling, see if we found a valley
            If( INPUT(I) .gt. ITABLE(3)+NOISE )    ITABLE(1) = 1
           End If
           ITABLE(3) = INPUT(I)
10         ITABLE(2) = ITABLE(2) + 1
        End If

        INPTR = -1                      !Normal exit all data processed.
        Return

        End
```

```
!File:  LABIOSAMP.FOR

        Program LABIO_SAMPLE
!
! This program samples channel #2 once every 10 seconds.
! It acquires 10 points at 1/tic, averages them and then
! Reports the date, time, and average value on logcial device
! LABIO_SAMPLE_DATA

        Include 'LABCHNDEF.FOR'

        Parameter MBX_NAME = 'LABIO_SAMPLE'
        Character*130 RETURN
        Character*15 COMMAND
        Character*24 DATE_TIME
        Logical*4 SUCCESS,SYS$CREMBX
        Integer*4 DELTA_TIME(2),NEXT_TIME(2)
        Integer*4 AVERAGE

        Parameter AD_CHANNEL = 2                        ! Channel
        Parameter AD_RATE = 1                           !
        Parameter AD_BUF_SIZE = 10
        Parameter SAMPLE_RATE = '0 0:0:10'
        Parameter MAX_SAMPLE = 10 000                   ! Maximum # samples

!
! Map To the Global Data Base and the event flags

!
        Call LABIO_INIT(0)
!
! Open Mailbox to LABIO_CONNECT
!
        Open ( Unit = 1, Name = 'LABIO_CONNECT' , Type = 'OLD' )
!
! Create Mailbox for response from LABIO_CONNECT
!
        SUCCESS = SYS$CREMBX(,MBX_CHANNEL,,,%Val('FD00'x),,MBX_NAME)
        If (.not.  SUCCESS )  Call FATAL_ERROR( SUCCESS, 'CREATING MAILBOX')
!
! Open via FORTRAN
!
        Open ( Unit = 2, Name = MBX_NAME, Type = 'Old' )
!
! Deassign the channel assigned when we created it
!
        Call SYS$DASSGN( %Val(MBX_CHANNEL) )
!
! Open A Data File
!
        Open( Unit = 3, Name = 'LAB_SAMPLE_DATA', Type = 'New' )
!
! Connect to the LABIO system
!
        COMMAND = 'CONNECT'
        Write(1,100) COMMAND,MBX_NAME
!
! Wait for Response from LABIO system
```

```
!
        Read(2,200) RETURN_CODE,RETURN
        If( RETURN_CODE .ne. 0 ) Go To 99       !Failed to connect!
!
! Allocate Channel AD_CHANNEL
!           Rate   = AD_RATE
!           Buffer size = AD_BUF_SIZE
!           Collect 1 buffer at a time

        COMMAND = 'ALLOCATE'
        Write(1,400) COMMAND,AD_CHANNEL,AD_RATE,AD_BUF_SIZE,1
        If( RETURN_CODE .ne. 0 ) Go To 99       !Failed to allocate!

!
! Every SAMPLE_RATE secs. we will collect one buffer of data
!

! Convert ASCII delta time to binary
        Call SYS$BINTIM( SAMPLE_RATE, DELTA_TIME )

! Schedule wake-ups every delt time interval
! But first cancel any previous wake-ups
        Call SYS$CANWAK(,)
        Call SYS$SCHDWK(,,DELTA_TIME,DELTA_TIME)

! Wait for scheduled time interval
10      Call SYS$HIBER()

! Enable data acqusition by setting event flag ACTIVITY and NOTIFY
!
        Call SYS$SETEF(%Val(EF_ACTIVITY_OFF+AD_CHANNEL))
        Call SYS$SETEF(%Val(EF_NOTIFY_OFF+AD_CHANNEL))
        Call SYS$ASCTIM(,DATE_TIME,,)

! Now, wait for buffer to be filled, event flag STATUS will be set
! when data are ready
        Call SYS$WAITFR( %Val(EF_STATUS_OFF+AD_CHANNEL) )

! Buffer is filled, get the buffer index
        INDEX = AD_BLOCK(7,AD_CHANNEL)

! Clear the STATUS event flag and notify the I/O process
        Call SYS$CLREF( %Val(EF_STATUS_OFF+AD_CHANNEL) )
        Call SYS$SETEF( %Val(EF_NOTIFY_OFF+AD_CHANNEL) )
! Average the points
        AVERAGE = 0
        Do 20 I = 1, AD_BUF_SIZE
20      AVERAGE = AVERAGE + DATA_BUFFER(I,INDEX,AD_CHANNEL)
        AVERAGE = AVERAGE/AD_BUF_SIZE

! Write out average with the acq. date/time
        Write(3,400) DATE_TIME,AVERAGE

! If we're all done, close files and exit
        If( AD_BLOCK(6,AD_CHANNEL) .lt. MAX_SAMPLE ) Go To 10

! All done, Call the exit routine

99      Call EXIT(1)                !Exit

100     Format(' ',A,A)
```

```
200     Format(I2,A)
400     Format(' ',A,4I)
        End
![End of File]




!File:  TESTLABIO.FOR
!
! Tests the LABIO system by allocating upto 16 channels
! Enter the number of channels, rate, and buffer size

        Program TEST_LABIO

        Include 'LABCHNDEF.FOR'

        Parameter MBX_NAME = 'TEST_LABIO2'
        Character*130 RETURN
        Character*15 COMMAND
        Character*24 DATE_TIME
        Logical*4 SUCCESS,SYS$CREMBX
        Integer*4 TEST_CHAN,TEST_RATE,TEST_BUF_SIZE

!
! Map To the Global Data Base and the event flags
!
        Call LABIO_INIT(0)
!
! Open Mailbox to LABIO_CONNECT
!
        Open ( Unit = 1, Name = 'LABIO_CONNECT' , Type = 'OLD' )
!
! Create Mailbox for response from LABIO_CONNECT
!
        SUCCESS = SYS$CREMBX(,MBX_CHANNEL,,,%Val('FD00'x),,MBX_NAME)
        If (.not.  SUCCESS )  Call FATAL_ERROR( SUCCESS, 'CREATING MAILBOX'
!
! Open via FORTRAN
!
        Open ( Unit = 2, Name = MBX_NAME, Type = 'OLD' )
!
! Deassign the channel assigned when we created it
!
        Call SYS$DASSGN( %Val(MBX_CHANNEL) )
!
! Connect to the LABIO system
!
        COMMAND = 'CONNECT'
        Write(1,100) COMMAND,MBX_NAME
!
! Wait for Response from LABIO system
!
        Read(2,200) RETURN_CODE,RETURN
        If( RETURN_CODE .ne. 0 ) Go To 99        !Failed to connect!
!
! Get parameters from operator
!
10      LAST_TEST_CHAN=TEST_CHAN
```

```
        Type 600,' Enter number of channels, rate(in tics), and buffer size'
        Accept 700, TEST_CHAN,TEST_RATE,TEST_BUF_SIZE
        If ( TEST_CHAN .eq. 0 ) CAll Exit(1)
!
! Deallocate Channels from last time
!
        Do 20 AD_CHANNEL=1,LAST_TEST_CHAN

        Call SYS$CLREF(%Val(EF_ACTIVITY_OFF+AD_CHANNEL)) !Stop Acq.
        Call SYS$SETEF(%Val(EF_NOTIFY_OFF+AD_CHANNEL))

        COMMAND = 'DEALLOCATE'
        Write(1,400) COMMAND,AD_CHANNEL
        Read(2,200) RETURN_CODE,RETURN
        If( RETURN_CODE .ne. 0 )
        1  Type 500, ' Deallocation failure',RETURN_CODE,RETURN
20      Continue
!
! Allocate Channels
!
        Do 30 AD_CHANNEL=1,TEST_CHAN

        COMMAND = 'ALLOCATE'
        Write(1,400) COMMAND,AD_CHANNEL,TEST_RATE,TEST_BUF_SIZE,0
        Read(2,200) RETURN_CODE,RETURN
        If( RETURN_CODE .ne. 0 )
        1  Type 500, ' Allocation failure',RETURN_CODE,RETURN


! Enable data acqusition by setting event flag ACTIVITY and NOTIFY
!
        Call SYS$SETEF(%Val(EF_ACTIVITY_OFF+AD_CHANNEL))
30      Call SYS$SETEF(%Val(EF_NOTIFY_OFF+AD_CHANNEL))
        Go To 10
!
! Connect Failure
!
99      Type 500, ' Connect failure',RETURN_CODE,RETURN
        Go To 10

100     Format(' ',A,A)
200     Format(I2,A)
400     Format(' ',A,4I)
500     Format(A/' ',I2,A)
600     Format(A)
700     Format(3I10)
        End







!File:  LABIOCOM.FOR
        Logical Function LABIO_INIT( PRIVILEGE )
!
! This routine is used to attach a LABIO user program to the
! LABIO system. It associated the two event flag clusters and
! maps to the LABIO global data section.
!
! INPUT:         PRIVILEGE - Privileged LABIO users can set this
!                  to 1 to allow write access to the data base.
!                  All others must set this to 0.
!
! OUTPUT:          None - Currently will always return with success code.
!                  If an error occurs, FATALERR is called to display
!                  the error messages and STOP THE PROCESS!
!
!

      Include 'LABCHNDEF.FOR'
      Logical*4 SYS$ASCEFC,SYS$MGBLSC,SUCCESS,SYS$WAITFR
      External SEC$M_WRT

!
! Create event flag cluster EF_NOTIFY and associate with event flags 64-95
! These are used to notify the Data Acquisition process.

      SUCCESS = SYS$ASCEFC( %VAL(EF_NOTIFY_1),EF_NOTIFY_CLSTR,,)
      If ( .not. SUCCESS)
      1      Call FATAL_ERROR( SUCCESS, 'CREATING EVENT FLAG CLUSTER')
!
! Create event flag cluster EF_STATUS and associate with event flags 96-12'
! These are used to notify and report the status of the user buffers
!
      SUCCESS = SYS$ASCEFC( %VAL(EF_STATUS_1),EF_STATUS_CLSTR,,)
      If ( .not. SUCCESS)
      1      Call FATAL_ERROR( SUCCESS, 'CREATING EVENT FLAG CLUSTER')

!
! Map to the GLOBAL DATA section created by the I/O program
! wait for event flag EF_CONNECT (non-privileged) or
! EF_DATA_ACQ  (privileged) before attempting mappimg.

      SECTION(1) = %Loc(LABIO_BUFFER_S)
      SECTION(2) = %Loc(LABIO_BUFFER_E) - 1

      SECTION_FLAGS = 0                 !Default flags

      If( PRIVILEGE .ne. 0 ) Then
        SECTION_FLAGS=%Loc(SEC$M_WRT)
        Call SYS$WAITFR( %Val( EF_DATA_ACQ ) )
      Else
        Call SYS$WAITFR( %Val( EF_CONNECT ) )
      End If

      SUCCESS = SYS$MGBLSC( SECTION,,,%Val(SECTION_FLAGS),'LABIOCOMMON',,
      If( .not. SUCCESS ) Call FATAL_ERROR(SUCCESS,'mapping data section'

      LABIO_INIT = SUCCESS

      Return

      End
!     FATAL_ERROR  -  FATAL ERROR HANDLER
!
!     This routine is used to report a fatal error and exit the image
!
!     INPUT:  ERROR_CODE - SYSTEM ERROR CODE TO REPORT
!             ERROR_MESSAGE - ERROR MESSAGE TO BE PRINTED
!
!     OUTPUT: NONE
```

```
!
!       >>>> THIS ROUTINE DOES NOT RETURN <<<<<
!
!       FUNCTION: TYPEs the message
!
!                'process name-FATAL ERROR - error_message'
!
!               Then prints system message corresponding to  ERROR_CODE
!
!
!               Finally, exits image by calling LIB$STOP
!
        Subroutine FATAL_ERROR(error_code,error_message)

        Integer*4 ERROR_CODE
        Character ERROR_MESSAGE*(*)
        Logical*4 SUCCESS,SYS$CREMBX,SYS$GETJPI
        Integer*2 JPI2(8),PROCESS_NAME_L
        Integer*4 JPI4(4)
        Character*15 PROCESS_NAME
        Equivalence (JPI2,JPI4)
        Parameter JPI$_PRCNAM='31C'X
!
!       Get the process name
!
        JPI2(1) =  15                   !Number of elements in name
        JPI2(2) = JPI$_PRCNAM           !Want process name
        JPI4(2) = %Loc(PROCESS_NAME)    !Address of process name
        JPI4(3) = %Loc(PROCESS_NAME_L)  !Address of process name length
        JPI4(4) = 0                     !Terminate list

        Call SYS$GETJPI(,,,JPI4,,,)
!
! Print the process name and error message
!
        Type 100, PROCESS_NAME(1:PROCESS_NAME_L),ERROR_MESSAGE
!
! Print the error message corresponding to ERROR_CODE and exit
!
        Call LIB$STOP( %Val(ERROR_CODE) )

100     Format(' 'A,' - FATAL ERROR ',A)

        Stop

        END
![End of File]
```

```
!File:  LABMBXDEF.FOR
!Define mailbox block for LAB_IO

        Parameter MAX_MESSAGE = 128                 !Maximum message length
        Parameter MBX_RESPONSE_L = 2                !Response Length
        Parameter MBX_ACK_L = MAX_MESSAGE+MBX_RESPONSE_L

        Integer*2 MBX_IO_STATUS, MBX_MESSAGE_L
        Integer*4 MBX_PID
        Byte MBX_RESPONSE(MBX_RESPONSE_L)
        Byte MBX_MESSAGE(MAX_MESSAGE)

        Common /MBX_BLOCK/ MBX_CHANNEL, MBX_IO_STATUS, MBX_MESSAGE_L,
        1                  MBX_PID, MBX_RESPONSE, MBX_MESSAGE

!                 > MBX_BLOCK <
!
!       __________________________________
!       !            MBX_CHANNEL          ! Word 1-2
!       ----------------------------------
!       ! MBX_MESSAGE_L   ! MBX_IO_STATUS ! Word 3-4
!       ----------------------------------
!       !              MBX_PID            ! Word 5-6
!       ----------------------------------
!       !           MBX_RESPONSE          ! Word 7-8
!       ----------------------------------
!       !                                 !
!       !                                 !
!       !           MBX_MESSAGE           ! Word 9-MAX_MESSAGE+8
!       !                                 !
!       !                                 !
!       ----------------------------------
!
![End of File]




!File:  LABCHNDEF.FOR
!
        Implicit Integer (A-Z)


!AD_CHANNEL STATUS BLOCK defined the parameters associated
!with each A/D channel
!
!For each A/D channel:
! 1)    Status of the channel (ACTIVE or INACTIVE)
! 2)    PID of the connected process allocated the channel
! 3)    Tics/sample (time between sample in tics)
! 4)    Buffer size in words
! 5)    Buffer count (0 if no limit)
! 6)    Buffers acquired
! 7)    Index to the last full buffer containing valid data
!               0 => No buffer available
! 8)    Number of data points in the last full buffer

! The following elements are used by the data acquisition interrupt service
! routine. In general, they will not be used by an application process.
!
! 9)    Index to the current data acquisition buffer
! 10)   Number of data points in the current data acquisition buffer
! 11)   Number of tics until the next sample
! 12)   Offset to the next data point to be acquired (wrst buffer #1)
!       (NOTE: Offset = Index - 1 )

        Parameter      MAX_AD_CHANNEL = 16     !Maximum number of channels
        Parameter      MAX_BUF_SIZE = 512      !Maximum buffer size
        Parameter      INACTIVE = 1            !Status values for AD_BLOCK
        Parameter       ACTIVE = 2              !

        Integer*4       AD_BLOCK(MAX_AD_CHANNEL,16)

!
! Data buffers
!
        Parameter       BUFFER_COUNT = 2        ! Number of buffers/channel

        Integer*2 DATA_BUFFER(MAX_BUF_SIZE,BUFFER_COUNT,MAX_AD_CHANNEL)


!This module defines the common data structures
!for the privileged LABIO processes.

!CONNECT BLOCK used to identify processes currently
!connected to the LABIO process.
!
!For each process CONNECT_BLOCK contains:
!       Process ID (PID)
!       Internal VMS I/O channel of the connected processes mailbox

        Parameter       MAX_PID = 16            !Maximum number of processes
        Integer*4       CONNECT_BLOCK(MAX_PID,2)




!
!
! DATA COMMON SECTION
! This will be mapped as a global data section
!

        Common  /LABIO_SECTION/ AD_BLOCK, DATA_BUFFER, CONNECT_BLOCK
        Common  /LABIO_SECTION/ LABIO_BUFFER_E !Last element of DATA section
        Equivalence (AD_BLOCK,LABIO_BUFFER_S)  !First element of DATA section
        Integer*4 SECTION(2),SECTION_SIZE



!
! Define Global Event Flag Cluster names and numbers
!
! EF_NOTIFY_CLUSTER is used to notify the priveleged LABIO process
! that change of status has occured, i.e. channel has
! become ACTIVE or INACTIVE, or a buffer has been freed.
! Flags Ø-15 of the cluster correspond to CHANNELS 1-16
! Flags 16-31 are not used.

        Parameter EF_NOTIFY_CLSTR = 'LABIO_EF_NOTIFY'
! First flag of notify
        Parameter EF_NOTIFY_1 = 64
! Offset to Notify
        Parameter EF_NOTIFY_OFF = 63
! Event Flag EF_DATA_ACQ is set when LABIO_DATA_ACQ has completed initialization
        Parameter EF_DATA_ACQ = EF_NOTIFY_1+17
```

```
! Event Flag EF_CONNECT is set when LABIO_CONNECT has completed initialization
        Parameter EF_CONNECT = EF_NOTIFY_1+18

! EF_STATUS is used to notify a applications process
! that a buffer is available, and used by an application
! process to inicate the status (ACTIVE or INACTIVE) of
! a channel.
!
! Flags 0-15 of the cluster are the ACTIVITY flags
! if set (by the application process), the corresponding
! channel(1-16) is active. If clear, the channel is inactive.
! When a change of state is made the corresponding flag must
! also be set in Cluster EF_NOTIFY_CLUSTER.
!
! Flags 16-31 are the buffer status flags, when set,
! a buffer for the corresponding channel (1-16) is available.
! The application process mus clear the flag and set the corresponding
! flag in EF_NOTIFY_CLUSTER when it is finished with the buffer.

        Parameter EF_STATUS_CLSTR = 'LABIO_EF_STATUS'
!First event flag in Activity and Status
        Parameter EF_ACTIVITY_1 = 96
        Parameter EF_STATUS_1 = EF_ACTIVITY_1 + 16
!Offset to Activity and Status
        Parameter EF_ACTIVITY_OFF = 95
        Parameter EF_STATUS_OFF = EF_ACTIVITY_OFF + 16

! BIT array, BIT(I) = has bit I set ( I = 1 to 32 )

        Integer*4 BIT(32)
        Data BIT/ '1'X,'2'X,'4'X,'8'X,'10'X,'20'X,'40'X,'80'X,
        1         '100'X,'200'X,'400'X,'800'X,'1000'X,'2000'X,
        1         '4000'X,'8000'X,'10000'X,'20000'X,'40000'X,
        1         '80000'X,'100000'X,'200000'X,'400000'X,
        1         '800000'X,'1000000'X,'2000000'X,'4000000'X,
        1         '8000000'X,'10000000'X,'20000000'X,'40000000'X,
        1         '80000000'X/
!
![End of File]
```

```
!File: LABIOSEC.FOR
! Block Data Routine to place the LABIO_SECTION Common
! on a page boundary. This is necessary because we will
! remap it. We could have used a MACRO program to
! declare the PSECT LABIO_SECTION to be paged aligned,
! but the LINKer would then give us a warning message.

        Block Data LABIO_SECTION
        Common /LABIO_SECTION/ AD_BLOCK
        End
!
![End of File]
```

```
!FILE:CONNECT.COM
! This command file loads the connect-to-interrupt handler (CONINTERR) and
! then connects the KW11-K to to it.
!
$ R SYS$SYSTEM:SYSGEN
LOAD CONINTERR
CONNECT KWA0 /ADAPTER=3/CSR=%O770444/VEC=%O404/DRIVER=CONINTERR
$ Exit
```

```
!File:  LABIOSTRT.COM
!
!Starts up the LABIO SYSTEM
!Runs the data acquisition process and connect process
!as detached tasks. Then runs the status program.
!
!Make the logical name assignments
$Assign/Group LABIO.LOG LABIO_LOG         !Log file
$Assign/Group LABIO.DAT LABIO_SEC_FILE    !Global Section File
$Assign/Group KWA0: LABIO_AD              !Connect-to-Interupt device is KW-11
$Set Noon                                 !Don't abort if we can't run a program
$                                         !It is probably already running!
$!Run the data acquisition program
$
$Run/Uic=         'F$USER()'-             !Run as a deatched process
/Ast_Limit=       20-                     !We need a large AST quote
/Output =         LABIOACQ.DAT-           !SYS$OUTPUT
/Priority=        17-                     !High, Real-Time priority
/Process_name=    LABIO_DATA_ACQ-         !Name of Process
/Privileges=      SAME-                   !Same privileges
 LABIOACQ                                 !Image to run
$
$!Run the connect program
$
$ RUN/Uic=        'F$USER()'-             !Run as a detached process
/Output=          LABIOCON.DAT-           !SYS$OUTPUT
/Priority=        15-                     !Give it a high but not mighty priority
/Privilege=       SAME-
/Process_name=    LABIO_CONNECT-          !Name of the process
 LABIOCON
$
$!Run the status program
$Run LABIOSTAT
$Set On
```

```
!File:  LABIOCOMP.COM
! Command procedure to compile and assemble
! the modules of the LABIO system.

$ Fortran LABIOACQ,LABIOCON,LABIOSTAT,LABIOCOM,LABIOSEC
$ Macro/List LABIOCIN+Sys$Library:LIB.MLB/Library
$ Macro/List GBLSECUFO
```

```
$! Demo Programs
$ Fortran LABIOSAMP,LABIOPEAK,PEAK,TESTLABIO
```

```
!File:  LABIOLINK.COM
! Command procedure to LINK the LABIO system
$ Link/Map LABIOACQ,GBLSECUFO,LABIOCOM,LABIOCIN/Option
$ Link/Map LABIOCON,LABIO/Option
$ Link/Map LABIOSTAT,LABIO/Option
$! Demo Programs
$ Link/Map LABIOPEAK,PEAK,LABIO/Opt
$ Link/Map LABIOSAMP,LABIO/Opt
$ Link/Map TESTLABIO,LABIO/OPt
```

```
!File: LABIO.OPT
!Linker OPTION file for linking any process to be used with LABIO
LABIOCOM
Cluster = LABIO_CLUSTER,,,LABIOSEC
```

```
!File:  LABIOCIN.OPT
!Linker OPTION file for linking LABIO_DATA_ACQ
Cluster = Labio_cluster,,,Labiocin
```

## 7.2 AIRLINE RESERVATION SYSTEM

This example shows a series of programs to make and cancel airline reservations. This is not a "real-time" example in the same sense as the data acquisition and manipulation example in Section 7.1. However, the airline reservation system does show a shareable image data base, access to which is synchronized by the use of common event flags. It also shows the use of a shared memory common event flag cluster.

The following commands define the logical names and install the global section for the airline reservation system (FORTRAN program examples). The shared memory is named SHM.

```
$ COPY DATABASE.EXE SYS$SHARE:DATABASE.EXE  !PUT IT IN LIBRARY
$ DEFINE GBL$DATABASE SHM:DATABASE !LOGICAL NAME DEF. FOR SECTION
$ RUN SYS$SYSTEM:INSTALL
INSTALL> SYS$SHARE:DATABASE/OPEN/HEADER_RESIDENT/SHARED
INSTALL> [CTRL/Z]
$ DEFINE/SYSTEM CEF$CEFN1 SHM:CEFN1 !LOG. NAME DEF. FOR CLUSTER
$ RUN [desired program in the reservation system]
```

```
C        DATADESC.FOR
C
C               VMS AIRLINE RESEVATION SYSTEM
C
C       BEING A SIMPLE DEMONSTRATION OF THE USE OF A GLOBAL
C       DATABASE AS A SHAREABLE IMAGE UNDER VAX/VMS.
C
C
C       DISCLAIMER:     THIS SOFTWARE IS FOR DEMONSTRATION PURPOSES
C       -----------     ONLY. NO AIRLINE IS EXPECTED TO HONOUR THESE
C                       RESERVATIONS. FURTHER, IT IS INTENDED ONLY TO
C                       DEMONSTRATE SOME OF THE TECHNIQUES AVAILABLE
                        WITH VAX/VMS AND VAX-11 FORTRAN.
C
        PARAMETER NDESTS = 4
        PARAMETER NDAYS = 3
        PARAMETER NSEATS = 10
        PARAMETER ITOTSEATS = NDESTS*NDAYS*NSEATS*2
C
        CHARACTER DESTINS(NDESTS)*6,SEATS(NSEATS,2,NDESTS,NDAYS)*20
        CHARACTER DAYS(NDAYS)*3
C
        INTEGER HOWPAID(ITOTSEATS)
C
        COMMON /FLIGHTDATA/SEATS,DAYS,DESTINS
        COMMON /PAIDDATA/HOWPAID
```

```
        BLOCK DATA DATABASE
C
        INCLUDE 'DATADESC.FOR'
C
        DATA DESTINS/'BOSTON','SYDNEY','LONDON','MADRID'/
        DATA DAYS/'MON','TUE','WED'/
        DATA SEATS/ITOTSEATS*'                    '/
C
        END
```

```
        SUBROUTINE  LOCKFLIGHT(IDEST,IDAY)
C
        INCLUDE  'DATADESC.FOR'
C
        EXTERNAL SS$_WASSET
        INTEGER PREVSTATE,EVFLAG
        INTEGER SYS$SETEF,SYS$CLREF,SYS$ASCEFC
C
        EVFLAG = 63 + NDAYS*(IDEST - 1) + IDAY
        IF ( .NOT. SYS$ASCEFC(%VAL(EVFLAG),%DESCR('FLIGHTLOCKS'),,)) GO TO 900
 10     PREVSTATE = SYS$SETEF(%VAL(EVFLAG))
        IF (PREVSTATE .EQ. %LOC(SS$_WASSET)) THEN
                GO TO 10
        ELSE
                IF ( .NOT. PREVSTATE) GO TO 900
                RETURN
        END IF
        ENTRY UNLOCKFLIGHT
        IF (.NOT. SYS$CLREF(%VAL(EVFLAG))) GO TO 900
        RETURN
 900    TYPE 910
 910    FORMAT(' **** EVENT FLAG SERVICE FAILURE ****')
        END
```

# PROGRAM EXAMPLES

```
        PROGRAM DISPLAY
C
        INCLUDE 'DATADESC.FOR'
C
        CHARACTER DESTIN*6,DAY*3,HOMERASE*4,BLANKS*6,SMOKE*1
        CHARACTER TIMEDELAY*13,TOPOFSCREEN*6
C
        INTEGER SYS$BINTIM,SYS$SETIMR,SYS$WAITFR,SYS$CLREF,DELAY(2)
C
        BYTE CTLERASE(4),CTLTOS(4)
C
        EQUIVALENCE (HOMERASE,CTLERASE(1)),(TOPOFSCREEN,CTLTOS(1))
C
        DATA CTLERASE/'1B'X,'H','1B'X,'J'/,BLANKS/'      '/
        DATA TIMEDELAY/'0 00:00:10.00'/
        DATA CTLTOS/'1B'X,'Y','22'X,'20'X/
C
 1000   FORMAT(' Enter flight destination: ',$)
 1010   FORMAT(A)
 1020   FORMAT(' There are no flights to ',A)
 1030   FORMAT(' On what day? ',$)
 1040   FORMAT(' ',A,'DESTIN DAY SEAT PASSENGER NAME       CREDIT
     1 CARD NO. (0 IF CASH)',/,' ------ --- ---- --------------
     1          ---------------',/)
 1050   FORMAT('+',A,' ',A,'  ',A,I2,' ',A,I10,/)
 1060   FORMAT(' ',A)
C
 10     TYPE 1000
        ACCEPT 1010, DESTIN
        DO 20 IDEST = 1,NDESTS
        IF (DESTIN(1:2) .EQ. DESTINS(IDEST)(1:2)) GO TO 40
 20     CONTINUE
C
        TYPE 1020, DESTIN
        GO TO 10
C
 40     TYPE 1030
        ACCEPT 1010, DAY
        DO 60 IDAY = 1,NDAYS
        IF (DAY(1:2) .EQ. DAYS(IDAY)(1:2)) GO TO 80
 60     CONTINUE
        IF (DAY(1:3) .EQ. 'ALL') THEN
                IDAY = -1
                GO TO 80
        END IF
        GO TO 40
C
 80     CONTINUE
        IF (IDEST .EQ. -1) THEN
                JDEST = 1
                KDEST = NDESTS
        ELSE
                JDEST = IDEST
                KDEST = IDEST
        END IF
        IF (IDAY .EQ. -1) THEN
                JDAY = 1
                KDAY = NDAYS
        ELSE
                JDAY = IDAY
                KDAY = IDAY
        END IF
C
        TYPE 1040, HOMERASE
 90     LINES = 0
        DO 500 IDEST = JDEST,KDEST
        ILOOP = 0
C
                DO 400 IDAY = JDAY,KDAY
                JLOOP = 0
C
                        DO 300 ISEAT = 1,2*NSEATS
                        ILOOP = ILOOP + 1
                        JLOOP = JLOOP + 1
                        IF (ISEAT .LE. NSEATS) THEN
                                SMOKE = 'N'
                                ISMOKE = 1
                                JSEAT = ISEAT
                        ELSE
                                SMOKE = 'S'
                                ISMOKE = 2
                                JSEAT = ISEAT - NSEATS
                        END IF
                        LSEAT = ISEAT + (IDEST-1)*2*NSEATS + (IDAY-1)*NDESTS
```

```
C
                        IF (LINES) 100,100,99
 99                     IF (ILOOP - 2) 100,120,140
 100                    DESTIN = DESTINS(IDEST)
                        GO TO 140
 120                    DESTIN = BLANKS
 140                    CONTINUE
C
                        IF (LINES) 160,160,150
 150                    IF (JLOOP - 2) 160, 180, 200
 160                    DAY = DAYS(IDAY)
                        GO TO 200
 180                    DAY = BLANKS
 200                    CONTINUE
                        IF (SEATS(JSEAT,ISMOKE,IDEST,IDAY)(1:4) .EQ. '    ') THEN
                                IF (ISEAT .NE. 1) THEN
                                GO TO 300
                                END IF
                        END IF
                        TYPE 1050, DESTIN,DAY,SMOKE,ISEAT,SEATS(JSEAT,ISMOKE,IDEST,IDAY),
     1 HOWPAID(LSEAT)
                        LINES = LINES + 1
                        IF (LINES .GE. 19) THEN
                                TYPE 1060, TOPOFSCREEN
                                LINES = 0
                        END IF
 300                    CONTINUE
 400            CONTINUE
 500    CONTINUE
C
        END
```

```
        PROGRAM RESERVATION
C
        INCLUDE 'DATADESC.FOR'
C
        CHARACTER DESTIN*6,DAY*3,SMOKE*3,PAYMENT*4
C
 1000   FORMAT(' Enter destination: ',$)
 1010   FORMAT(A)
 1020   FORMAT(' There are no flights to ',A)
 1030   FORMAT(' On what day? ',$)
 1040   FORMAT(' Do you want a smoking area seat? ',$)
 1050   FORMAT(' The flight to ',A,' is full on ',A)
 1060   FORMAT(' No smoker seats left. Is non-smoking acceptible ?',$)
 1070   FORMAT(' Non-smoking is full. Is smoking area acceptible ?',$)
 1080   FORMAT(' Your seat is number ',I4,' on the ',A,' flight next ',A)
 1090   FORMAT(' Enter passenger name: ',$)
 1100   FORMAT(' Payment by cash or credit card? ',$)
 1110   FORMAT(' Enter credit card number: ',$)
 1120   FORMAT(I10)
 1130   FORMAT(' *** INVALID CREDIT CARD NUMBER ***')
C
 10     TYPE 1000
        ACCEPT 1010, DESTIN
        DO 20 IDEST = 1,NDESTS
        IF (DESTIN(1:2) .EQ. DESTINS(IDEST)(1:2)) THEN
                GO TO 40
        END IF
 20     CONTINUE
C
        TYPE 1020, DESTIN
        GO TO 10
```

```
C
 40     TYPE 1030
        ACCEPT 1010, DAY
        DO 60 IDAY = 1,NDAYS
        IF (DAY(1:2) .EQ. DAYS(IDAY)(1:2)) THEN
                GO TO 80
        END IF
 60     CONTINUE
        GO TO 40
C
 80     TYPE 1040
        ACCEPT 1010, SMOKE
        IF (SMOKE(1:1) .EQ. 'Y') THEN
                ISMOKE = 1
        ELSE IF (SMOKE(1:1) .EQ. 'N') THEN
                        ISMOKE = 0
                ELSE
                        GO TO 80
        END IF
C
        CALL LOCKFLIGHT(IDEST,IDAY)
C
        DO 100 ISEAT = 1,NSEATS
        IF (SEATS(ISEAT,ISMOKE+1,IDEST,IDAY)(1:4) .EQ. '    ') THEN
                GO TO 200
        END IF
 100    CONTINUE
        JSMOKE = ISMOKE .XOR. 1
        DO 110 ISEAT = 1,NSEATS
        IF (SEATS(ISEAT,JSMOKE+1,IDEST,IDAY)(1:4) .EQ. '    ') THEN
                GO TO 150
        END IF
 110    CONTINUE
        TYPE 1050, DESTINS(IDEST),DAYS(IDAY)
 120    CALL UNLOCKFLIGHT
        GO TO 900
C
 150    IF (ISMOKE .EQ. 1) THEN
                TYPE 1060
                GO TO 170
        ELSE
                TYPE 1070
        END IF
 170    ACCEPT 1010, SMOKE
        IF (SMOKE(1:1) .EQ. 'N') THEN
                GO TO 120
        END IF
        ISMOKE = JSMOKE
C
 200    JSEAT = ISEAT + (NSEATS*ISMOKE)
        KSEAT = JSEAT + (IDEST-1)*2*NSEATS + (IDAY-1)*NDESTS
        TYPE 1080, JSEAT,DESTINS(IDEST),DAYS(IDAY)
        TYPE 1090
        ACCEPT 1010, SEATS(ISEAT,ISMOKE+1,IDEST,IDAY)
 220    TYPE 1100
        ACCEPT 1010, PAYMENT
        IF ( PAYMENT(1:2) .EQ. 'CA') THEN
                HOWPAID(KSEAT) = 0
        ELSE IF (PAYMENT(1:2) .NE. 'CR') THEN
                GO TO 220
        ELSE
 240            TYPE 1110
                ACCEPT 1120, HOWPAID(KSEAT)
                IF (HOWPAID(KSEAT) .NE. 0) GO TO 260
                TYPE 1130
                GO TO 240
        END IF
 260    CONTINUE
        GO TO 120
 900    CONTINUE
        END
        PROGRAM CANCEL
C
        INCLUDE 'DATADESC.FOR'
C
        CHARACTER DESTIN*6,DAY*3,NAME*20,BLANKS*20
C
        DATA BLANKS/'                    '/
C
 1000   FORMAT(' Enter destination: ',$)
 1010   FORMAT(A)
 1020   FORMAT(' There are no flights to ',A)
 1030   FORMAT(' On what day? ',$)
 1040   FORMAT(' ',A' does not hold a seat to ',A,' on ',A,' flight')
 1050   FORMAT(' Seat number ',I4,' cancelled on the ',A,
     1' flight next ',A)
 1090   FORMAT(' Enter passenger name: ',$)
C
        TYPE 1090
        ACCEPT 1010, NAME
 10     TYPE 1000
        ACCEPT 1010, DESTIN
        DO 20 IDEST = 1,NDESTS
        IF (DESTIN(1:2) .EQ. DESTINS(IDEST)(1:2)) THEN
                GO TO 40
        END IF
 20     CONTINUE
C
        TYPE 1020, DESTIN
        GO TO 10
C
 40     TYPE 1030
        ACCEPT 1010, DAY
        DO 60 IDAY = 1,NDAYS
        IF (DAY(1:2) .EQ. DAYS(IDAY)(1:2)) THEN
                GO TO 80
        END IF
 60     CONTINUE
        GO TO 40
C
C
 80     ISMOKE = 0
 90     DO 100 ISEAT = 1,NSEATS
        IF (SEATS(ISEAT,ISMOKE+1,IDEST,IDAY)(1:10) .EQ. NAME(1:10)) THEN
                CALL LOCKFLIGHT(IDEST,IDAY)
                GO TO 200
        END IF
 100    CONTINUE
        IF (ISMOKE .EQ. 0) THEN
                ISMOKE = 1
                GO TO 90
        ELSE
                TYPE 1040, NAME, DESTIN, DAY
                GO TO 900
        END IF
C
 200    JSEAT = ISEAT + (NSEATS*ISMOKE)
        KSEAT = JSEAT + (IDEST-1)*2*NSEATS + (IDAY-1)*NDESTS
        TYPE 1050, JSEAT,DESTINS(IDEST),DAYS(IDAY)
        SEATS(ISEAT,ISMOKE+1,IDEST,IDAY)(1:20) = BLANKS(1:20)
        HOWPAID(KSEAT) = 0
        CALL UNLOCKFLIGHT
 900    CONTINUE
        END
        PROGRAM MONITOR
C
        INCLUDE 'DATADESC.FOR'
C
        CHARACTER DESTIN*6,DAY*3,HOMERASE*4,BLANKS*6,SMOKE*1
        CHARACTER TIMEDELAY*13,TOPOFSCREEN*6
C
        INTEGER SYS$BINTIM,SYS$SETIMR,SYS$WAITFR,SYS$CLREF,DELAY(2)
C
        BYTE CTLERASE(4),CTLTOS(6)
C
        EQUIVALENCE (HOMERASE,CTLERASE(1)),(TOPOFSCREEN,CTLTOS(1))
C
        DATA CTLERASE/'1B'X,'H','1B'X,'J'/,BLANKS/'      '/
        DATA TIMEDELAY/'0 00:00:10.00'/
        DATA CTLTOS/'1B'X,'Y','22'X,'20'X,'1B'X,'J'/
C
 1000   FORMAT(' Enter flight destination: ',$)
 1010   FORMAT(A)
 1020   FORMAT(' There are no flights to ',A)
 1030   FORMAT(' On what day? ',$)
 1040   FORMAT(' ',A,'DESTIN DAY SEAT PASSENGER NAME       CREDIT
     1 CARD NO. (0 IF CASH)',/,' ------ --- ---- --------------
     1         ---------------',/)
 1050   FORMAT('+',A,' ',A,'  ',A,I2,' ',A,I10,/)
 1060   FORMAT(' ',A)
C
 10     TYPE 1000
        ACCEPT 1010, DESTIN
        DO 20 IDEST = 1,NDESTS
        IF (DESTIN(1:2) .EQ. DESTINS(IDEST)(1:2)) THEN
                GO TO 40
        END IF
 20     CONTINUE
C
        IF (DESTIN(1:3) .EQ. 'ALL') THEN
                IDEST = -1
                GO TO 40
        END IF
        TYPE 1020, DESTIN
        GO TO 10
C
 40     TYPE 1030
        ACCEPT 1010, DAY
        DO 60 IDAY = 1,NDAYS
        IF (DAY(1:2) .EQ. DAYS(IDAY)(1:2)) THEN
                GO TO 80
        END IF
 60     CONTINUE
        IF (DAY(1:3) .EQ. 'ALL') THEN
                IDAY = -1
                GO TO 80
        END IF
        GO TO 40
C
 80     CONTINUE
        IF (IDEST .EQ. -1) THEN
                JDEST = 1
                KDEST = NDESTS
        ELSE
                JDEST = IDEST
                KDEST = IDEST
        END IF
        IF (IDAY .EQ. -1) THEN
                JDAY = 1
                KDAY = NDAYS
        ELSE
                JDAY = IDAY
                KDAY = IDAY
        END IF
C
        TYPE 1040, HOMERASE
 90     LINES = 0
        DO 500 IDEST = JDEST,KDEST
        ILOOP = 0
```

```
C
                DO 400 IDAY = JDAY,KDAY
                JLOOP = 0
C
                        DO 300 ISEAT = 1,2*NSEATS
                        ILOOP = ILOOP + 1
                        JLOOP = JLOOP + 1
                        IF (ISEAT .LE. NSEATS) THEN
                                SMOKE = 'N'
                                ISMOKE = 1
                                JSEAT = ISEAT
                        ELSE
                                SMOKE = 'S'
                                ISMOKE = 2
                                JSEAT = ISEAT - NSEATS
                        END IF
                        LSEAT = ISEAT + (IDEST-1)*2*NSEATS + (IDAY-1)*NDESTS
C
                        IF (LINES) 100,100,99
 99                     IF (ILOOP - 2) 100,120,140
 100                    DESTIN = DESTINS(IDEST)
                        GO TO 140
 120                    DESTIN = BLANKS
 140                    CONTINUE
C
                        IF (LINES) 160,160,150
 150                    IF (JLOOP - 2) 160, 180, 200
 160                    DAY = DAYS(IDAY)
                        GO TO 200
 180                    DAY = BLANKS
 200                    CONTINUE
                        IF (SEATS(JSEAT,ISMOKE,IDEST,IDAY)(1:4) .EQ. '    ') THEN
                                IF (ISEAT .NE. 1) THEN
                                GO TO 300
                                END IF
                        END IF
                        TYPE 1050, DESTIN,DAY,SMOKE,ISEAT,SEATS(JSEAT,ISMOKE,IDEST,IDAY),
     1 HOWPAID(LSEAT)
                        LINES = LINES + 1
                        IF (LINES .GE. 19) THEN
                                IX = SYS$BINTIM(%DESCR(TIMEDELAY),DELAY)
                                IF (.NOT. IX) GO TO 900
                                IX = SYS$CLREF(%VAL(1))
                                IF (.NOT. IX) GO TO 900
                                IX = SYS$SETIMR(%VAL(1),DELAY,,)
                                IF (.NOT. IX) GO TO 900
                                IX = SYS$WAITFR(%VAL(1))
                                IF (.NOT. IX) GO TO 900
                                TYPE 1060, TOPOFSCREEN
                                LINES = 0
                        END IF
 300                    CONTINUE
 400            CONTINUE
 500    CONTINUE
C
        IX = SYS$BINTIM(%DESCR(TIMEDELAY),DELAY)
        IF (.NOT. IX) GO TO 900
        IX = SYS$CLREF(%VAL(1))
        IF (.NOT. IX) GO TO 900
        IX = SYS$SETIMR(%VAL(1),DELAY,,)
        IF (.NOT. IX) GO TO 900
        IX = SYS$WAITFR(%VAL(1))
        IF (.NOT. IX) GO TO 900
        TYPE 1060, TOPOFSCREEN
        GO TO 90
C
 900    CALL LIB$SIGNAL(%VAL(IX))
        END
```

```
$!                      BLDVMSAIR.COM
$!
$!      COMMAND FILE TO REBUILD FROM SOURCE
$!      THE AIRLINE RESERVATION SYSTEM WHICH IS
$!      A DEMO OF SHAREABLE IMAGES
$!
$ FORTRAN/LIST/MACHINE_CODE DATABASE
$ FORTRAN/LIST/MACHINE_CODE INTERLOCK
$ FORTRAN/LIST/MACHINE_CODE RESERVE
$ FORTRAN/LIST/MACHINE_CODE DISPLAY
$ FORTRAN/LIST/MACHINE_CODE CANCEL
$ FORTRAN/LIST/MACHINE_CODE MONITOR
$ LINK/SHAREABLE/MAP/FULL/CROSS DATABASE,INTERLOCK,DATABASE/OPTIONS
$ LINK/MAP/FULL/CROSS RESERVE,GETSHRIMG/OPTIONS
$ LINK/MAP/FULL/CROSS DISPLAY,GETSHRIMG/OPTIONS
$ LINK/MAP/FULL/CROSS MONITOR,GETSHRIMG/OPTIONS
$ LINK/MAP/FULL/CROSS CANCEL,GETSHRIMG/OPTIONS
$ PURGE *.*
```

```
!                       DATABASE.OPT
!
!       LINK TIME OPTIONS DESCRIPTION FILE TO BUILD
!       THE SHARABLE IMAGE CONTAINING THE DATA BASE AND
!       THE INTERLOCK ROUTINE
!
UNIVERSAL=LOCKFLIGHT,UNLOCKFLIGHT       ! MAKE ROUTINE ENTRY POINTS
                                        ! ACCESSIBLE TO USER PROGRAMS
GSMATCH=LEQUAL,0,0000                   ! SET GLOBAL SECTION MATCH CONTROL
```

```
!                       GETSHRIMG.OPT
!
!       LINK TIME OPTIONS FILE TO ACQUIRE THE SHARED
!       DATABASE AND INTERLOCKING ROUTINE.
!
DATABASE/SHARE=NOCOPY           ! MAPPED INTO ADDRESS SPACE
```

# APPENDIX A

# LOCKING A RESOURCE

A semaphore is a metering device that provides the capability of controlling access to a set of resources. A semaphore that controls access to a single resource is called a mutex (mutual exclusion) or, more commonly, a lock.

You can perform two operations on a mutual exclusion semaphore (lock):

- Lock - Test to see if the resource is free. If it is, then take (use) it and proceed with execution. If the resource is not free, execution is stalled until the resource becomes available.

- Unlock - Give the resource back (make it available to others) when it is no longer needed. If any other processes are stalled waiting for the resource, they are awakened.

Locking and unlocking must be interlocked operations, so that no race conditions result. An example of a race condition is as follows: in the middle of the first process's test for a resource's availability, the resource is returned by another process, but the return goes unnoticed by the first process.

Two methods of creating a lock are (1) using a common event flag or (2) using a queue. In selecting either method, you must consider how you want to service requests for the resource, how important is ease of use, and how quickly the method must execute. Table A-1 contrasts the two methods.

Table A-1
Two Methods of Creating a Lock

| Event Flag | Queue |
|---|---|
| 1. Requests serviced according to process priority | 1. Requests serviced on a first-in first-out (FIFO) or a last-in first-out (LIFO) basis |
| 2. Easy to use | 2. More complicated to use (requires a global section and special data structures) |
| 3. Uses time manipulating the event flag | 3. Executes at hardware instruction speed when no conflict occurs |

# LOCKING A RESOURCE

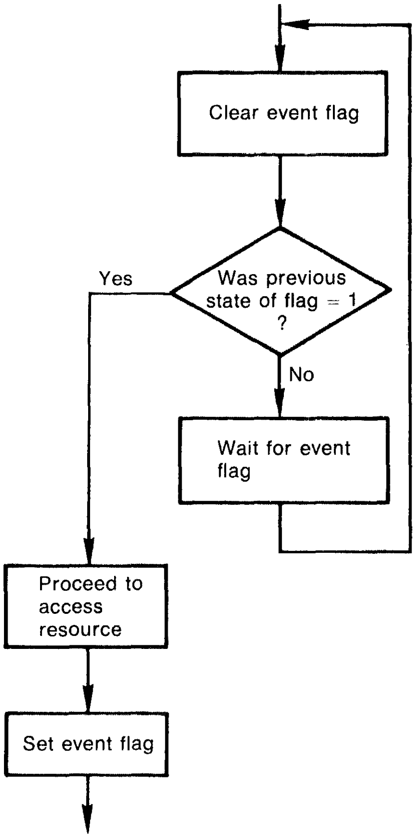
## A.1 USING AN EVENT FLAG

Cooperating processes can control access to a resource by using a common event flag as a lock. The procedure is as follows:

1. An initialization process is run to create a permanent common event flag cluster and to set the initial state of all 32 flags to 1. This provides 32 individual locks.

2. Each cooperating process must associate with the common event flag cluster.

3. Before any process uses the resource represented by a particular event flag, it must execute the logic shown in Figure A-1.



```
5$:     $CLREF_S   EFN=#65


        CMPL  R0,#SS$_WASSET
        BEQL  10$


        $WAITFR_S   EFN=#65
        BRW       5$

10$:      .       ;Proceed
          .
          .


        $SETEF_S   EFN=#65
```

Figure A-1 Event Flag Lock Logic

Because the initial state of the event flag is 1, only one process at a time will be able to clear the event flag and find its previous state to be a 1. All subsequent processes will find the previous state to be 0, and thus will wait until the owner process sets the flag. (This occurs when the owner process is finished with the resource and returns it.)

Setting the event flag causes all the waiting processes to awaken and compete for CPU time according to their process priority (unless an outstanding I/O request or some other factor prevents a higher-priority process from becoming computable). However, only one waiting process will be able to clear the event flag and find its previous state to be a 1. (Note: Clearing an event flag is an interlocked operation implemented by VAX/VMS.)

Figure A-2 is a VAX-11 FORTRAN example using a common event flag as a lock. Note that in Figure A-2 it is not necessary to run an initialization process (step 1 at the beginning of this section), because the program logic prevents a race condition from occurring during lock initialization.

```
  INTEGER*4 SYS$ASCEFC,SYS$SETEF,SYS$CLREF,SYS$WAITFR,STATUS
  EXTERNAL SS$_WASSET,SS$_WASCLR

C-- Associate with a common event flag cluster to be used as a mutual exclusion
C-- semaphore.  If the cluster does not exist, it is created.  The first two
C-- flags are used to avoid any race conditions during initialization.

  STATUS = SYS$ASCEFC(%VAL(64),'MUTEX',,)
  IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
  IF (SYS$SETEF(%VAL(64)) .EQ. %LOC(SS$_WASCLR)) THEN   ! If creator
          CALL SYS$SETEF(%VAL(66))                      ! Init mutex
          CALL SYS$SETEF(%VAL(65))                      ! Set initialized
  ELSE
          CALL SYS$WAITFR(%VAL(65))                     ! Initialized wait
  END IF

C-- Perform any other program initialization

  CONTINUE

C-- Obtain exclusive access to the mutex to make sure no other process
C-- will execute its critical section while we do.  If the mutex cannot be
C-- obtained, wait for it to be released.

50        STATUS = SYS$CLREF(%VAL(66))
  IF (STATUS .EQ. %LOC(SS$_WASSET)) GOTO 100
  STATUS = SYS$WAITFR(%VAL(66))
  GOTO 50

C-- Execute the critical section of the program

100       CONTINUE

C-- Release the mutex and unblock any other processes that might have
C-- been waiting.  If more than one is waiting, the first one to obtain the
C-- the mutex will get it, and the others will fail and wait again.

  CALL SYS$SETEF(%VAL(66))

  GOTO 50

  END
```

Figure A-2 Event Flag Lock Example

### A.1.1 Shared Memory Considerations

You can use an event flag in a shared memory common event flag cluster to guarantee that only one process uses a resource at a time. However, because of potential differences in the speeds and workloads of the processors connected to the shared memory, there is no assurance that the highest-priority waiting process will get the resource each time it becomes available.

## A.2 USING A QUEUE

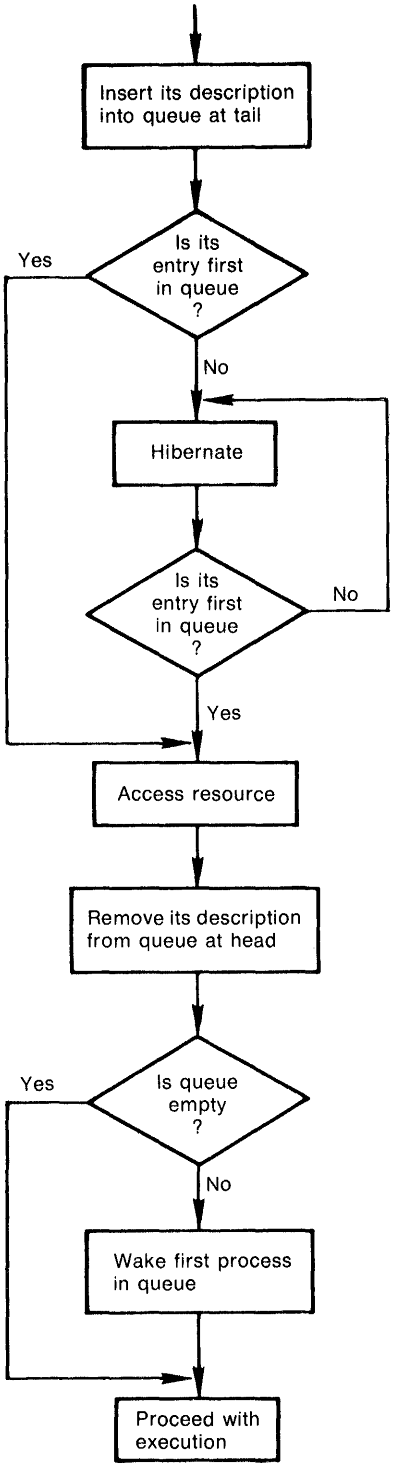
Cooperating processes can use a queue to lock a resource and, after unlocking, make the resource available on either a first-in first-out (FIFO) or last-in first-out (LIFO) basis. (Queues and the queue instructions are explained in the VAX-11 Architecture Handbook.) The procedure is as follows.

1. An initialization process must be run to create a permanent global section and initialize a queue header.

2. To use the resource represented by the queue header, each process must map the global section. Each process must also create a 3-longword description with the following format in the global section:

| |
|---|
| Forward link |
| Backward link |
| Process ID |

3. Before any process uses the resource represented by the queue header, it must execute the logic shown in Figure A-3. (Figure A-3 shows a FIFO queuing policy. Figure A-4 later in this appendix shows a LIFO policy.)


Insert its description into queue at tail
Is its entry first in queue ?
Yes
No
Hibernate
Is its entry first in queue ?
No
Yes
Access resource
Remove its description from queue at head
Is queue empty ?
Yes
No
Wake first process in queue
Proceed with execution

```
        INSQUE   DESC,@HEADER+4

        BEQL     10$

5$:     $HIBER_S

        CMPL     DESC,HEADER
        BNEQ     5$

10$:       .
           .
           .

        REMQUE   @HEADER,R0

        BEQL     20$

        MOVL     HEADER,R1
        $WAKE_S  PIDADR=8(R1)

20$:
```

Figure A-3  FIFO Queuing Policy

Because the initial state of the queue is empty, only one process will be able to insert its entry in the queue and find it to be the first entry. Each subsequent process will find more than one entry after inserting itself, and thus will hibernate.

When the owner process is finished using the resource, it simply removes its description from the head of the queue. If the queue is then empty, no process is waiting. If the queue is not empty, the process whose ID is first in the queue is awakened, and that process can now use the resource. (Note: The queue instructions are interlocked operations implemented by the VAX-11 processor.)

Figure A-3 and its explanation described a FIFO queuing policy. Figure A-4 shows the logic for a LIFO queuing policy.

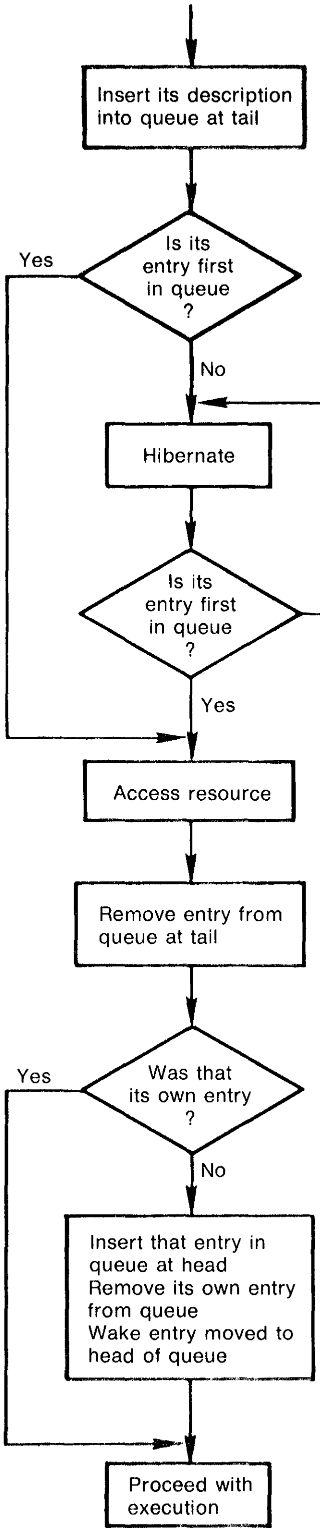
### A.2.1 Shared Memory Considerations

The logic and coding in Section A.2 cannot be used with a queue in shared memory for the following reasons:

- The Wake ($WAKE) system service cannot be used to wake a process running on another processor.
- The interlocked queue instructions must be used (INSQHI, INSQTI, REMQHI, REMQTI).

To use a queue in shared memory, you must devise a more complicated mechanism. (Such a mechanism is beyond the scope of this manual.)

## LOCKING A RESOURCE


Insert its description into queue at tail
Is its entry first in queue ?
Yes
No
Hibernate
Is its entry first in queue ?
No
Yes
Access resource
Remove entry from queue at tail
Was that its own entry ?
Yes
No
Insert that entry in queue at head Remove its own entry from queue Wake entry moved to head of queue
Proceed with execution

```
          INSQUE   DESC,@HEADER+4

          BEQL     10$

5$:       $HIBER_S

          CMPL     DESC,HEADER
          BNEQ     5$

10$:        .
            .
            .

          REMQUE   @HEADER+4,R1

          BEQL     20$

          REMQUE   @HEADER,R0
          INSQUE   (R1),@HEADER
          $WAKE_S  PIDADR=8(R1)

20$:
```

A-4 LIFO Queuing Policy

# APPENDIX B

# LPA11-K CONSIDERATIONS

Users should consider three factors in selecting and using the Laboratory Peripheral Accelerator (LPA11-K) for a real-time application:

- The effect on performance of resource availability and hardware configuration

- Throughout and response-time requirements of the application

- The LPA11-K driver's use of parameters in data acquisition calls

The remainder of this appendix discusses each of these considerations.

## B.1 RESOURCES, CONFIGURATION, AND PERFORMANCE

One factor that determines the performance of the LPA11-K is its interaction with other devices and applications in the system. The LPA11-K is designed as a real-time device. Its function is to sample data synchronously with a real-time clock. However, if for any reason the LPA11-K cannot maintain this synchronous transfer of data, a nonretriable error is generated. This method of operation contrasts with that of a disk, which can perform a retry because the original data is still available (in memory for a write or on disk for a read). In a real-time application, however, after the event of interest has passed it may no longer be of interest.

Therefore, the resources needed to carry out an application in real time must be guaranteed to be available. Some of the resources that must be available to use the LPA11-K in real time are UNIBUS adaptor map registers to map the buffers, UNIBUS adaptor data path, UNIBUS direct memory access (DMA) transfer bandwidth, processor interrupt response time, memory in the working set for data buffers, and CPU execution time for the application program. If the application buffers the data for storage on disk, the following resources must also be available: the disk controller and drive, and sufficient bandwidth and adaptors for the MASSBUS or UNIBUS (depending on where the disk is interfaced).

The VAX/VMS system gives the application program control over many system resources, to guarantee their availability when these resources are needed. Processes can lock critical pages, thus ensuring the availability of that memory. Processes can adjust their priority to guarantee access to CPU execution time and to mass storage controllers.

In other areas, however, control over resources is difficult, often because the resources are being used concurrently and involve interrupt handling and contention for bandwidth on I/O buses. In fact, several studies have concluded that the major impact on LPA11-K performance is UNIBUS I/O bandwidth contention.

The LPA11-K detects two classes of errors associated with real-time performance:

- Buffer overrun/underrun -- deals with the ability of the application program to supply new memory buffers fast enough (for example, to process data at least as fast as it is coming in)

- Data overrun/underrun -- deals with the ability of the device to arbitrate for UNIBUS cycles and to transfer data to and from main memory

Buffer overrun/underrun errors often reflect inadequate application control over resources; data overrun/underrun errors are usually caused by I/O contention.

The first class of errors, buffer overrun/underrun, is a function of the application. The application must run at a priority high enough to guarantee it sufficient CPU time. It must also have a working set large enough to hold in physical memory the data buffers and the code that performs computation on the data, to prevent excessive paging (or perhaps to prevent any paging at all). However, if these control measures have been taken and the buffers are large enough, and if buffer overrun/underrun errors still occur repeatedly, then the data rate is too fast for the work that needs to be done. In this case, the solution might be to buffer the data to intermediate mass storage for future processing.

The second class of errors, data overrun/underrun, is a function of UNIBUS and memory I/O contention. As other DMA devices on the UNIBUS become concurrently active, the effective throughput rate of the LPA11-K can be significantly reduced. If LPA11-K throughput falls below the application's requirements, an additional UNIBUS adaptor may be needed.

## B.2 THROUGHPUT AND RESPONSE-TIME REQUIREMENTS

The LPA11-K and its support under VAX/VMS are tailored primarily for throughput-intensive applications. This device can recognize a simple event, such as a single digital signal, and start data acquisition when the event occurs. However, if the application must respond quickly to events represented by the contents of the data being acquired, the LPA11-K might not be suitable for two reasons:

- The LPA11-K samples analog or digital data and stores it in large data buffers in main memory, generating an interrupt only when a buffer is full. Thus, if the application must detect a particular data value and respond quickly, it might have to wait for an entire buffer to be filled before it could start searching for the value.

- VAX/VMS is designed to manage LPA11-K data buffers transparently for application programs. This buffer management involves some software overhead. Thus, if data buffers were made very small (the smallest being one data point per buffer) in an effort to access data points sooner, the software overhead would grow considerably.

## B.3 PARAMETERS FOR DATA ACQUISITION CALLS

The LPA11-K uses parameters in some data acquisition procedures. For example, assume that an application must acquire a stream of analog data from several points at a specific rate per point, store the digitized data in memory, and stop when enough data has been taken.

To accomplish these goals, you must specify the following parameters: analog-to-digital conversion, the analog data channels to sample, the sample rate, the place in memory to store the data, and the amount of data to be taken. At the start of each data acquisition session, the application provides these values as parameters to the LPA11-K driver.

Data acquisition calls using parameters have the advantages of isolating the application from the actual hardware and simplifying the programming: the application programmer does not need to write interrupt service routines in assembly language or microcode. However, this approach might not be adequate for certain complicated applications requiring a sophisticated sampling algorithm or complex interactions between multiple data acquisition streams. If the application requires capabilities not provided by the LPA11-K parameters, other devices should be investigated.

# APPENDIX C

## VAX-11 BLISS-32 PROGRAM EXAMPLE

This appendix shows a VAX-11 BLISS-32 program using the connect-to-interrupt capability. The functions performed by the program are described in the "Abstract" near the beginning of the listing and in comments throughout the program. This program is only a simple illustration of connecting to an interrupt vector and does not reflect a typical real-time situation (for example, the line printer is not a real-time device).

```
MODULE lpmultast(%TITLE'line printer driver' MAIN=lp_main, IDENT='X02')=
!
!                       COPYRIGHT (c) 1980 BY
!               DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
! ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
! INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
! COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
! OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
! TRANSFERRED.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
! AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
! CORPORATION.
!
! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
!
!++
!
! FACILITY:
!
!       A sample program that illustrates the use of the connect to
!       interrupt facility.
!
! ABSTRACT:
!
!       This program assigns a channel to a line printer device, and
!       then connects to the device via the connect to interrupt
!       facility. The program then requests the name of a file from
!       the user, and outputs that file on the line printer.
!
!--
%SBTTL  'External and local symbol definitions'
BEGIN

LIBRARY 'SYS$LIBRARY:LIB';              ! Get important definitions

PSECT
```

```
    !+
    ! Define some PSECTs which we will need to refer to later
    !-
    OWN=        sharedata(ALIGN(9),WRITE),
    OWN=        data;

LINKAGE
    intrupt=    JSB(REGISTER=2, REGISTER=4, REGISTER=5):
                         NOPRESERVE(0,1,2,3,4) NOTUSED(6,7,8,9,10,11),
    cancel=     JSB(REGISTER=2, REGISTER=3, REGISTER=4, REGISTER=5):
                        NOTUSED(6,7,8,9,10,11);


FORWARD ROUTINE
    lp_interrupt:       intrupt PSECT(sharedata),        ! Interrupt server
    lp_cancel:  NOVALUE cancel PSECT(sharedata),         ! Cancel I/O
    lp_main,
    lp_isr_ast,
    lp_iodone_ast;


!
! Static Definitions
!

LITERAL
    true =      1,
    false =     0,

    io_page_count       = 1,            ! Pages needed in UNIBUS I/O space

    io_space_base = %x'20100000',       ! Physical address of UBA 0 space
                                        !  for VAX-11/780. Other processors
                                        !  need different magic number...

    unibus_lp_addr= %O'777514',         ! 18-bit addr of LP11 CSR

    !
    ! Calculate the page-frame number to map to get the physical address
    ! that the unibus is mapped on.
    !
    io_page_pfn = (io_space_base + unibus_lp_addr)/512,

    lp_csr_offset = %O'514',            ! Offset to printer CSRs.

    filename_length = 100,

    record_bufsiz =     256,
    prompt_length =     28;


OWN
    lpchan:     WORD,                           ! Line printer channel number.

    filename_buffer:    VECTOR[filename_length,BYTE],
    file_descr:         VECTOR[2] INITIAL( filename_length, filename_buffer),
    fdlen:              WORD,

    record_buffer:      VECTOR[record_bufsiz,BYTE],

    file_fab:   $fab(                           ! Input file fab
! Functional description:
!
!       This routine services an interrupt from the line printer
!       device. If the interrupt was expected, the routine disables
!       output interrupts. The disable is an optimization to prevent one
!       interrupt per character. With output interrupts disabled, the
!       line printer buffers characters until the device needs to output
!       the characters. Then the main program enables output interrupts
!       only for the period of time necessary for the device to empty
!       the buffer.
!
!       Then the interrupt service routine loads a success status into
!       R0 and returns.
!
!       If the interrupt was not expected, the routine just loads
!       an error status into R0 to prevent delivery of an AST to the
!       owning process and returns.
!
! Inputs:
!
!       R2      - address of a counted argument list
!       R4      - address of the IDB
!       R5      - address of the UCB
!
!       The counted argument list is as follows:
!
!                0(R2)  - count of arguments (4)
!                4(R2)  - the system-mapped address of the user buffer
!                8(R2)  - the system-mapped address of the device's CSR
!               12(R2)  - the IDB address
!               16(R2)  - the UCB address
!
! Outputs:
!       The routine must preserve all registers except R0-R4.
!--
    BEGIN
    MAP
        arglist:        REF VECTOR[,LONG],
        ucb:            REF BLOCK[,BYTE],
        idb:            REF BLOCK[,BYTE];
    BIND
        bufadr = arglist[1]:    REF BLOCK FIELD(buf);   ! System adr of buffer

    BUILTIN
        TESTBITCC;

    IF TESTBITCC( bufadr[buf$l_flags] )
    THEN
        RETURN 0;                       ! No interrupt expected, no AST wanted

    (.idb[idb$l_csr])<0,16> = 0;        ! Disable the output interrupt

    ss$_normal
    END;
%SBTTL  'LP_CANCEL, Cancel I/O on Line Printer'

ROUTINE lp_cancel( chan_idx, irp, pcb, ucb ): NOVALUE cancel PSECT(sharedata)=
!++
! Functional description:
!
!       This routine disables output interrupts from the line printer.
```

```
                fac=get,
                fna=filename_buffer,
                org=seq,
                rfm=var,
                dnm='TEST.LIS'),

    file_rab:   $rab(
                fab=file_fab,
                rac=seq,
                ubf=record_buffer,
                usz=record_bufsiz),

    io_page_limits:     VECTOR[2]       ! Addresses of process-mapped
                        INITIAL(200,    ! UNIBUS I/O page. 200 tells $CRMPSC
                                200);   ! to map pages in P0 space




BIND
    onesecond_delta=                    ! Delta time format for one
        UPLIT(-10*1000*1000,-1);        ! second.

!+
! Define offsets into the buffer that will be shared by the user
! process and the process routines that execute in kernel mode.
!-

FIELD
   buf=
        SET
        buf$l_flags=    [0,0,32,0],     ! Flags longword.
            buf$v_int=  [0,0,1,0],      ! Interrupt expected

        buf$w_charcount=[4,0,16,0],     ! Number of chars in buffer
        buf$l_startdata=[8,0,32,0]      ! Start of data in buffer.
        TES,

    lp=
        SET
        lp_csr=         [0,0,16,1],     ! Offset to line printer CSR
        lp_dbr=         [2,0,8,0]       ! Offset to line printer data
        TES;
%SBTTL  'Double Mapped Page Buffers'

OWN
    output_buffer:      BLOCK[512,BYTE] FIELD(buf) PSECT(sharedata);

PSECT
    OWN=        sharedata,
    PLIT=       sharedata;

!
! The routines to be executed in kernel mode must follow directly
! after this allocation of bytes to hold output data.
!
%SBTTL  'LP_INTERRUPT, Interrupt service routine'

ROUTINE lp_interrupt( arglist, idb, ucb ): intrupt PSECT(sharedata)=
!++
!
! Inputs:
!
!       R2      - negated value of the channel index number
!       R3      - address of the current IRP (I/O request packet)
!       R4      - address of the PCB (process control block) for the
!                 process canceling I/O
!       R5      - address of the UCB (unit control block)
!
! Outputs:
!
!       none
!
!--
    BEGIN
    MAP
        irp:    REF BLOCK[,BYTE],
        pcb:    REF BLOCK[,BYTE],
        ucb:    REF BLOCK[,BYTE];

    BIND
        crb=    .ucb[ucb$l_crb]:        BLOCK[,BYTE];

    LOCAL
        csr:    REF BLOCK[,BYTE] FIELD(lp);     ! UNIBUS addr.


    csr = ..(crb[crb$l_intd] + BLOCK[0, vec$l_idb;0,BYTE]);     ! Addr of CSR

    csr[lp_csr] = 0                             ! Disable output interrupts.
    END;
%SBTTL  'LP_MAIN, the main routine'

ROUTINE lp_main: PSECT($CODE$)=
!++
! LP_MAIN, the routine that controls the others
!
! Functional description:
!
!       1. Assign a channel to the line printer.
!       2. Map the process to the I/O page.
!       3. Issue a connect to interrupt QIO to get the line printer.
!       4. Prompt the user for a file name.
!       5. Open and connect to the file.
!       6. Write the contents of the file to the line printer.
!
! Inputs:
!
!       none
!
! Outputs:
!
!       R0      - status code
!                       SS$_NORMAL      - success
!                       RMS_code        - error in opening or reading
!                                               the file
!                       SS$_DEVOFFLINE  - error is writing to printer
!
!--
    BEGIN
    PSECT
```

```
        OWN=   $OWN$;
    OWN
        buffer_desc: VECTOR[2] INITIAL(         ! Descriptor of buffer shared
                            512+512,            ! by process and kernel mode
                            output_buffer),     ! process routines.

        entry_list: VECTOR[4] INITIAL(          ! List of offsets to kernel
                    0,                          ! mode routines: init device;
                    0,                          !       start device;
                    lp_interrupt-output_buffer, !       interrupt servicing;
                    lp_cancel-output_buffer);   !       cancel I/O.
    LOCAL
        csr:    REF BLOCK[,BYTE] FIELD(lp) VOLATILE,
        status;

    EXTERNAL ROUTINE
        lib$get_input;

!
! Assign a channel to the line printer.
!

    status = $assign(                           ! Assign channel to line
                devnam=$DESCRIPTOR('LPA0'),     !  printer
                chan=lpchan);

    IF NOT .status THEN RETURN .status;
!
! Map the UNIBUS I/O page to the process so that the line printer's
! device registers are accessible.
!

    status = $crmpsc(                   ! Map I/O page to process.
                inadr=io_page_limits,
                retadr=io_page_limits,
                flags=sec$m_wrt OR sec$m_pfnmap OR sec$m_expreg,
                pagcnt=io_page_count,
                vbn=io_page_pfn );

    IF NOT .status THEN RETURN .status;

!
! Issue a connect to interrupt QIO to the line printer device. This
! connection will allow the program to control and handle interrupts
! from the device.
!

    status = $qio(                      ! Connect the process to the
                chan=.lpchan,           ! line printer device.
                func=io$_conintread,    ! Specify a read only buffer.
                astadr=lp_iodone_ast,   ! Specify an AST routine.
                p1=buffer_desc,         ! Specify a shared buffer.
                p2=entry_list,          ! Specify routine entry points.
                p3=cin$m_isr OR cin$m_cancel,
                                        ! Specify ISR, cancel routines.
                p4=lp_isr_ast,          ! Specify an interrupt AST.
                p6=5);                  ! Specify an AST count.

    IF NOT .status THEN RETURN .status;

!
```

```
! Ask user what file to print.
!

    status = lib$get_input( file_descr,
                            $descriptor('Name of file to be printed: '),
                            file_descr[0]);

    IF NOT .status THEN RETURN .status;



!
! Open and connect file.
!

    file_fab[fab$b_fns] = .file_descr[0];               ! Length of spec.

    status = $open(fab=file_fab);                       ! Open file.

    IF NOT .status THEN RETURN .status;

    status = $connect( rab = file_rab );                ! Connect file.

    IF NOT .status THEN RETURN .status;
!
! Get a record at a time until end of file. Surround record's contents
! with a linefeed and a carriage return.
!

    WHILE status = $get(rab=file_rab) DO
        BEGIN
        LOCAL
            inp,
            outp;

        outp = output_buffer[buf$l_startdata];  ! Target for first character
        CH$WCHAR_A( %CHAR(%X'A'), outp);        ! Start with a line-feed

        inp = record_buffer;

        !
        ! Load length of this output buffer in the buffer header. Then copy
        ! the contents of the input buffer to the output buffer. Translate all
        ! lower case alphabetics to upper case characters.
        !

        output_buffer[buf$w_charcount] = .file_rab[rab$w_rsz] + 2;

        DECR i FROM .file_rab[rab$w_rsz]-1 TO 0 DO
            BEGIN
            LOCAL
                char;

            char = CH$RCHAR_A( inp );
            SELECTONE .char OF
                SET
                [%C'a' TO %C'z']:       char = .char - %X'20';  ! Upcase
                TES;


            CH$WCHAR_A( .char, outp )
            END;

        CH$WCHAR_A( %CHAR(%X'0D'), outp );      ! Put CR at end.


!
! Send characters one at a time to the line printer. Before sending a
! character, see if the line printer is still in ready state. If not,
! set a timer to go off in one second, and go to sleep. When an AST
! occurs -- either because of a line printer interrupt, or because
! the timer runs out, the AST routine will wake the process up again.
!
! If the line printer is still in ready state, just send the next
! character.
!

        outp = output_buffer[buf$l_startdata];  ! Addr of output string
        csr = .io_page_limits + lp_csr_offset;  ! Addr of LP's CSR

        DECR i FROM .output_buffer[buf$w_charcount]-1 TO 0 DO
            WHILE 1 DO
                BEGIN
                BIND
                    devbits= csr[lp_csr]: VOLATILE SIGNED WORD;

                CASE SIGN(.devbits) FROM -1 TO 1 OF
                    SET
                [-1]:   RETURN ss$_devoffline;          ! Paper problem, maybe

                [1]:    BEGIN                           ! Output a character
                        csr[lp_dbr] = CH$RCHAR_A( outp );
                        EXITLOOP                        ! Back for next char
                        END;

                [0]:    !+
                        ! Line printer is not ready. See whether it's in
                        ! trouble, or just busy. If it's in trouble, stop
                        ! program with error status. Otherwise, just wait
                        ! until it comes ready again.
                        !-
                        BEGIN
                        output_buffer[buf$v_int] = true;        ! Interrupt expected
                        csr[lp_csr] = .csr[lp_csr] OR %X'40';   ! Enable LP interrupts
                        status = $setimr(                       ! Set a one second timer.
                            daytim=onesecond_delta,
                            astadr=lp_isr_ast);

                        IF NOT .status THEN RETURN .status;

                        $hiber;                         ! Go to sleep.
                        $cantim()                       ! Cancel timer request
                        END
                    TES
                    END
        END;                            ! End $GET loop

    IF .status NEQ ss$_endoffile
    THEN
        RETURN .status;
```

## APPENDIX D

## REAL-TIME OPTIMIZATION CHECKLIST

This appendix lists suggestions that usually improve real-time program performance. There is no guarantee, however, that any suggestion is appropriate for all applications. You must consider the needs of each application and the overall system activity when you evaluate any suggestion.

1. Avoid costly operations in time-critical code. Costly operations include:

   a. File opens or extensions

   b. Mailbox creation

   c. Common event flag cluster creation

   d. Device allocation

   e. Error reporting

2. Avoid window turns on critical files. Suggestions:

   a. Use contiguous files

   b. Specify a large window size

3. Inhibit system paging. Specify parameter values to the SYSGEN utility to:

   a. Disable system code paging (SYSPAGING = 0)

   b. Disable paging of pageable dynamic pool (POOL_PAGING = 0)

   c. Specify a large system working set (SYSMWCNT)

   However, before adjusting any of the parameter values, read the explanation of the parameter and any cautions in the VAX/VMS System Manager's Guide.

4. Use the Queue I/O Request ($QIO) system service directly for I/O.

   a. Setting an event flag is the fastest means of signalling I/O completion

   b. Using an AST is more time-consuming

5. Global sections provide the lowest-overhead means of interprocess communication.

6. Waiting for an event flag and using hibernate/wake provide the fastest methods of interprocess signalling.

# INDEX

VAX/VMS
Real-Time
User's Guide
AA-H784A-TE

## READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Other (please specify)________________

Name________________ Date________________

Organization________________

Street________________

City________________ State________________ Zip Code
or
Country________________

Please cut along this line.

Do Not Tear - Fold Here and Tape

digital

No Postag
Necessary
if Mailed in t
United Stat

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here